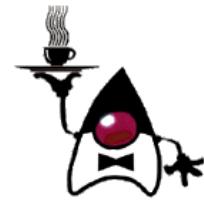


Java (V)
Systèmes logiciels embarqués :
mobiles et cartes à puce

Claude VILVENS

claude.vilvens@hepl.be

Sommaire



Introduction

XXVII. Les mobiles sous Java : introduction à J2ME

1. Le mondes des mobiles	
1.1 Les grandes catégories de mobiles	3
1.2 Les connexions sans fil	4
2. Différents OS et différentes plates-formes de développement	4
3. Le WAP	
3.1 Des mobiles pour Internet	5
3.2 L'architecture du WAP	5
3.3 Les couches du WAP	6
4. La technologie J2ME	
4.1 Une architecture assez complexe	7
4.2 Le panorama des architectures Java	9
5. La plate-forme de développement de base : CLDC et MIDP	
5.1 La configuration CLDC	10
5.2 Le profil MIDP	11
5.3 Le cycle de vie d'une MIDlet	11
5.3 L'interface utilisateur selon MIDP	12
6. Une MIDlet basique	
6.1 L'exemple classique "Hello world !"	13
6.2 L'afficheur Display	14
6.3 Le formulaire	14
6.4 La mise en place d'une zone de texte	16
6.5 La gestion des boutons de commande	17
6.6 Les classes et interfaces de base de CLDC/MIDP	18
7. Le Sun Java Wireless Toolkit	
7.1 Présentation	20
7.2 Un projet basique conforme aux JSRs	21
7.3 L'empaquetage du projet et son obfuscation	23
7.4 L'exécution de la MIDlet	24
7.5 Installation d'un projet MIDlet sur sa cible	26
8. Un exemple de mobile	
8.1 Présentation du mobile palmOne	31
8.2 Une JVM pour mobile	31
8.3 Le test de l'application basique	33
9. Une MIDlet avec un menu-liste	
9.1 Un système de menu classique	36
9.2 La mise en place d'une liste	36

9.2 La mise en place d'une liste	
10. Netbeans et son Mobility Pack	
10.1 Génération d'une MIDlet basique	41
10.2 Utilisation des outils de développement	45
10.3 Utilisation de la MIDlet générée avec le Sun Java Wireless Toolkit	51
11. Une MIDlet à formulaire élémentaire placée sur un serveur Web	
11.1 Une MIDlet à formulaire avec Netbeans	54
11.2 Une MIDlet sur un serveur Web	61
12. Un second exemple de mobile	
12.1 Le téléphone tactile Htc Touch	63
12.2 La synchronisation et Activesync	63
12.3 Le déploiement d'une MIDlet par câble	64
12.4 Le déploiement d'une MIDlet par OTA	65
13. Un formulaire plus élaboré	
13.1 La construction du formulaire	67
13.2 L'interaction avec le formulaire	69
14. Une application de gestion de contacts : GUI de création	72
15. Une application de gestion de contacts : enregistrement des contacts	
15.1 Le stockage des données en J2ME et la classe RecordStore	77
15.2 L'enregistrement dans un RecordStore	77
16. Une application de gestion de contacts : liste des contacts	
16.1 La lecture des enregistrements d'un RecordStore	81
16.2 L'affichage des contacts	82
17. Une application de gestion de contacts : effacer tous les contacts	84
18. Les communications réseaux	
18.1 Le framework GCF	86
18.2 La méthode open de Connector	87
18.3 Les interfaces Connection de base	87
18.4 Les connexions HTTP	88
18.5 Les connexions TCP	88
18.6 Le diagramme de classe UML des interfaces et classes réseaux	89
19. Une application réseau J2ME-J2SE basée TCP et HTTP	
19.1 Description de l'application	91
19.2 Analyse de l'application	93
19.3 Le thread pour HTTP	94
19.4 Le thread pour TCP	95
19.6 L'exécution de la MIDlet sur le mobile	99
20. Une MIDlet authentifiée	
20.1 Le mode opératoire	101
20.2 La signature de la MIDlet	101
20.3 L'installation du certificat sur le mobile	105
20.4 L'installation de la MIDlet sur le mobile	108

XVIII. Un autre Java embarqué : Google Android

1. Un framework pseudo-Java pour mobiles	110
2. Le modèle en couches d'Android	
2.1 Le schéma bloc	111

2.2 La couche Linux Kernel	111
2.3 La couche librairies	111
2.4 La couche Java	112
2.5 La couche Application Framework	113
3. La machine virtuelle Dalvik	
3.1 Une machine orientée registres	114
3.2 Un bytecode particulier	114
3.3 Le fichier de distribution apk	116
4. Les composants applicatifs Android	117
5. Le cycle de vie d'une activité Android	120
6. Le cycle de vie d'un service Android	121
7. Une application Android de base	122
8. Le plugin Android pour Eclipse	123
9. Développement d'une application Android basique avec Eclipse	
9.1 Génération d'une application basique	129
9.2 Les ressources générées	131
9.3 Le manifeste de l'application	133
10. Exécuter une application sur un mobile virtuel	134
11. Installation d'une application sur un mobile	
11.1 Installation des drivers du mobile Android	138
11.2 Déploiement automatique de l'application depuis Eclipse	139
11.3 L'Android Debug Bridge (ADB)	142
11.4 Déploiement manuel de l'application	144
12. Le développement manuel d'une application Android	148
13. Une application avec vue explicite	151
14. Les communications réseaux	155

XXIX. Les cartes à puce sous Java

1. Cartes magnétiques et cartes à puces	159
2. L'anatomie matérielle et logicielle d'une smart card	
2.1 L'aspect général	160
2.2 Les composants du circuit intégré	161
2.3 Le système d'exploitation des smart cards	162
2.4 Les potentialités en matière de sécurité et d'authentification	162
2.5 La sécurité des paiements par cartes à puces	162
3. La communication entre une carte à puce et un ordinateur	
3.1 Les dispositifs de lecture	163
3.2 La connexion et la déconnexion	163
3.3 Le protocole de communication	164
4. Diverses spécifications de smart cards	166
5. La spécifications PC/SC d'utilisation d'une carte à puce	
5.1 Le vocabulaire de la spécification PC/SC	167
5.2 La structure globale de la spécification	169
6. La technologie Java Card	
6.1 Java avec peu de mémoire	170
6.2 Le sous-ensemble de Java pour les smart cards	170
6.3 La plate-forme Java Card	171

7. L'API Java Card	
7.1 Les 4 packages	172
7.2 Les classes d'exception	173
8. L'exécution d'une applet pour Java Card	
8.1 Le convertisseur	174
8.2 Le téléchargement et l'exécution de l'applet	175
9. L'identification d'une applet	175
10. Les méthodes de base d'une applet pour Java card	175
11. L'exemple classique du e-commerce : le porte monnaie électronique	176
12. La construction de l'applet porte-monnaie	
12.1 Les fonctionnalités	177
12.2 L'algorithme de sécurité	177
12.3 La définition de l'AID	178
12.4 Les commandes de la smart card	178
12.5 La définition des codes d'instructions	179
12.6 L'instanciation de l'applet	180
12.7 La méthode de réponse à la sélection	183
12.8 La méthode de reconnaissance des commandes	183
12.9 L'envoi et la réception des données	187
12.10 Les méthodes de traitement des commandes	188
12.11 La méthode de désélection	192
13. Les outils de développement Java	196
14. La mise au point de l'applet porte-monnaie	
14.1 La compilation	196
14.2 La conversion en fichier CAP	197
14.3 L'émulateur de lecteur de cartes	198
14.4 Les scripts APDU	199
14.5 L'installation de l'applet sur la carte	200
14.6 Le test de l'applet	205
15. La simplification : l'Open Card Framework	
15.1 L'objectif	208
15.2 L'architecture Open Card Framework	208
15.3 La programmation d'un CardService	210
15.4 La programmation d'une CardServiceFactory	218
15.5 L'utilisation d'une CardServiceFactory et d'un CardService	220
16. Remplacer l'émulateur par un lecteur série	223
17. Le développement avec le simulateur Gemplus	
17.1 Les composants logiciels de base	226
17.2 Les variables d'environnement	227
17.3 Le développement avec simulateur : GemXpresso Simulator	227
17.4 Le JCardManager	229
17.5 Le déploiement d'une applet de base	231
17.6 Le déploiement d'une applet	233
17.7 Le déploiement d'un card service	238
18. La simplification (bis) : l'API smartcardio et GlobalPlatform	
18.1 L'API javax.smartcardio	242
18.2 Le déploiement avec GlobalPlatform/GPj	244
19. En conclusion : la chaîne de sécurité des plates-formes Java	246

XXX. La carte d'identité électronique belge

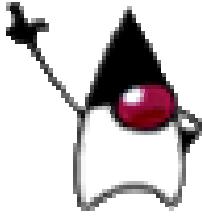
1. Une carte à puce pour carte d'identité	247
2. Les informations contenues dans une carte d'identité	247
3. Les informations cryptographiques	
3.1 Les trois niveaux de certificats de l'identité électronique belge	248
3.2 Les ressources cryptographiques disponibles	248
4. Les deux services fondamentaux	
4.1 Le service d'authentification	249
4.2 La signature électronique	249
5. La vérification des certificats	250
6. Les lecteurs de cartes	252
7. Le middleware d'utilisation et l'enregistrement des certificats	253
8. Le magasin à certificats de Windows	255
9. Le principe des fonctionnalités cryptographiques de la carte d'identité	
9.1 L'authentification	257
9.2 La signature	257
10. L'accès des applications aux informations cryptographiques	
10.1 L'architecture générale	258
10.2 Le module PKCS#11	264
10.3 Le module PKCS#15	264
11. L'utilisation des certificats d'identité avec les e-mails (Thunderbird)	
11.1 Paramétrier son compte de messagerie	266
11.2 L'enregistrement du module PKCS#11	267
11.3 L'enregistrement des certificats	269
11.4 La configuration du mécanisme de signature	271
11.5 La signature d'un message	273
11.6 La réception d'un message signé	273
12. L'utilisation des certificats d'identité avec un browser (Firefox)	
12.1 L'enregistrement du module PKCS#11	276
12.2 L'enregistrement des certificats	278
13. L'API général de la carte d'identité belge	
13.1 La librairie de développement	279
13.2 Les fonctions générales d'initialisation et de terminaison	281
13.3 Les fonctions d'initialisation en Java	281
14. La fonctions d'identité	283
15. La sécurité	
15.1 Les principes généraux	286
15.2 Les informations de sécurité en Java	291
15.3 La récupération d'un objet certificat	294
16. Utilisation de la carte d'identité dans un système de keystores	
16.1 Un keystore PKCS#11	297
16.2 L'exportation de certificats du magasin de Windows	303
16.3 Un keystore PKCS#12	305
16.4 La vérification d'une signature avec un keystore PKCS#12	307
17. Le principe de l'utilisation de la carte d'identité pour l'authentification d'une MIDlet	310

18. La vérification d'un certificat par OCSP	
18.1 Description du protocole OCSP	314
18.2 Les classes OCSP de Bouncy Castle	318
18.3 La vérification des certificats d'un keystore PKCS#12	319

Ouvrages consultés

Annexe : Introduction au développement sous Eclipse

Introduction



Que de chemin parcouru depuis la découverte du principe de la machine virtuelle Java ! On se souviendra que tout a débuté avec un projet de domotique, pour ensuite aller vers le Web puis les applications classiques sur des postes clients ou serveurs : les plate-formes J2SE et J2EE prenaient leur envol.

Mais le XXI^e siècle s'est révélé être celui de l'informatique embarquée, non pas celle des satellites ou des robots industriels (cela existe déjà depuis longtemps), mais celle de Monsieur Tout-le-monde : **GSMs et agendas électroniques**, bien sûr, mais aussi **cartes à puce** devenus le sésame de tant de nos activités (même faire ses courses au supermarché ;-)).

Java ne pouvait ignorer cela (.NET non plus, d'ailleurs ...) et deux autres plateformes ont émergé : la plate-forme **J2ME** caractéristique des GSM et Pocket PC (les "mobiles"), et la technologie **Java Card**, à la cible évidente et devenue une plate-forme à part entière.

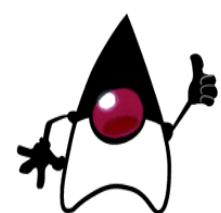
Nous découvrirons ici l'une et l'autre, en réalisant que le développement dans ces univers n'est plus aussi simple : il faut tenir compte du manque d'espace physique et donc limiter les possibilités au nom d'une performance raisonnable, il faut développer non pas directement sur le mobile mais d'abord sur une machine classique utilisant un simulateur, des problèmes de transfert peuvent ensuite apparaître, etc. Au point que des solutions propriétaires comme celle de **Google Android** sont à présent prises en compte par les professionnels, quitte à commettre un crime de lèse-"Java standard" ;-), justifiant ici une introduction à ce monde hérétique.

Nous domestiquerons donc tout cela pour ensuite passer à la découverte de la fierté nationale (ça existe encore ?) : **la carte d'identité électronique belge** !

Ainsi armés, nous pourrons en venir, dans le volume suivant, à quelques questions de e-commerce proprement dit, puisque nous disposerons de tout l'arsenal des 4 plates-formes Java !

Java Everywhere ! Ainsi parlait le Sage ;-)

Claude Vilvens



Reprenons à nouveau le fil de nos idées ! Les chapitres sont numérotés en poursuivant la séquence du volume IV. Nous commençons donc par le chapitre XXVII qui va nous plonger immédiatement dans le monde des mobiles ...

XXVII. Les mobiles sous Java : introduction à J2ME



*Vivez, si m'en croyez, n'attendez à demain.
Cueillez dès aujourd'hui les roses de la vie.*

(P. de Ronsard, Sonnets pour Hélène)

Qui ne connaît les "mobiles", terme générique désignant aussi bien les GSMs que les PDAs ou les Smartphones ? Nous allons découvrir ici les rudiments de base de leur utilisation selon Java :-o ...

1. Le monde des mobiles

Les dispositifs mobiles programmables constituent une famille populeuse et diversifiée. Tous sont basés sur l'idée de la communication réseau sans fil. On a coutume de distinguer quelques familles, mais en ces temps où l'on ne respecte plus rien ;-), les transfuges et hybrides sont nombreux !

1.1 Les grandes catégories de mobiles

1) Les **PDA** (Personal Digital Assistant – "Assistant personnel" en français) se présentent sous la forme d'un boîtier de la taille d'une calculatrice de poche. Il comporte un processeur (tout à fait performant), une mémoire (loin d'être minuscule : elle peut atteindre une taille du Go), souvent un clavier incorporé et un écran tactile que l'on peut utiliser avec un stylet, en faisant ainsi simultanément un organe d'entrée et de sortie. On parle encore aussi de **Pocket PC** pour ce genre d'appareils lorsqu'ils fonctionnent sous le système d'exploitation du même nom édité par Microsoft.

2) Les téléphones mobiles ou **GSM** (Global System for Mobile communication) : faut-il vraiment les présenter ? Depuis leur apparition fin des années '90, ils ont beaucoup évolué au point de vue taille, résolution de l'écran, types de communications. Nonobstant leur succès, ce sont pourtant les dispositifs mobiles qui offrent le moins de possibilités.

3) Les **Smartphones** est une espèce d'enfant naturel dont les parents seraient un GSM et un PDA : ce genre de téléphone portable intègre des fonctions avancées de PDA, comme la navigation Web ou Bluetooth. Les possibilités sont très nettement supérieures à celles d'un GSM classique.

4), Les **tablettes électroniques**, dont l'iPad d'Apple est l'archétype, sont encore plus évoluées : orientées vers le multimedia (films, musiques, jeux, mais aussi livres, journaux, magazines) et vers l'Internet et le courrier électronique; leurs possibilités techniques et leur poids (moins d'un kg) les palcent entre les smartphones et les PC portable.



1.2 Les connexions sans fil

Comme leur nom l'indique, les mobiles permettent de réaliser des connexions réseaux sans être entravés par des connexions filaires : nous sommes dans le monde des réseaux sans fil, l'information étant véhiculée par des ondes électromagnétiques (principalement radios ou infrarouges). Selon leur portée, on peut distinguer diverses catégories de réseaux sans fil :

1) les WPAN (Wireless (= portable) Personal Area Network) : il s'agit des réseaux personnels sans fil dont la portée est limitée à quelques dizaines de mètres. On range dans cette catégorie :

- ◆ Bluetooth : très économique en énergie, cette technologie permet des liaisons limitées à une trentaine de mètres avec un débit approchant le Mbps; son utilisation la plus connue est le couplage d'un PDA avec un récepteur GPS;
- ◆ la liaison infrarouge : popularisée par les télécommandes, cette technologie se limite à quelques mètres pour un débit de plusieurs Mbps.

2) les WLAN (portable Local Area Network) : cette fois, la portée est de l'ordre de 100 m. La technologie la plus connue est Wi-Fi (WIreless FIdelity), avec des débits de l'ordre de 50 Mbps.

3) les WMAN (portable Metropolitan Area Network) : encore connue sous le nom de Boucle Locale Radio (BLR), cette technologie est surtout celle des opérateurs de télécommunications avec des portées de l'ordre de 5 à 10 km pour des débits pouvant atteindre 10 Mbps.

4) les WWAN (portable Worldwide Area Network) : on se trouve ici dans le domaine des téléphones mobiles. Se rangent dans cette catégorie :

- ◆ le réseau **GSM** : déjà évoqués, ils comportent une carte SIM pour l'identification de l'utilisateur et sont tous identifiés par un numéro de série IMEI; les débits sont de l'ordre d'une dizaine de Kbps;
- ◆ le **GPRS** (General Packet Radio Service) : extension du GSM, il apporte des débits plus élevés (jusqu'à 40 Kbps) et ajoute la transmission par paquets;
- ◆ l'**UMTS** (Universal Mobile Telecommunications Service) : qualifiée de technologie mobile de 3^{ème} génération (le GSM est la 2^{ème}), il atteint des débits de l'ordre de 300 Kbps et est donc mieux adapté aux transferts multimedia.

2. Différents OS et différentes plates-formes de développement

Les systèmes d'exploitation des mobiles sont évidemment divers, mais les plus répandus sont

- ◆ **Google Android** : cette technologie propriétaire donc dédiée à une architecture bien précise utilise comme langage de développement Java ... mais non standard; en fait, Android est un framework combinant un système d'exploitation pour mobile et un SDK permettant le développement d'application pour tout mobile géré par cet OS; ce framework est à l'heure actuelle en pleine ascension et sert de référence pour les autres;
- ◆ **Windows Phone** : héritier de Windows CE et Windows Mobile, cet OS est en pleine évolution; les applications y trouvent sur une plate-forme .NET allégée avec C# comme langage de développement privilégié.
- ◆ **Blackberry OS** : proposé par Research In Motion (RIM), avec un navigateur mobile qui utilise un moteur WebKit offrant un bon rendu des pages Web et offrant la possibilité de naviguer par onglets ou d'utiliser des gestuelles tactiles pour zoomer dans une page; on y développe avec un Blackberry Java Plug-in ou en HTML 5.
- ◆ **iOS** : à la base, c'est le système d'exploitation du célèbre iPhone d'Apple; mais il a ensuite été utilisé pour l'iPad, l'iPod touch et Apple TV; à nouveau, l'interface utilisateur est très

fluide et fait la part belle aux gestuelles tactiles ("*multi-touch gestures*"); on y développe en Objective C ou en HTML 5.

♦ **Symbian** : c'est le système d'exploitation de base des smartphones Nokia, même si ce constructeur a utilisé Windows Phone pour ses produits les plus récents; le gros intérêt de et OS est que l'on peut y utiliser différents langages de programmation, comme C++, Python, Ruby, HTML 5 et ... Java ME.

♦ **HP web OS**, successeur de **Palm OS** : la société PalmOne construit des PDAs et des smartphones gérés par Palm OS, tandis que PalmSource développe le software correspondant - mais Palm OS est aussi utilisé par d'autres constructeurs, comme Sony ou Samsung et, surtout, Palm a été rachetée par HP qui se déploie sur ses machines; cet OS est un multi-tâches basé sur Linux.

Ce qui nous intéresse ici est bien sûr le développement d'application pour mobiles. Parmi les solutions de développement de type Java, **Java 2 Micro Edition (J2ME)** est évidemment le point de départ, mais d'autres voies sont possibles, comme SuperWaba, qui vise plus particulièrement les PDAs et les smartphones, et bien sûr le Java propriétaire **d'Android**. Mais au-delà des systèmes d'exploitation, et avant d'investiguer ces deux frameworks Java-mobiles, on ne peut passer sous silence à l'architecture des communications sans fil dédiées à Internet. L'architecture de base est la suivante ...

3. Le WAP

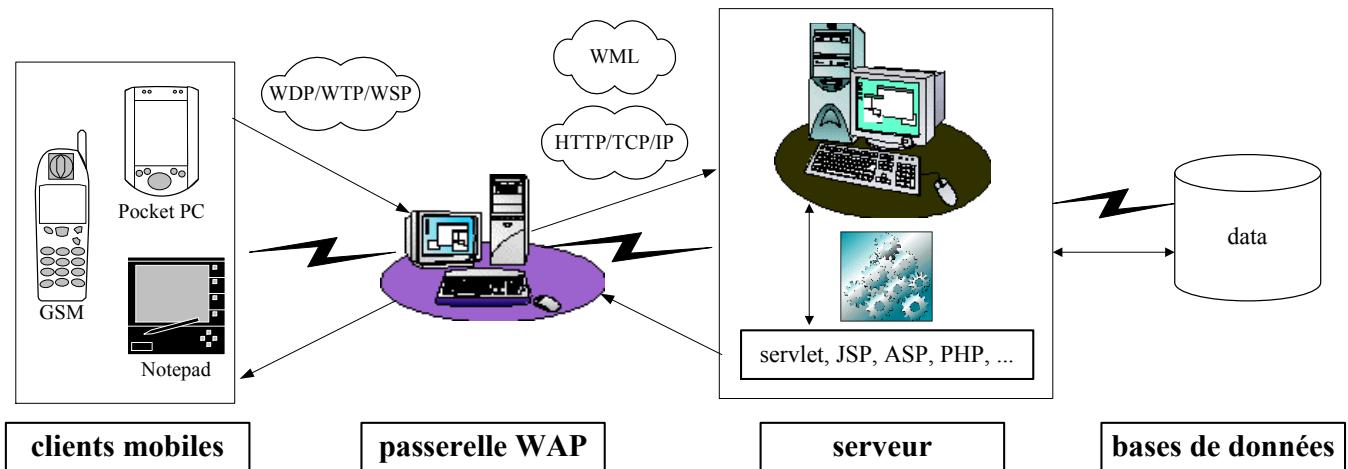
3.1 Des mobiles pour Internet

Dès que les mobiles ont été suffisamment répandus est apparue l'idée de les utiliser pour accéder à Internet : on a ainsi commencé à parler de services Internet "nomades". Si l'idée est séduisante, il n'en reste pas moins qu'il faut tenir compte des restrictions engendrées dans un tel contexte par la relative étroitesse de la bande passante et les possibilités plus réduites des terminaux mobiles par rapport aux terminaux fixes. Un protocole spécifique à ce type de communication a donc du être mis au point : le **WAP (portable Application Protocol)**.

Ce protocole se présente comme bâti sur la couche de transmission de données évoquée ci-dessus, comme GPRS ou UMTS. Il définit aussi dans la foulée le langage **WML** (**portable Markup Language**) qui permet de décrire les documents manipulés par le WAP et affichables sur un terminal mobile ainsi qu'un langage de script nommé **WMLScript**. Les spécifications du WAP sont rédigées par le WAP Forum, consortium indépendant des sociétés de télécommunication – il s'agit donc d'une espèce d'équivalent du W3C pour le Web classique.

3.2 L'architecture du WAP

L'architecture mettant en œuvre un serveur WAP suit un modèle 3 tiers : le tier intermédiaire est une passerelle réalisant l'interface entre le mobile (qui constitue le 1^{er} tier) et le serveur applicatif WAP (qui est le 3^{ème} tier) fournissant les services que le mobile veut atteindre. Eventuellement, un 4^{ème} tier complète le 3^{ème} en la personne d'un SGBD. Donc :



Le cheminement d'une requête est alors le suivant.

- 1) Le mobile se connecte à la passerelle à l'aide d'un numéro de téléphone et lui transmet la requête au moyen de cette connexion sans fil; celle-ci respecte les protocoles WDP-WTP-WSP.
- 2) La passerelle transmet ensuite cette requête au serveur WAP au moyen d'une connexion filaire classique http/TCP/IP ou sous la forme WML assez voisine de celle de HTTP.
- 3) Le serveur applicatif construit la réponse à la requête au format WML; pour ce faire, il peut exécuter, par exemple, un CGI ou une servlet ou un ASP ou un JSP ou un script PHP - donc, comme un serveur Web classique, si ce n'est que la servlet, par exemple, construit un document WML au lieu d'une page HTML.
- 4) Le serveur transmet le résultat à la passerelle (transmission filaire) qui, après avoir compacté cette réponse pour le rendre plus appropriée à la faible bande passante du réseau wireless, la renvoie au mobile (transmission sans fil).

3.3 Les couches du WAP

Le protocole WAP est en fait composé de 5 couches qui forment, en quelque sorte, l'équivalent des couches Application et Transport du modèle TCP/IP :

modèle WAP		équivalent modèle TCP/IP
couche WAE (<i>Wireless Application Environment</i>)	couche application	
couche WSP (<i>Wireless Session Protocol</i>)	couche session	couche application
couche WTP (<i>Wireless Transaction Protocol</i>)	couche transaction	
couche WTLS (<i>Wireless Transport Layer Security</i>)	couche sécurité	
couche WDP (<i>Wireless Datagram Protocol</i>)	couche transport	couche transport
couche GSM, GPRS, SMS, ..	couche opérateurs	couche réseau
couche physique	couche physique	couche physique

- 1) La couche application comporte le **WML**, langage qui, à l'aide de ses tags, indique comment le document doit être présenté sur un terminal mobile; ce langage possède une DTD de type XML. On y considère qu'un document est constitué de cartes (au sens d'un jeu de cartes) représentant chacune un composant comme un texte, un bouton, etc. Un document WML ressemble par exemple à ceci :

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">

<wml>
  <card id="carte1" title="INPRES-ISIL">
    <p align="center">
      <big>InPrES</big>
      <br/> Les kings de l'informatique
    </p>
  </card>
</wml>
```

Le mobile recevant un tel document réalisera un affichage du type :

InPrES-ISIL

Les kings de l'informatique

A ce niveau application se trouve aussi le **WTA** (Wireless Telephony Applications), ensemble d'interfaces permettant de développer des interfaces téléphoniques. Tout mobile WAP est supposé disposer d'une WMLUA (WML user-agent) et d'un WTAUA (WTA user-agent).

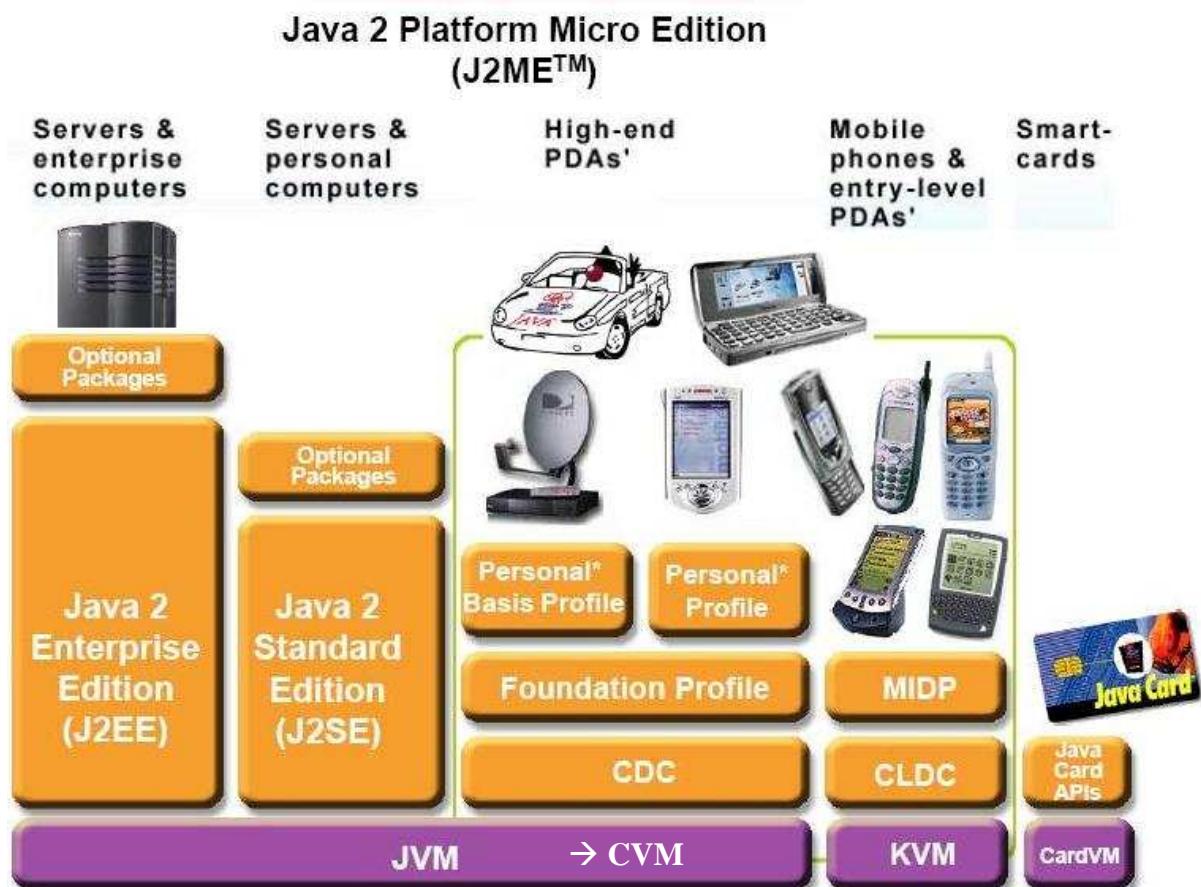
- 2) La couche session WSP peut utiliser un protocole orienté connexion au-dessus de la couche transaction ou un protocole non orienté connexion qui agit alors directement sur la couche transport. Elle comporte les fonctionnalités de http.
- 3) La couche transaction WTP rappelle un peu TCP puisqu'elle gère les ACK (avec ou sans, half ou full duplex).
- 4) La couche sécurité est basée sur SSL.
- 5) La couche transport WDP constitue l'interface avec les protocoles de transmission de données utilisés par les opérateurs de télécommunication. Elle rappelle un peu la couche UDP pour les connexions classiques.

Tout séduisant qu'il soit, le WAP n'est pas le seul contexte permettant de construire des applications. Si l'on se tourne vers une solution Java (ce qui sera bien sûr notre cas), le standard est bien sûr J2ME.

4. La technologie J2ME

4.1 Une architecture assez complexe

Alors que les technologies Java Card, J2SE et J2EE sont construites sur un seul système en couches, la technologie J2ME est par contre "à géométrie variable". En fait, il n'y a rien d'étonnant à cela : cette plateforme J2ME doit fournir un environnement pour des applications fonctionnant sur des dispositifs personnels bien différents, comme les PDAs, les téléphones portables, les smartphones et autres systèmes embarqués. Il serait donc fort maladroit de mettre sur le même pied des systèmes "de haut niveau", aux larges potentialités, et des systèmes plus frustes, aux fonctionnalités plus basiques. Si donc J2ME fonctionne sur une **machine virtuelle** et un **ensemble d'APIs** (éventuellement structurés en plusieurs niveaux), il y a plusieurs machines virtuelles possibles et plusieurs APIs de même niveau envisageables. Le schéma global ci-dessous des plates-formes Java laisse apparaître synthétiquement cette spécificité :



a) Plusieurs **machines virtuelles** sont donc envisageables pour fonctionner dans un environnement de mobile, forcément limité en terme de mémoire, d'énergie et d'accès réseau (mais pas au point des limites de la carte à puce – voir plus loin) : outre la JVM classique (ce qui est peu probable), on peut utiliser une machine virtuelle appelée une "**CVM**"(Compact Virtual Machine = une JVM digne de ce nom, compacte et optimisée pour les mobiles) ou encore une "**KVM**" (KiloByte Virtual Machine = une JVM allégée, ne possédant donc pas toutes les fonctionnalités d'une JVM classique), ceci suivant le type de mobile sur lequel on va travailler. Plus précisément, la KVM

- ◆ ne gère pas les float et les double;
- ◆ n'admet pas les chargeurs de classes définis par le développeur;
- ◆ utilise une classe Object sans finalize();
- ◆ ne gère pas la réflexion;
- ◆ ne dispose des notions de groupes de threads et de thread démon.

b) Les APIs de bas niveau, noyau minimal pour développer, constituent ce que l'on appelle une **configuration**. Bien clairement, ces APIS correspondent à des opérations assez générales (comme l'accès à la mémoire ou au réseau) et dépendent de la machine virtuelle choisie. Les configurations J2ME les plus courantes sont :

- ◆ **CLDC** (Connected Limited Device Configuration), destinée aux PDAs simples et aux GSMs; elle fonctionne sur une KVM;
- ◆ **CDC** (Connected Device Configuration), utilisée sur des smartphones évolués ou, par exemple, sur des décodeurs de télévision; elle fonctionne sur une CVM.

c) Les APIs de niveau supérieur constituent un "**profile**" : un profil est dédié à un dispositif donné et aux dispositifs qui lui sont similaires, afin de les utiliser de manière optimale. Ces APIs traitent de la manière dont les applications sont exécutées (leur "cycle de vie"), l'interface utilisateur et les accès aux propriétés spécifiques du dispositif : par exemple, l'application peut être utilisée sur un GSM – l'utilisateur agit avec les touches – ou sur un Palm – l'utilisateur agit avec son stylet. Le profile est donc capable d'adapter la même application à des configurations similaires dans le principe mais différentes par l'agencement des moyens d'interaction de l'utilisateur :



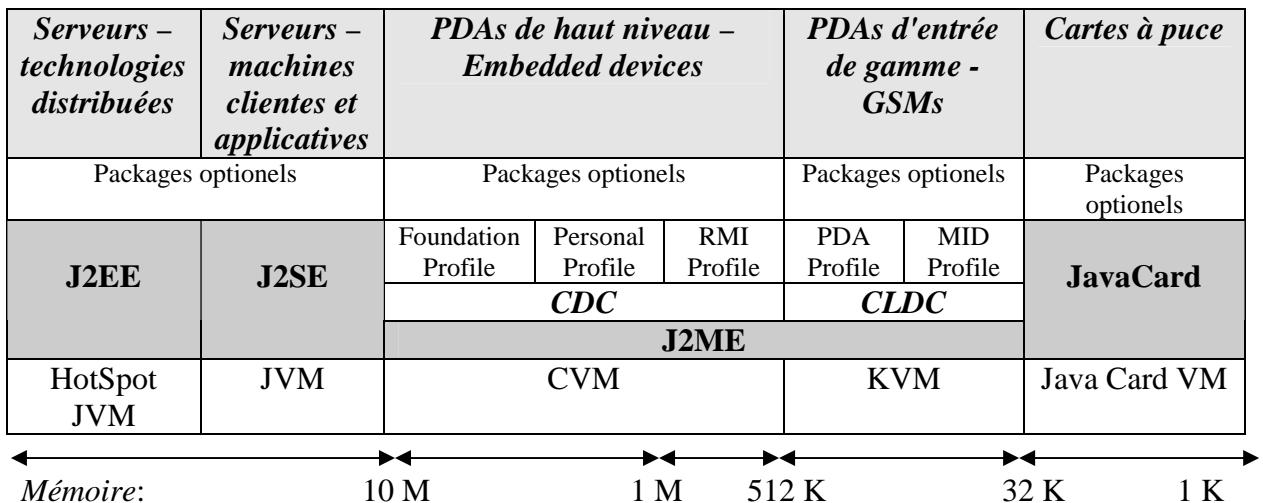
Avec la configuration sous-jacente, un profil fournit donc un environnement d'exécution complet pour une catégorie spécifique de dispositifs mobiles. Ainsi, le profil **Foundation** est destiné à la configuration CDC et implémente l'ensemble des fonctionnalités de J2SE. Le profil le plus répandu basé sur la configuration CLDC est **MIDP (Mobile Information Device Profile)** : il offre un environnement Java de développement aux téléphones cellulaires et autres dispositifs similaires. Il prend en charge un nombre limité des classes de J2SE et définit des classes d'entrée/sortie et d'interface spécialisées pour une configuration CLDC. Il suppose que le mobile utilisé dispose au minimum de

- ◆ 128 Ko de mémoire persistante pour l'implémentation de MIDP;
- ◆ 8 Ko de mémoire persistante pour les données persistantes;
- ◆ 32 Ko de mémoire volatile;
- ◆ un moyen d'interagir avec l'utilisateur;
- ◆ une connexion réseau quelconque.

d) Il arrive que l'on utilise encore des APIs complémentaires (correspondant à des packages supplémentaires) qui permettent l'utilisation de technologies supplémentaires comme les connexions aux bases de données, les messageries sans fil, le multimédia, BlueTooth et les Web Services.

4.2 Le panorama des architectures Java

En résumé, on peut donc distinguer les architectures Java suivantes :



Au sein de J2ME, on peut donc parler de différents "stacks". Schématiquement, nous travaillerons dans la suite sur le stack suivant :

profile MIDP
configuration CLDC
machine virtuelle KVM
système d'exploitation et hardware hôte

5. La plate-forme de développement de base : CLDC et MIDP

5.1 La configuration CLDC

La configuration CLDC est donc celle qui est destinée aux mobiles ayant une puissance de traitement, une mémoire et une capacité graphique limitées. Dans ces conditions, comme pour les Java Cards, mais en moins grave, il a été décidé de conserver pour les mobiles de base les packages Java fondamentaux en version allégée :

- ◆ **java.lang** : les types primitifs mais les réels flottants ne sont pas supportés, les classes Thread (mais sans les groupes), String et StringBuffer, Throwable et System, ainsi que l'interface Runnable;
- ◆ **java.io** : les flux classiques bytes (DataXXXStream) et caractères (Reader et Writer) ainsi que les flux tableaux ByteArrayXXXStream;
- ◆ **java.util** : les containers de base (Vector, Stack, Hashtable avec l'interface Enumeration) et les Calendar;

CLDC apporte un package spécifique nommé **javax.microedition.io**, qui fournit des classes pour gérer des connections génériques : Connection, StreamConnection, Datagram, DatagramConnection, etc.

CLDC modifie également le mécanisme de vérification du bytecode. En effet, le classique vérificateur de classes de J2SE-J2EE est bien trop gros pour pouvoir tenir sur un mobile du style de ceux visés par CLDC. Par conséquent, le gros du travail de vérification s'effectue sur la machine (classique) de développement après la compilation, au moment de l'empaquetage. C'est le rôle d'un programme particulier, le **prévérificateur** (preverify.exe qui se trouve dans NetBeans 6.*\mobility8\WTK2.5.2\bin dans le cas où l'on développe avec Netbeans), d'effectuer cette vérification. En fait, il ajoute des informations dans le fichier

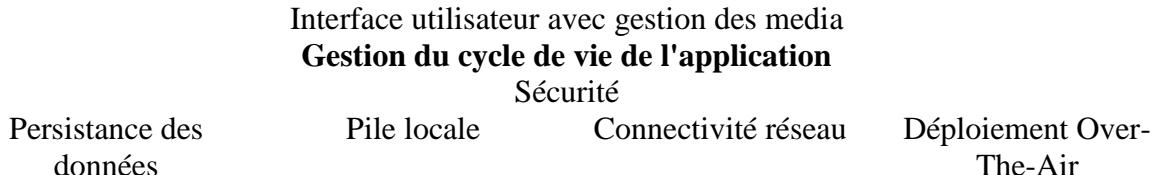
class, informations qui seront utilisées par la "vérification légère" qui sera effectuée sur le mobile. Il faut cependant remarquer que dans de telles conditions, une politique de sécurité utilisant du code certifié s'impose, puisqu'un hacker pourrait contourner beaucoup plus facilement l'étape de vérification.

5.2 Le profil MIDP

Son rôle est donc de fournir le support d'applications classiques dans le contexte des mobiles de type GSM et PDA. De telles applications sont appelées des **MIDlets**. Une MIDlet assure essentiellement la gestion

- ◆ de l'interface utilisateur (support d'écran tactile, méthodes graphiques de bas niveau dessinant pixel par pixel, utilisation d'images, mais au format .png seulement);
- ◆ du réseau, mais seulement selon le protocole HTTP;
- ◆ de la sauvegarde des données dans une base de données RMS (voir plus loin).

Schématiquement, on peut visualiser MIDP 2.0 ainsi :



Les packages que ce profil apporte en supplément sont :

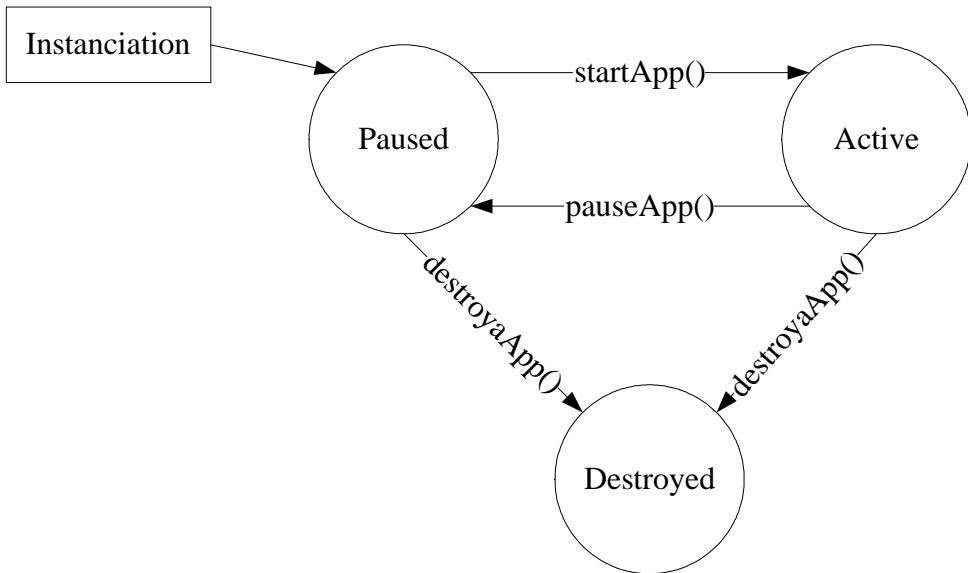
- ◆ **javax.microedition.MIDlet** : contient la classe **MIDlet** et fournit l'interface entre une MIDlet et son environnement;
- ◆ **javax.microedition.rms** : gère le stockage persistant des données;
- ◆ **javax.microedition.lcdui** : fournit les classes d'interface graphique utilisateur.

5.3 Le cycle de vie d'une MIDlet

Toute application doit hériter d'une classe abstraite **MIDlet** (du package `javax.microedition.MIDlet`) qui déclare et impose l'implémentation de trois méthodes qui reflètent son cycle de vie : à son instantiation, une MIDlet est dans l'état Paused; elle peut agir quand elle passe dans l'état Active et passe à l'état Destroyed quand elle se termine et libère les ressources qu'elle utilise. Les trois méthodes sont :

- public void **startApp()**
lance l'application (ou la relance après une pause);
- public void **pauseApp()**
suspend l'application, libère les ressources partagées (threads, connexions, ...);
- public void **destroyApp()**
libère toutes les ressources et termine l'application.

et schématiquement :



Ces méthodes sont appelées par le gestionnaire des applications sur le mobile utilisé : le **JAM** (Java Application Manager) ou l'**AMS** (Application Management Software).

De plus, la classe MIDlet fournit des méthodes qui permettent une interaction de l'application avec ce gestionnaire en lui envoyant une requête demandant :

- ◆ **NotifyPaused()** : de mettre l'application en pause;
- ◆ **ResumeRequest()** : de relancer une application après une mise en pause;
- ◆ **NotifyDestroyed()** : de fermer l'application (il faudra appeler `destroyApp()` auparavant puisque le gestionnaire ne le fera pas automatiquement si `NotifyDestroyed()` est appelée);
- ◆ **GetAppProperty()** : de fournir des informations concernant l'application.

5.4 L'interface utilisateur selon MIDP

On l'a dit, MIDP est conçu pour fonctionner aussi bien sur des GSMs que sur des Palm – désignons ces cibles par le terme de "terminal". Ces terminaux ont des points communs comme par exemple un petit écran; mais tous ne possèdent pas un système de pointage (stylet). Comme les applications MIDP doivent toujours intégrer les mêmes fonctionnalités quel que soit le terminal utilisé, il a fallu décomposer l'interface utilisateur en deux niveaux :

- ◆ l'API de bas niveau : il exploite **toutes les fonctionnalités du terminal**, donc donne accès direct à l'écran du terminal et aux événements associés aux touches et au système de pointage; les composants (**Canvas**) qui y sont utilisés sont évidemment moins portables;
- ◆ l'API de haut niveau : il n'exploite que **ce qui est portable**, donc fournit des composants aux fonctionnalités simples (**Form**, **List**, **Alert**, **Ticker**), sans accès direct à l'écran (les classes implémentant MIDP décideront de la manière de représenter les composants) ou au dispositif de touches ou de pointage (ces mêmes classes choisiront le mécanisme de gestion des saisies de l'utilisateur – par exemple, comme le GSM ne possède pas de stylet, ce sera son bouton multidirectionnel qui lui permettra de faire défiler l'écran d'affichage).

Pour implémenter une application minimale en restant au haut niveau, il nous faudra en définitive utiliser

- ◆ un objet de type **Display** pour assurer les affichages sur le mobile;
- ◆ un container pour y placer les composants de GUIs, le plus souvent de type **Form**;
- ◆ des composants assez classiques comme des objets **TextField**, **Gauge**, etc

Mais nous allons examiner cela en détails.

6. Une MIDlet basique

6.1 L'exemple classique "Hello world !"

Sans aucune originalité, nous allons entrer dans le monde CLDC/MIDP avec une application stéréotypée affichant "Hello world" sur l'écran du mobile. Elle ressemble à ceci :

HelloMobileWorld.java

```
import javax.microedition.MIDlet.*;
import javax.microedition.lcdui.*;

public class HelloMobileWorld extends MIDlet implements CommandListener
{
    private Display afficheur;
    private Form formulaire;
    private TextField zoneTexte;
    private Command exitCommand;

    public HelloMobileWorld ()
    {
        afficheur = Display.getDisplay(this);
        formulaire = new Form("Test de HelloWorld");
        zoneTexte = new TextField("", "Hello World !", 15, TextField.ANY);
        exitCommand = new Command("Exit", Command.SCREEN, 1);

        // ajout des composants au formulaire
        formulaire.append(zoneTexte);
        formulaire.addCommand(exitCommand);
        formulaire.setCommandListener(this);
    }

    public void startApp()
    {
        afficheur.setCurrent(formulaire);
    }

    public void pauseApp() { }

    public void destroyApp(boolean unconditional) { }
}
```

```
public void commandAction(Command c, Displayable s)
{
    if (c == exitCommand)
    {
        destroyApp(false);
        notifyDestroyed();
    }
}
```

Nous pouvons déjà constater que, comme prévu, notre MIDlet dérive de la classe **MIDlet** du package javax.microedition.MIDlet. Voyons plus loin.

6.2 L'afficheur **Display**

Notre MIDlet doit tout d'abord posséder comme variable membre un objet instance de la classe **Display** du package javax.microedition.lcdui. Comme son nom l'indique, il a pour rôle d'assurer les affichages et, pour ce faire, comporte les méthodes de récupération des propriétés du terminal et d'affichage d'objets divers sur celui-ci. Un tel objet, qui doit évidemment être unique pour la MIDlet, est obtenu au moyen de la méthode

```
public static Display getDisplay (MIDlet m)
```

qui permet d'obtenir l'objet **Display** associé à la MIDlet.

6.3 Le formulaire

a) Les éléments que la MIDlet va afficher doivent se trouver un container qui doit être une implémentation de la classe abstraite **Displayable**. L'application doit disposer au moins d'un tel objet, mais peut en fait en avoir plusieurs de disponibles, un seul étant actif parce qu'il a été sélectionné au moyen de la méthode :

```
public void setCurrent (Displayable nextDisplayable)
```

En fait, si elle permet de désigner le composant à afficher, son exécution peut prendre éventuellement un certain temps (un autre appel de **setCurrent()** pourrait donc faire oublier l'appel précédent, car les demandes de ce type ne sont pas mises en queue).

b) La librairie de J2ME ne fournit pas immédiatement des classes dérivées utilisables de **Displayable**. En fait, on trouve d'abord deux classes filles elles-mêmes abstraites :

- ◆ **Canvas** : elle sert de base aux classes gérant des événements de saisie de bas niveau, donc essentiellement actions sur des touches clavier ou des pointages pour choisir sur un écran tactile.
- ◆ **Screen** : elle correspond aux composants GUI classiques en apportant
 - la gestion des changements d'états des composants du GUI (comme la sélection dans une liste ou la modification d'un texte dans une zone d'entrée);

- le traitement des commandes venant du mobile lui-même, c'est-à-dire l'appui sur les différents boutons de contrôle de celui-ci.

c) Quatre classes dérivées implémentent effectivement la classe Screen :

- ◆ **Form** : le formulaire classique qui contient des items instance de l'une des classes dérivées de la classe abstraite Item : **ChoiceGroup**, **CustomItem**, **DateField**, **Gauge**, **ImageItem**, **Spacer**, **StringItem** et **TextField** – ce que font ces classes est bien clair;

Ces items sont ajoutés au formulaire au moyen de la méthode

```
public int append (Item item)
```

Le layout implicite est un layout qui gère les composants selon des lignes, en grandissant vers le bas avec scrolling éventuel. Son constructeur

```
public Form (String title)
```

ne réclame que le titre à placer au dessus du GUI.

- ◆ **TextBox** : le composant qui permet l'introduction et l'édition de texte au moyen d'une "fenêtre" avec un titre déterminé par le premier paramètre de son constructeur

```
public TextBox (String title, String text, int maxSize, int constraints)
```

Cette fenêtre va comporter une zone d'édition contenant un texte initial fourni par le deuxième paramètre.

- ◆ **List** : une liste classique qui implémente l'interface **Choice**. Son constructeur usuel est

```
public List (String title, listType, String[] stringElements, Image[] imageElements)
```

Le premier paramètre détermine évidemment le titre de la fenêtre tandis que la troisième désigne évidemment les chaînes de caractères constitutives de la liste. Le deuxième paramètre caractérise le type de liste désirée (classique, à choix exclusif, à choix multiples), en prenant comme valeur l'une des constantes définies dans l'interface Choice.

- ◆ **Alert** : une boîte de dialogue permettant l'affichage éventuellement accompagné d'une image ou d'un son et qui désactive le GUI jusqu'à l'écoulement d'un temps ou jusqu'à ce que l'utilisateur appuie sur un bouton de commande.

d) Pour notre exemple, ce donnera donc finalement que si notre MIDlet a une variable membre :

```
| private Form formulaire;
```

alors nous placerons dans le constructeur de la MIDlet

```
| formulaire = new Form("Test de HelloWorld");
```

tandis qu'au démarrage de l'application on provoquera l'affichage du formulaire par :

```
| afficheur.setCurrent(formulaire);
```

Evidemment, entre les deux, il faudrait placer au moins un composant dans le formulaire, par exemple donc une zone de texte.

6.4 La mise en place d'une zone de texte

Une application MIDP est destinée à des dispositifs aussi divers qu'un PDA ou qu'un GSM. Dès lors, on conçoit qu'il faut abandonner toute idée de disposition bien précise des composants graphiques formant l'interface utilisateur. En fait, MIDP les place les uns à la suite des autres. Si l'écran du dispositif n'est pas assez grand, une barre de défilement est automatiquement créée afin de pouvoir accéder aux éléments situés dans la zone non visible. A l'exécution, la MIDlet sera adaptée aux caractéristiques propres du dispositif mobile utilisé. Par exemple, le déplacement vertical de l'écran s'effectuera avec un stylet sur un PDA, mais sur un GSM, ce sera un bouton directionnel qui réalisera ce travail; ou encore, les commandes désignées au moyen du stylet sur le PDA seront plutôt associées aux touches sur le GSM.

Concrètement, le composant qui a été placé ici est une instance de la classe **TextField**, qui se trouve toujours dans le package javax.microedition.lcdui. Ce type de composant permet l'introduction et l'édition de texte. Son constructeur est

```
public TextField (String label, String text, int maxSize, int constraints)
```

Une instance de cette classe a donc pour but de faire apparaître sur l'écran du mobile, après avoir évidemment sélectionné la MIDlet, une "fenêtre" avec un titre déterminé par le premier paramètre. Cette fenêtre va comporter une zone d'édition contenant un texte initial fourni par le deuxième paramètre. Le troisième paramètre fixe bien entendu le nombre maximum de caractères de la zone, tandis que le quatrième permet de spécifier un filtre sur ce qui sera entré dans la zone (par exemple, ne laisser entrer que des nombres); une série de constantes de classe permettent de définir ces filtres :

```
public static final int ANY = 0;  
public static final int EMAILADDR = 1;  
public static final int NUMERIC = 2;  
public static final int PHONENUMBER = 3;  
public static final int URL = 4;  
public static final int DECIMAL = 5;
```

Ces flags peuvent être combinés avec d'autres constantes de classe comme **PASSWORD** ou **SENSITIVE**. Dans le cas le plus simple, on pourra entrer n'importe quoi et il suffira d'écrire :

```
| zoneTexte = new TextField("","Hello World !",15,TextField.ANY);  
| formulaire.append(zoneTexte);
```

Une dernière chose maintenant : le traitement des commandes de l'utilisateur ...

6.5 La gestion des boutons de commande

Dans un GUI classique, l'utilisateur déclenche les opérations, notamment, en appuyant sur des boutons de type "Ok" ou "Cancel". L'équivalent sur les mobiles est constitué par des boutons dont les intitulés sont assez variables de modèle à modèle. C'est le rôle de la classe **Command** de mettre en place un handler permettant de répondre à ces commandes. Une commande est toujours associée à un type qui est désigné par l'une des constantes de classe BACK, CANCEL, EXIT, HELP, OK, SCREEN, STOP ou ITEM. En fait, chaque dispositif place les commandes selon une disposition correspondant à ces types et qui lui est propre. Ainsi, une certaine marque de GSM dispose toujours sa commande OK sur le bouton le plus à gauche mais une autre marque, elle, choisira de toujours placer plutôt la commande CANCEL sur cette touche. Donc, en définissant une commande comme une commande "OK" ou une commande "CANCEL", le dispositif va la reconnaître et placer celle-ci selon une disposition qui lui est spécifique. Le constructeur le plus simple de la classe matérialisant ces commandes est :

```
public Command (String label, int commandType, int priority)
```

On peut voir qu'il suffit de fournir en 1^{er} paramètre le texte qui sera affiché lorsque le bouton dont le type est précisé en 2^{ème} paramètre sera sélectionné. Plus accessoirement, une commande possède également une priorité, un nombre de valeur faible désignant une commande importante. Dans le cas de notre MIDlet, on pourrait définir :

```
| Command exitCommand = new Command("Exit", Command.SCREEN, 2);
```

Cette commande sera associée au formulaire au moyen de la méthode héritée de Displayable :

```
public void addCommand (Command cmd)
```

donc ici :

```
| formulaire.addCommand(exitCommand);
```

Pour que cette commande soit traitée lorsqu'elle est invoquée, il faut encore lui définir un listener de commandes qui peut être un objet quelconque mais implémentant l'interface **CommandListener**. Cet interface comporte la méthode :

```
public void commandAction (Command c, Displayable d)
```

Le plus souvent, c'est l'application elle-même qui est le listener :

```
| public class HelloMIDlet extends MIDlet implements CommandListener
| { ... }
```

à condition, comme toujours, de s'enregistrer comme tel au moyen de la méthode héritée de Displayable :

```
public void setCommandListener (CommandListener l)
```

Donc ici :

```
| formulaire.setCommandListener(this);
```

Il reste dès lors à effectivement doter notre MIDlet de la méthode commandAction() qu'elle s'est engagée à posséder. Comme cette méthode sera le siège du traitement de toutes les commandes, il sera d'ores et déjà prudent de prévoir le test du type de la commande reçue :

```
public void commandAction(Command c, Displayable s)
{
    if (c == exitCommand)
    {
        destroyApp(false); notifyDestroyed();
    }
}
```

Comme on le voit, nous terminons simplement l'application, la méthode

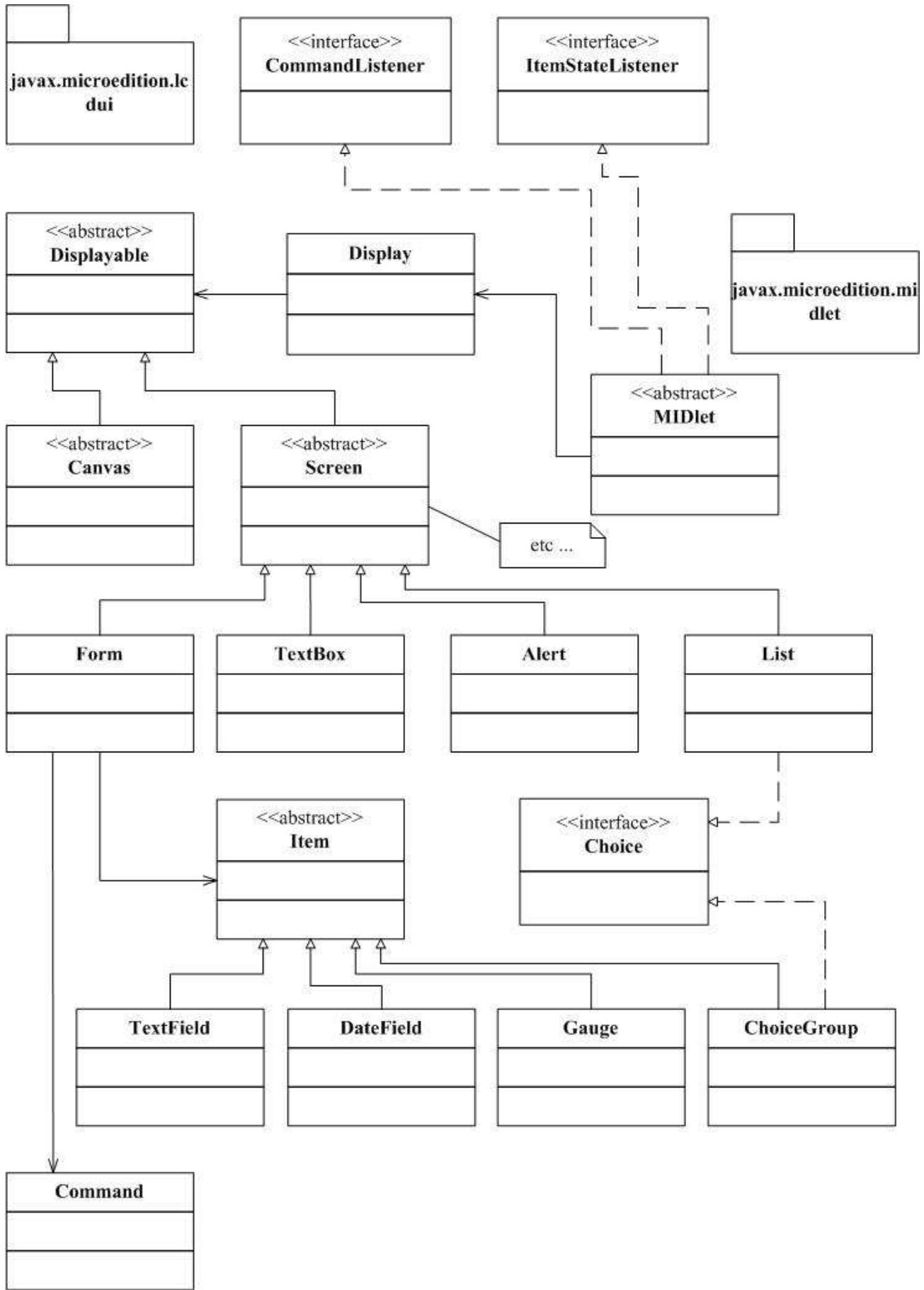
```
public final void notifyDestroyed ()
```

ne servant qu'à prévenir le software gérant le mobile que l'application passe à l'état Destroyed.

Fort bien ... mais comment tester cette MIDlet ?

6.6 Les classes et interfaces de base de CLDC/MIDP

Nous y venons. Mais on peut auparavabt visualiser dans le schéma UML suivant les relations unissant les principales classes permettant de créer des MIDlets avec GUIs pour J2ME selon CLDC/MIDP :



On peut encore y ajouter

- ◆ comme signalé plus haut, les classes **Spacer**, **CustomItem**, **StringItem** et **ImageItem**, dérivées d'Item et à la signification évidente;
- ◆ la classe **Ticker** qui matérialise un texte qui défile dans la partie supérieure de la zone d'affichage, dans une direction et à une vitesse qui est fixée par l'implémentation de J2ME sur le mobile utilisé; le constructeur se limite donc à

```
public Ticker (String str)
```

On le met en place avec la méthode

```
public void setTicker(Ticker ticker)
```

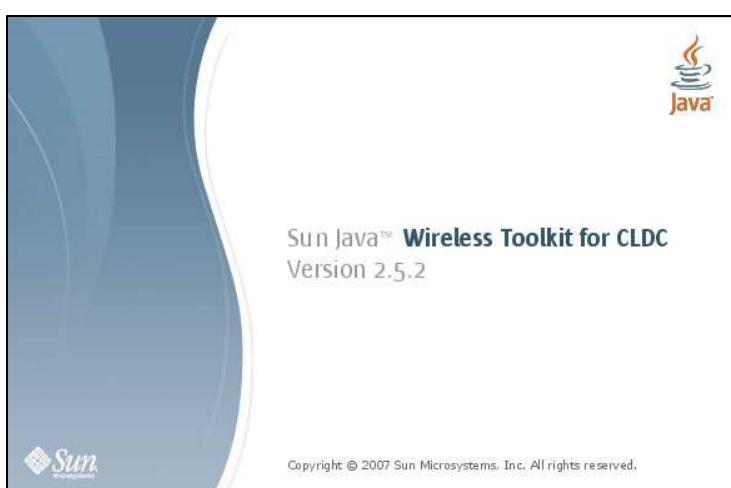
de Displayable.

7. Le Sun Java Wireless Toolkit

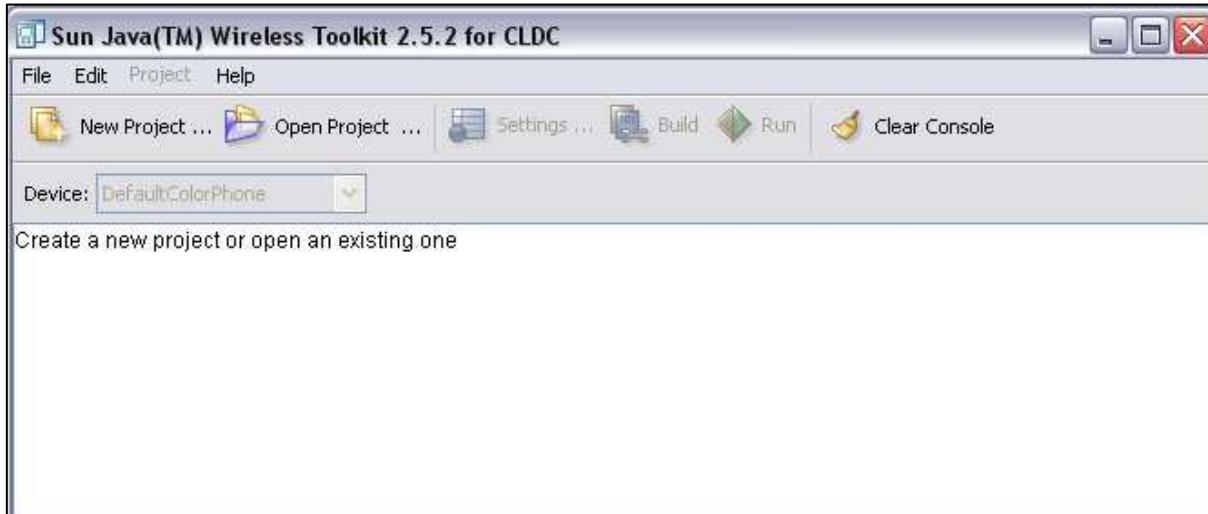
7.1 Présentation



Il n'est pas forcément simple de trouver un environnement de développement pour créer des applications mobiles. Mais Sun fournit un Sun Java Wireless Toolkit pour la configuration CLDC – l'installation est immédiate à partir d'un fichier sun_java_wireless_toolkit-2_5_2-windows.exe téléchargé sur le site de Sun (<http://java.sun.com/products/sjwtoolkit/>). Depuis peu, on trouve aussi sur le même site un Java Micro Edition SDK 3.0 qui ressemble fort à un sous-ensemble de Netbeans 6.5/6.7 ;-). Une fois l'installation réalisée, on peut faire démarrer l'application depuis le menu Démarrer :



On se retrouve devant un environnement assez prévisible pour un développeur :



7.2 Un projet basique conforme aux JSRs

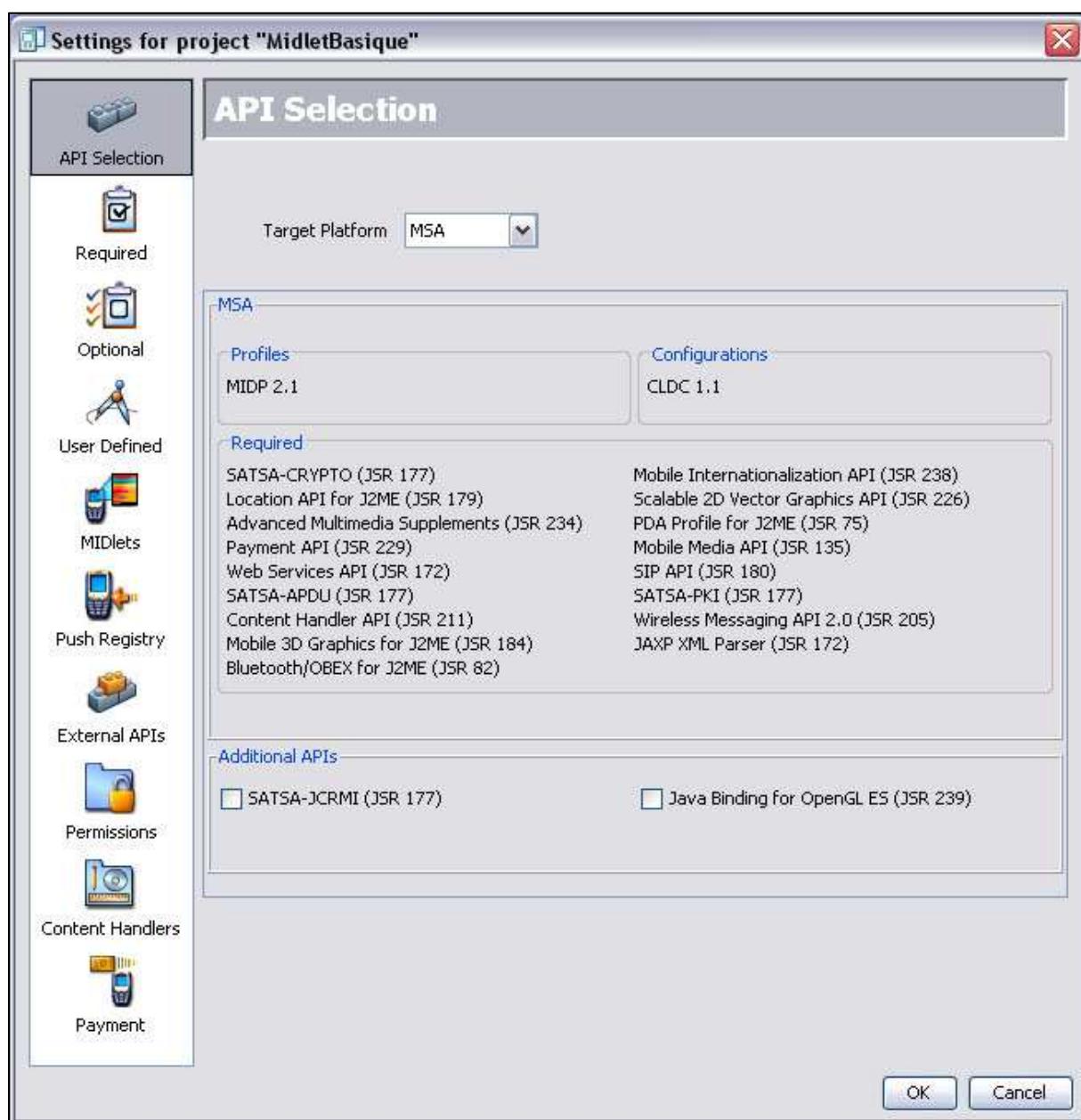
Bien sûr, nous créons un nouveau projet (appelons-le MidletBasique) dans l'intention d'y placer notre MIDlet :



L'appui sur le bouton de création a pour effet de faire apparaître un paramétrage assez complexe du projet, faisant intervenir toute une série de JSR. Pour rappel, une **JSR** (Java Specification Request) est un ensemble de spécifications concernant le plus souvent des APIs dédiés à une technologie logicielle ou matérielle. L'objectif est clairement de mettre en place la compatibilité, l'interopérabilité et la complémentarité entre les différents produits logiciels d'un domaine donné, dans notre cas la technologie J2ME des mobiles. Dans ce dernier contexte, les JSR les plus importantes sont :

- ◆ JSR-30 : la spécification de Connected, Limited Device Configuration (**CLDC** 1.0);
- ◆ JSR-118 : l'extension actuelle de la JSR-37, qui définit le Mobile Information Device Profile (**MIDP**) et qui se base sur la JSR-30; ici, il s'agit de MIDP 2.0;
- ◆ JSR-120: le Wireless Messaging API qui définit les accès à des ressources par réseaux wireless (ex: **WMA** 1.1);
- ◆ JSR-135 : le Java Mobile Media API qui régit l'accès et le contrôle aux ressources multimedia indépendamment de leurs tailles (scalability) (ex: **MMAPI** 1.1);

La JSR-185, la **JTWI** (Java Technology for the Wireless Industry), intègre les 4 précédentes en considérant les deux premières comme obligatoires, la troisième est optionnelle (mais si elle est suivie, elle l'est dans sa version 1.1) et la dernière comme la configuration minimale.



On peut voir que d'autres JSR sont référencées, comme par exemple la SATSA (Security and Trust Services API – JSR 177) qui régit les questions de sécurité et particulièrement d'authentification.

Acceptons les valeurs par défaut (provisoirement peut-être) : l'EDI répond par des informations sur les répertoires qu'il utilisera. Ainsi, sur une machine Windows classique, cela donnerait ceci :

```

Creating project "MidletBasique"
Place Java source files in "C:\Documents and Settings\Vilvens.ADMINISTRATION\
j2mewtk\2.5.2\apps\midbas\src"
Place application resource files in "C:\Documents and Settings\Vilvens.ADMINISTRATION\
j2mewtk\2.5.2\apps\midbas\res"
Place application library files in "C:\Documents and Settings\Vilvens.ADMINISTRATION\
j2mewtk\2.5.2\apps\midbas\lib"
Settings updated
Project settings saved

```

Il s'agit donc à présent de coder le source de notre MIDlet. Ceci peut se faire avec n'importe quel éditeur – l'EDI n'en fournit pas un ☺ ! Disons que nous créons le fichier HelloMobileWorld.java par exemple avec un utilitaire sympa d'édition comme jEdit et que nous y encodons nos instructions Java, puis, comme indiqué dans l'EDI, nous le sauvons dans

C:\Documents and Settings\Vilvens.ADMINISTRATION\j2mewtk\2.5.2\apps\MidletBasique\src

Nous pouvons alors lancer le processus de Build (appui sur le bouton correspondant), c'est-à-dire de compilation :

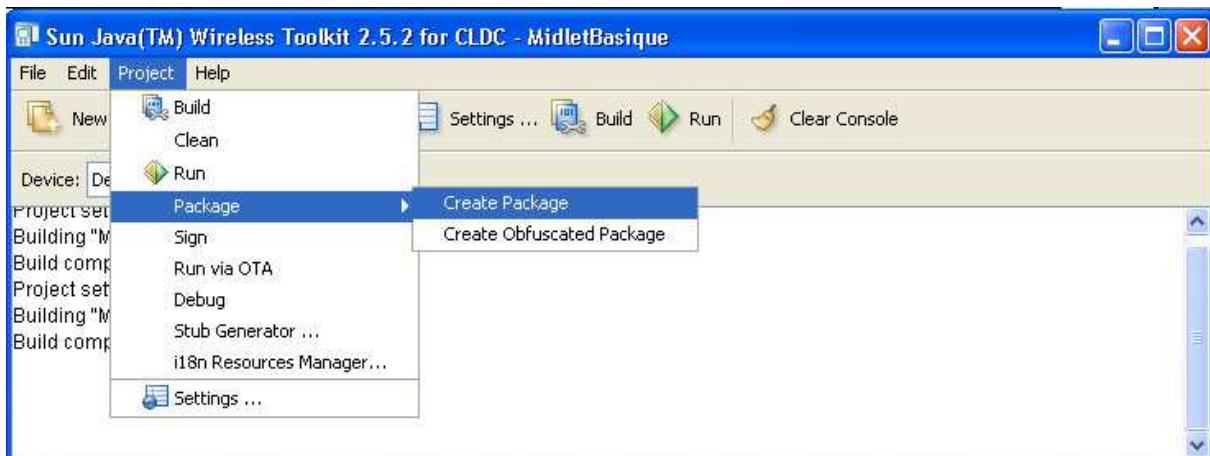
Building "MidletBasique"
Build complete

Un fichier MidletBasique.class a été créé dans le répertoire ...**classes** tandis qu'un fichier MidletBasique.jad est apparu en plus dans ...**bin**. Ce fichier jad est en fait un fichier de description de l'application (un "MIDlet descriptor") – on peut s'en convaincre en l'éditant :

MidletBasique.jad	
MIDlet-1:	MidletBasique, MidletBasique.png, HelloMobileWorld
MIDlet-Jar-Size:	1312
MIDlet-Jar-URL:	MidletBasique.jar
MIDlet-Name:	MidletBasique
MIDlet-Vendor:	Unknown
MIDlet-Version:	1.0
MicroEdition-Configuration:	CLDC-1.1
MicroEdition-Profile:	MIDP-2.1

L'idée est que l'invocation dans un browser de ce fichier descripteur provoquera le démarrage d'une MIDlet empaquetée dans un fichier MidletBasique.jar ... que nous n'avons pas encore créé. Agissons donc céans ...

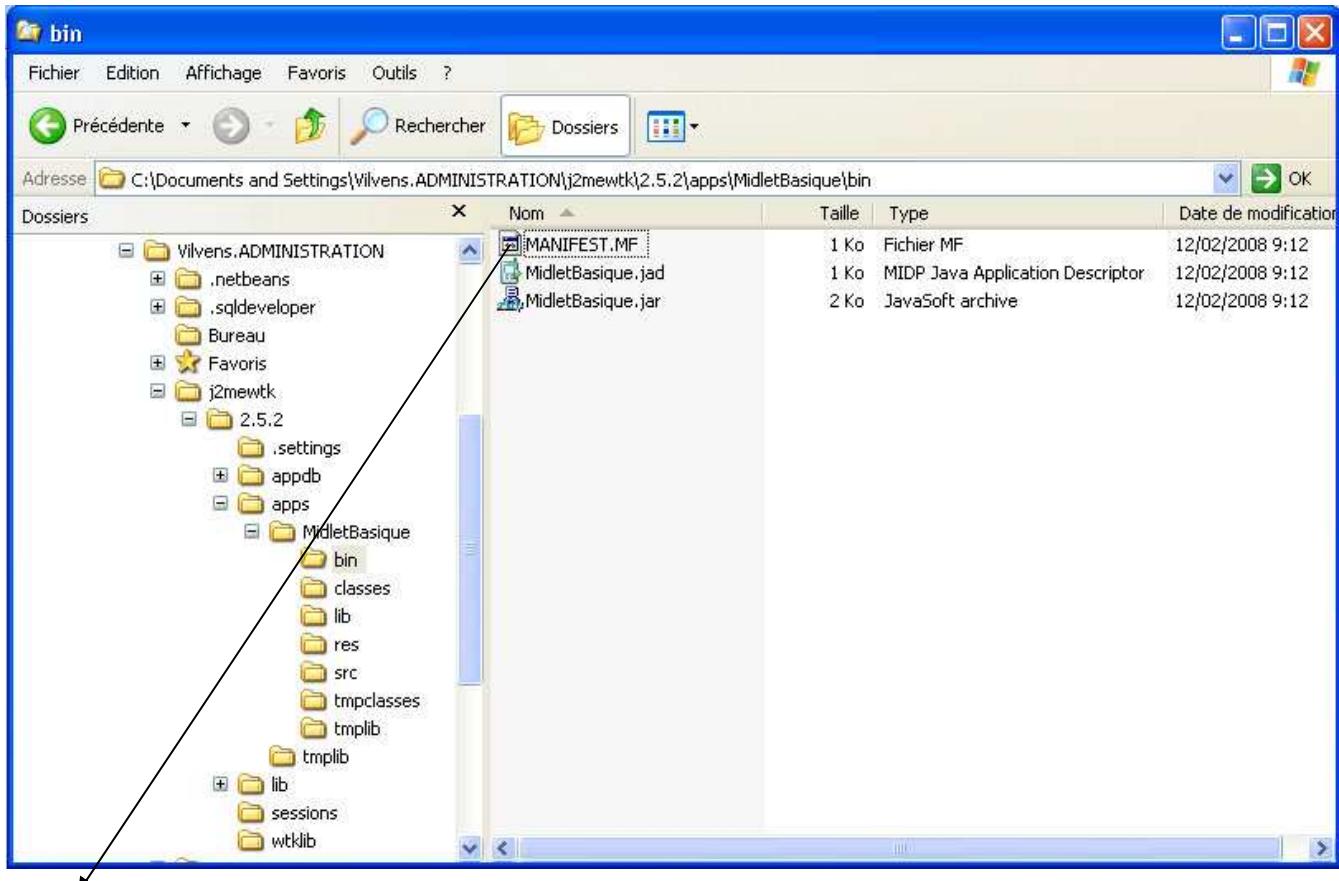
7.3 L'empaquetage du projet et son obfuscation



On peut remarquer la possibilité de créer un package "*obfuscated*" c'est-à-dire "obscurci" : l'idée est de rendre le code moins facile à décompiler. On y parvient en retirant des informations inutilisables de l'extérieur, comme par exemple en retirant le nom des

variables locales ou en renommant en interne avec des noms plus courts les classes, interfaces et méthodes, rendant la lecture éventuelle compliquée voire ambiguë. L'effet secondaire de cette pratique est intéressant : elle diminue la taille du code et conduit à la production de jars plus légers. Le travail d'obfuscation est réalisé par des classes Obfuscator.

Contentons-nous ici du package de base. On peut vérifier que le fichier a bien été créé dans ...\\bin : il contient le fichier class de notre MIDlet avec un manifeste inspiré du fichier jad :



MANIFEST.MF

Manifest-Version: 1.0

MicroEdition-Configuration: CLDC-1.1

MIDlet-Name: MidletBasique

Created-By: 1.5.0_06 (Sun Microsystems Inc.)

MIDlet-Vendor: Unknown

MIDlet-1: MidletBasique, MidletBasique.png, HelloMobileWorld

MIDlet-Version: 1.0

MicroEdition-Profile: MIDP-2.1

7.4 L'exécution de la MIDlet

Il suffit d'invoquer le bouton Run pour obtenir :



L'appui sur le bouton d'arrêt termine bien l'application avec comme message dans l'EDI :

```

Sun Java(TM) Wireless Toolkit 2.5.2 for CLDC - MidletBasique
File Edit Project Help
New Project ... Open Project ... Settings ... Build Run Clear Console
Device: DefaultColorPhone
Running with storage root C:\Documents and Settings\Wilvens\ADMINISTRATION\j2mewt\2.5.2\appdb\DefaultColorPhone
Running with locale: French_Belgium.1252
Running in the identified_third_party security domain
Execution completed.
3435542 bytecodes executed
238 thread switches
1668 classes in the system (including system classes)
18003 dynamic objects allocated (545032 bytes)
2 garbage collections (463820 bytes collected)

```

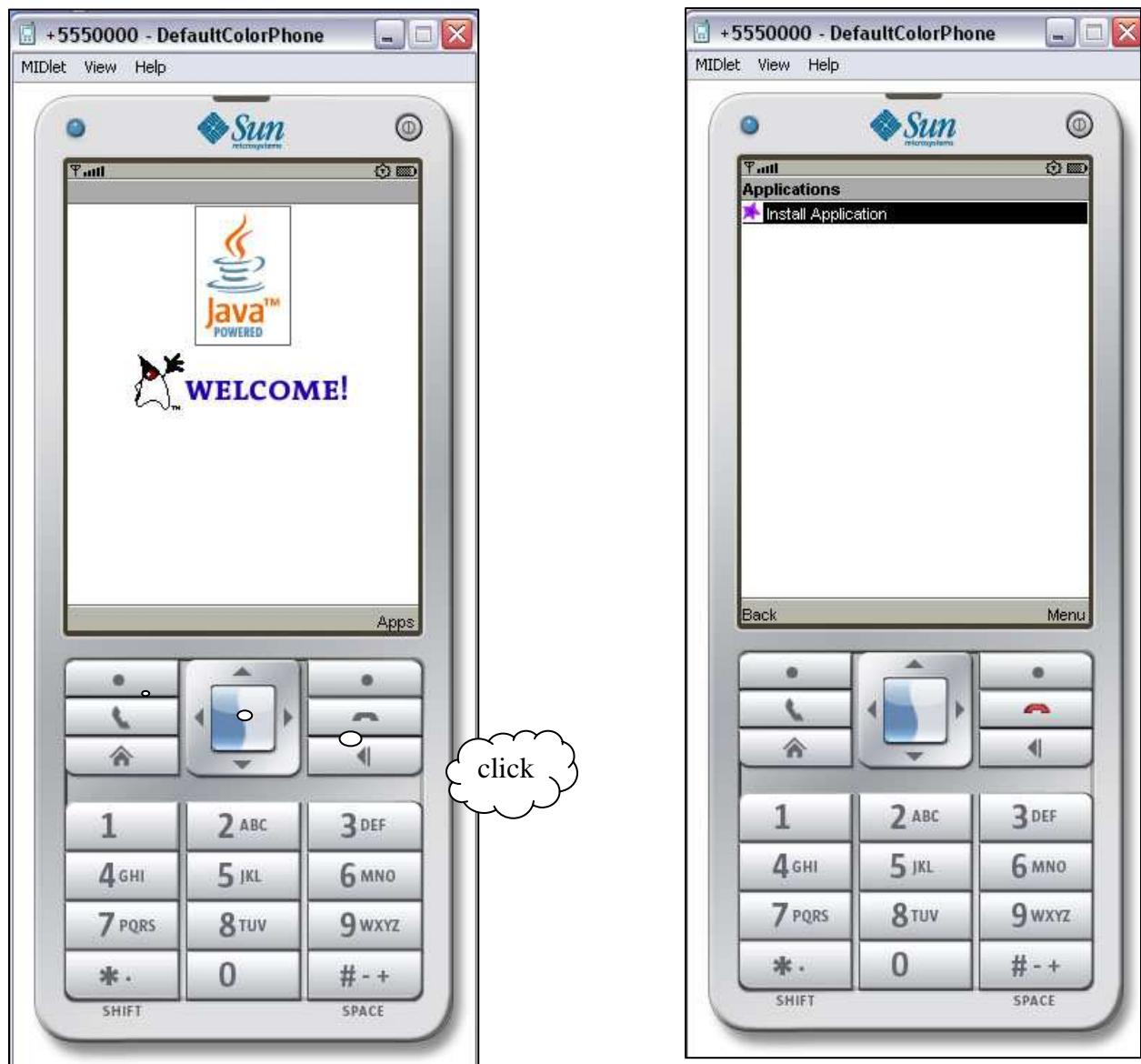
Evidemment, nous n'avons jamais que testé notre MIDlet "en vrai". Comment faire pour l'installer effectivement sur l'appareil qui doit lui servir de siège ?

7.5 Installation d'un projet MIDlet sur sa cible

Dans le test ci-dessus, deux opérations distinctes sont réalisées:

- le build : création du(des) fichier(s) class et jad de la MIDlet;
- le packaging : soit la création du fichier jar décrit ci-dessus.

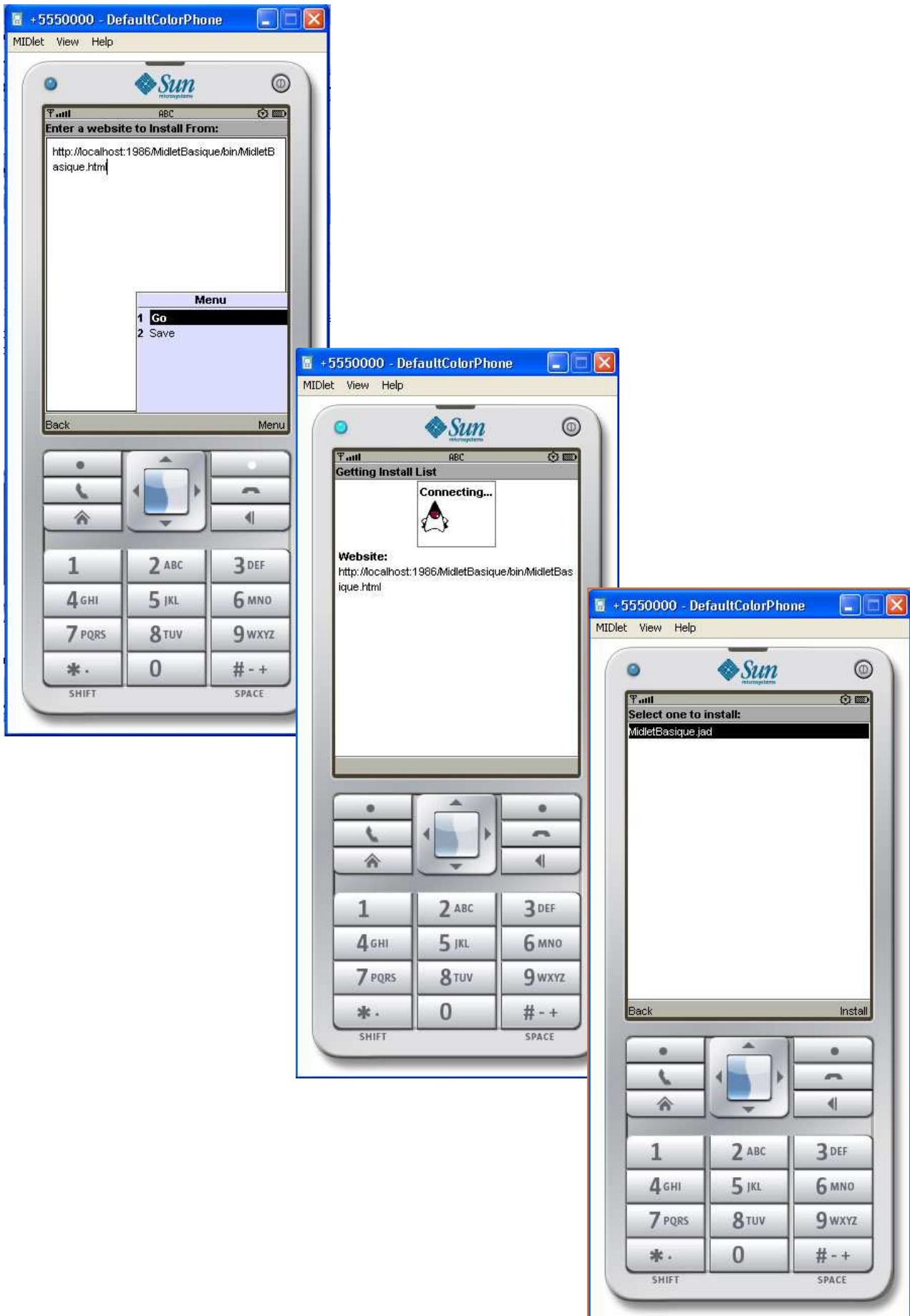
L'opération Run décrite ci-dessus ne fait que tester la MIDlet. Dans la perspective de la faire fonctionner sur un véritable mobile, il faut l'installer de manière à ce que celui-ci soit en état de la reconnaître pour la faire fonctionner (avec l'étape préalable de vérification légère). L'Application Management Software (AMS) réalise cette mise en place au niveau de l'émulateur. Dans le menu de l'EDI, le choix : Project → Run via OTA (OTA = Over The Air) fait apparaître l'"Emulator AMS Welcome Screen" :

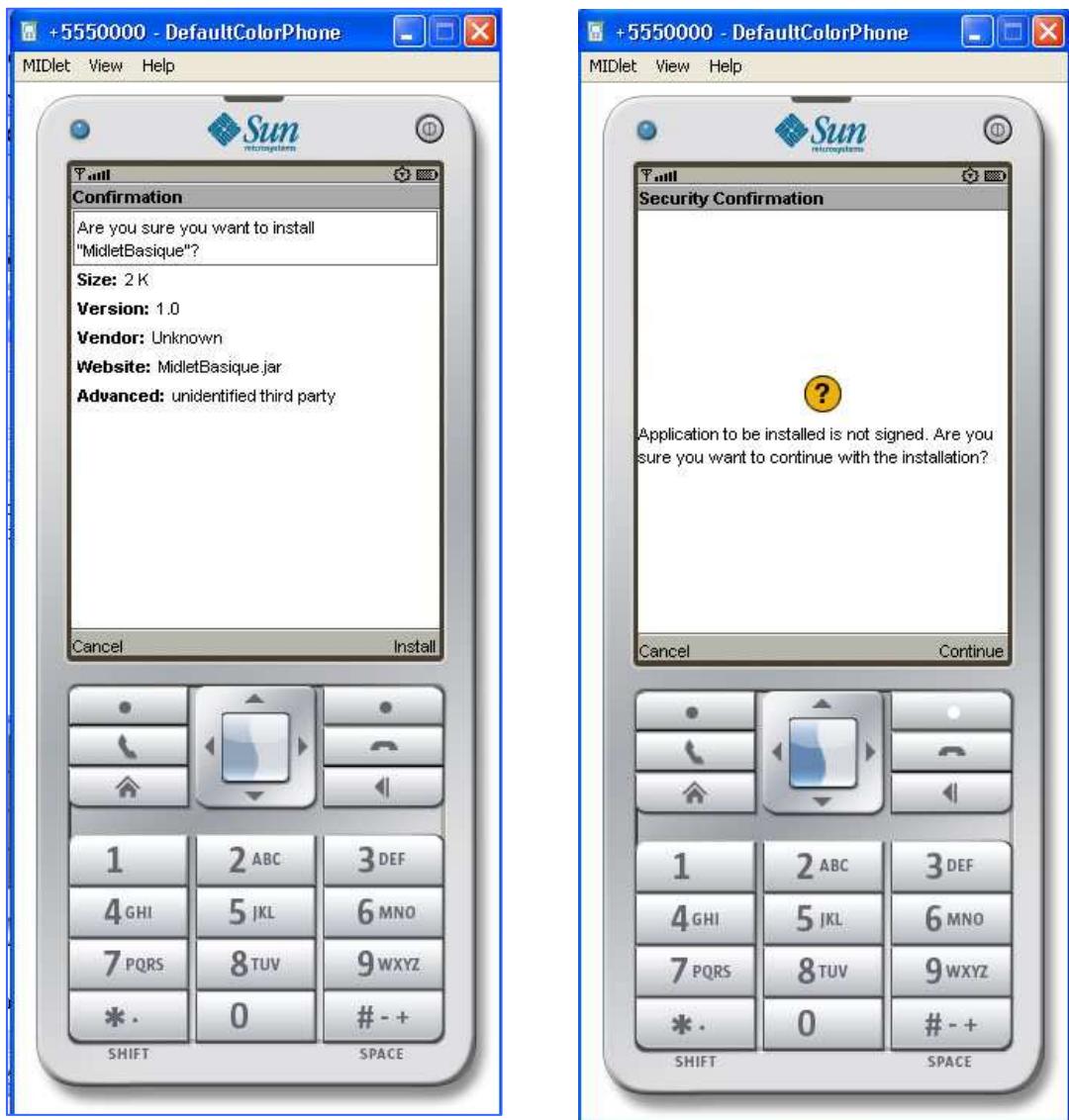


On sélectionne "Install application" puis "go" qui fournit l'URL de l'application à installer :

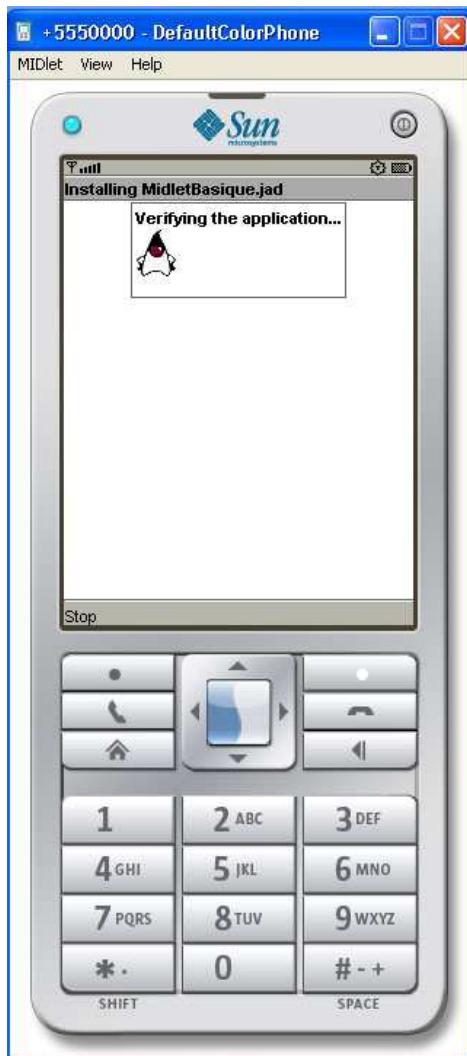


Le choix de "Go" permet de lancer l'installation : le fichier jad peut être sélectionné et il est demandé de confirmer l'installation





Une fois l'installation terminée, il est proposé de lancer l'application :



avec

```
File Edit Project Help
New Project ... Open Project ... Settings ... Build Run Clear Console
Device: DefaultColorPhone
Running with storage root C:\Documents and Settings\Wilvens.ADMINISTRATION\j2mewt\2.5.2\appdb\DefaultColorPhone
Running with locale: French_Belgium.1252
Device Bluetooth Address: 00003FC9317C
Execution completed.
4733882 bytecodes executed
23899 thread switches
1669 classes in the system (including system classes)
21129 dynamic objects allocated (745188 bytes)
2 garbage collections (451696 bytes collected)
Device Bluetooth Address: 00003FC9317C
Execution completed.
3521997 bytecodes executed
580 thread switches
1675 classes in the system (including system classes)
18569 dynamic objects allocated (568516 bytes)
2 garbage collections (476616 bytes collected)
```

On se retrouve enfin avec l'exécution de la MIDlet ... Mais il serait temps de passer à la réalité concrète ;-)

8. Un exemple de mobile

8.1 Présentation du mobile palmOne

Pour passer de l'émulateur à un mobile concret, nous avons utilisé tout d'abord l'ordinateur de poche palmOne Zire 72. Sa prise en charge par la machine de développement réclame l'instalaltion d'un logiciel de connexion dont l'installation ne présente aucune dificulté :



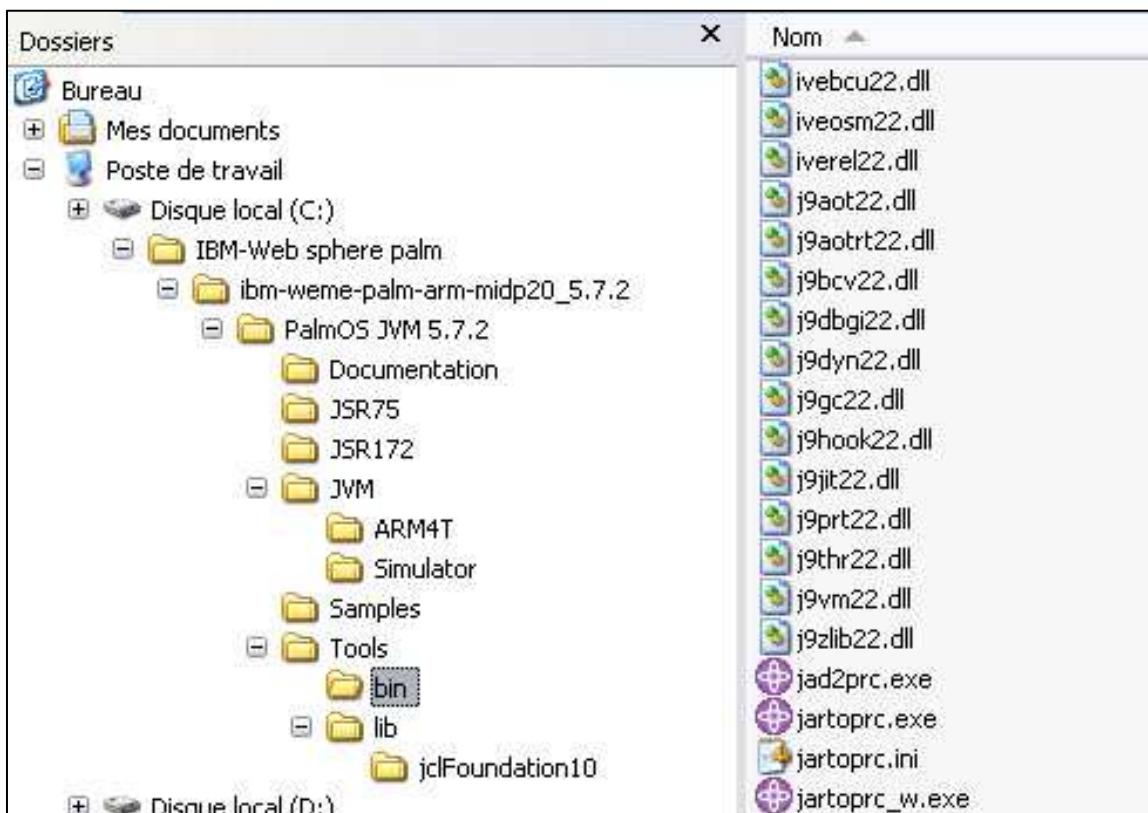
A remarquer cependant qu'il faudra introduire un nom d'utilisateur (ici, par exemple, James) pour identifier les opérations de synchronisation entre le Palm et le PC de développement. Dans ce qui a été installé, c'est l'item de menu "Installation rapide palmOne" qui nous sera le plus utile :



La "synchronisation" constitue simplement l'action de mettre à jour automatiquement des informations saisies sur un périphérique (ici, l'ordinateur de poche ou le PC de développement) sur un autre périphérique. Bien sûr, pour ce qui nous intéresse ici, ces "informations" seront en fait nos applications Java pour mobiles.

8.2 Une JVM pour mobile

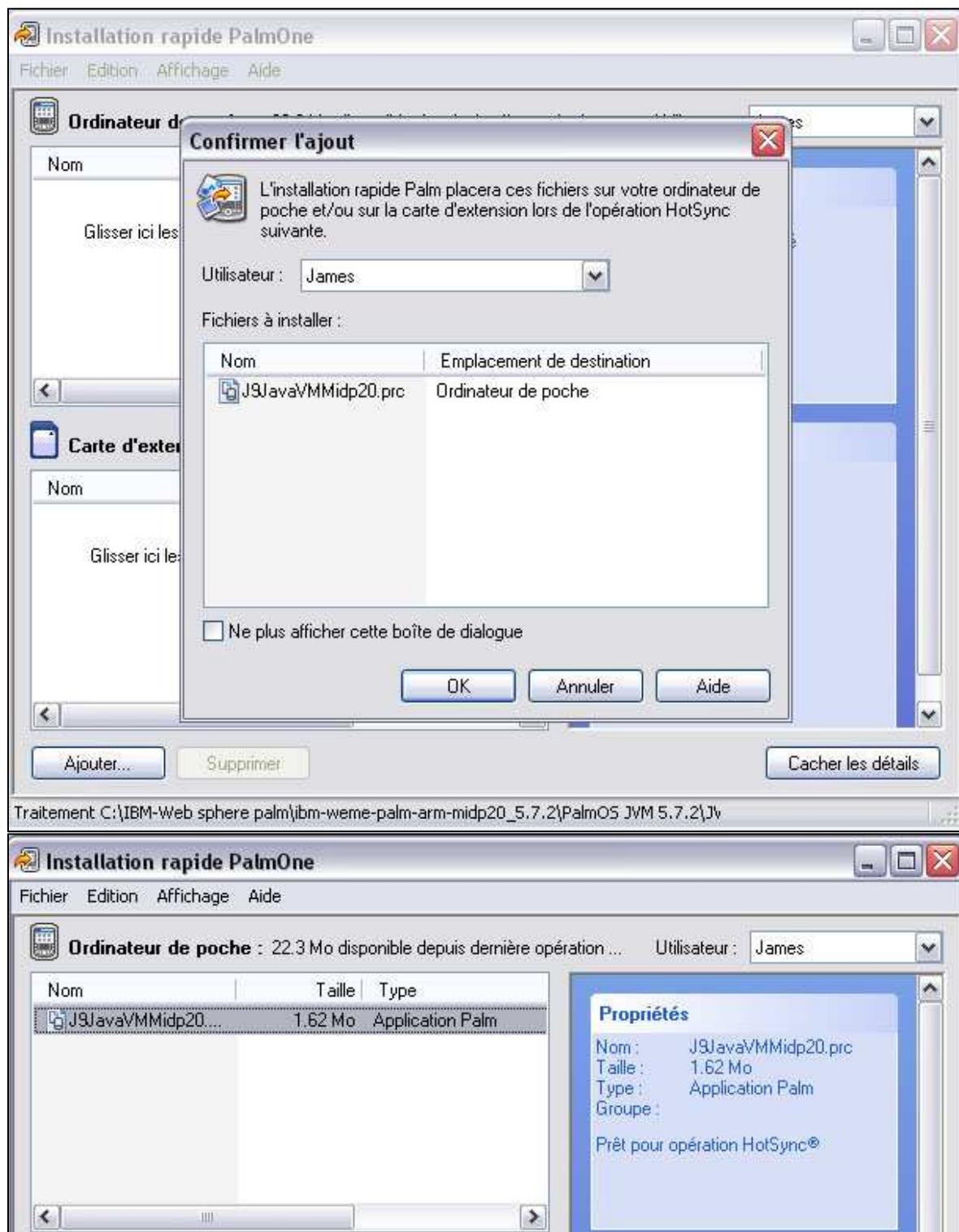
Reprendons notre application basique MidletBasique : nous voudrions la faire fonctionner sur l'ordinateur de poche. Il nous faut tout d'abord y installer une **machine virtuelle compatible** avec lui. Cette JVM est la J9VM de **WebSphere Everyplace Micro Environment (WEME)**, implémentation créée par IBM (en accord avec Sun) de la JVM ver 1.3 avec les librairies J2ME de type CLDC et MIDP2.0. On peut la trouver, par exemple, sur <http://www.mgmaps.com/weme/>, sous forme d'un fichier ibm-weme-palm-arm-midp20_5.7.2.zip qui, une fois décomprimé, présente l'aspect suivant :



Les mobile palmOne ne supportent l'installation que d'applications au format "**prc**" (**Palm Resource Code**) : or, précisément, la JVM d'IBM écrite pour Palm se présente sous la forme des fichiers :

J9JavaVMMidp20.prc
j9pref.prc

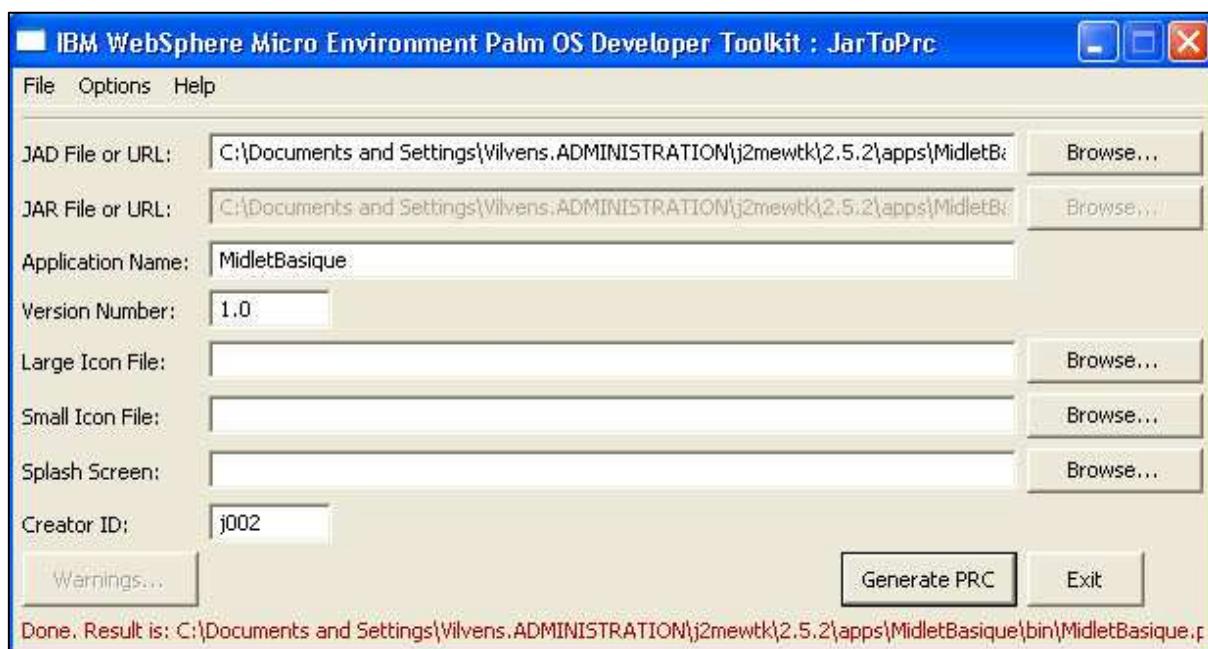
déclinés en diverses langues. Pour les télécharger, il suffit d'activer l'"Installation rapide palmOne" :



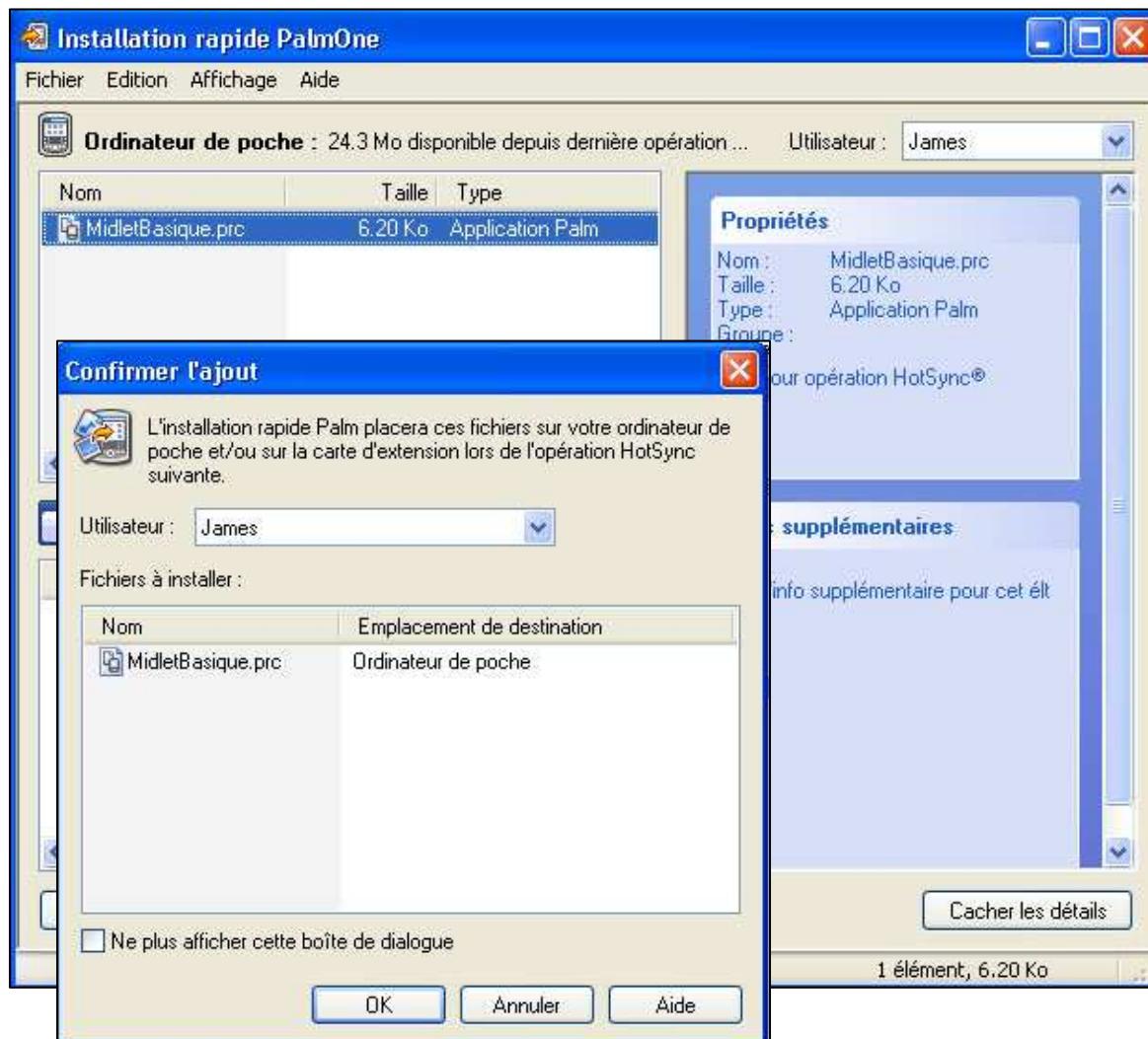
Une fois la liste complète, il suffit de réaliser l'opération de synchronisation (le "HotSync") pour que le transfert soit effectué : notre mobile comporte donc à présent une JVM !

8.3 Le test de l'application basique

Il nous reste à présent à placer notre application sur le palmOne. Un petit problème apparaît cependant immédiatement : le palmOne ne sait faire fonctionner que des fichiers au format "prc", pas des "jar" classique. Cependant, le package WEME comporte un outil permettant de réaliser cette conversion : **jartoprc_w.exe**. Celui-ci une fois lancé, présente l'aspect suivant :

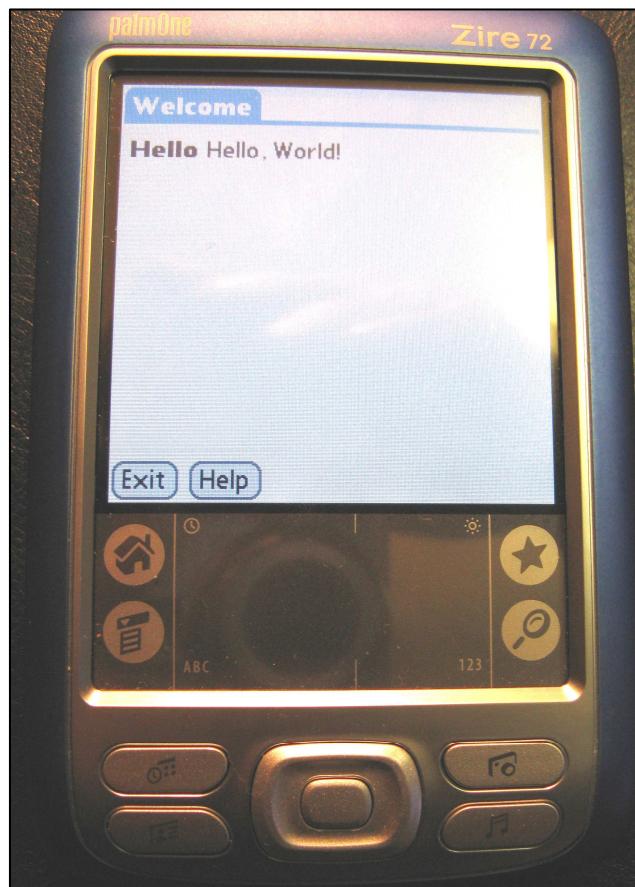


Il suffit donc de choisir le fichier jad associé à l'application : le jar sera retrouvé automatiquement et l'appui sur le bouton "Generate PRC" provoquera la création du fichier prc souhaité. Un HotSync permettra de transférer l'application sous son format prc :





Résultat :



9. Une MIDlet avec un menu-liste

9.1 Un système de menu classique

Evidemment, notre première MIDlet ne fait pas grand-chose. Nous allons donc envisager de créer une deuxième MIDlet qui présentera à son utilisateur un menu offrant différentes possibilités axées sur l'idée de mémoriser des contacts téléphoniques. En clair, on souhaite en finale obtenir le résultat suivant :



9.2 La mise en place d'une liste

A la place de la fenêtre limitée à une zone d'entrée, nous allons tout d'abord mettre en place une fenêtre qui présente une liste de choix possibles. La classe nécessaire est **List**, toujours du package javax.microedition.lcdui et qui implémente l'interface **Choice**. Son constructeur usuel est

```
public List (String title, listType, String[] stringElements, Image[] imageElements)
```

Le premier paramètre détermine évidemment le titre de la fenêtre tandis que la troisième désigne évidemment les chaînes de caractères constitutives de la liste. Le deuxième paramètre caractérise le type de liste désirée, en prenant comme valeur l'une des constantes définies dans l'interface Choice :

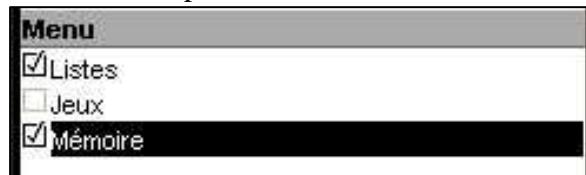
public static final int **IMPLICIT** : liste classique →



public static final int **EXCLUSIVE** : liste dont on ne peut sélectionner qu'un seul élément, comme l'indiquent les boutons radio →



public static final int **MULTIPLE** : liste dont on peut sélectionner plusieurs éléments, comme l'indiquent les cases à cocher →



A nouveau, nous allons associer une commande à cette liste et déclarer que notre MIDlet est listener de cette commande – ceci est analogue à ce que nous avons programmé pour la zone de texte :

MenuMIDlet.java

```
/*
 * MenuMIDlet.java
 * Created on 6 octobre 2005, 10:33
 * Last modified on 27 février 2008, 07:35
 */

package MIDletGUIs;

import javax.microedition.MIDlet.*;
import javax.microedition.lcdui.*;

/**
 * @author Vilvens
 * @version
 */

public class MenuMIDlet extends MIDlet implements CommandListener
{
    private Display display;
    private List Menu;
    private Command exitListes;

    public MenuMIDlet()
    {
        display = Display.getDisplay(this);
        String[] elements = {"Listes", "Jeux", "Mémoire"};
    }
}
```

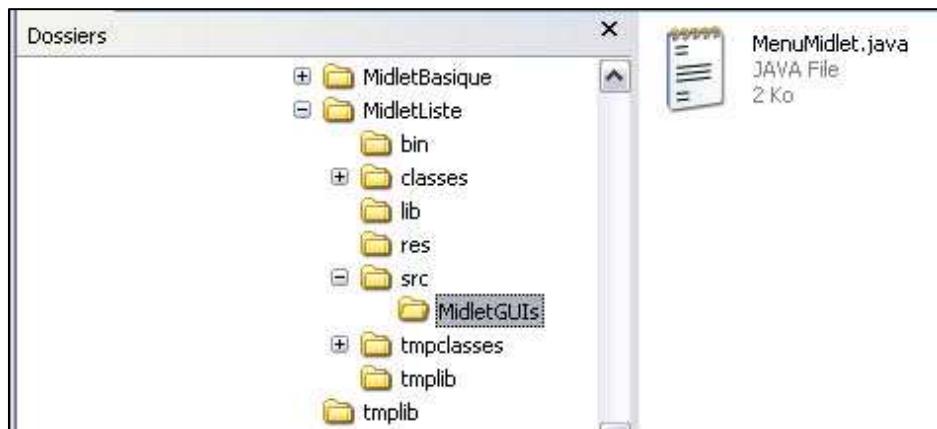
```
exitListes = new Command("Listes", Command.EXIT, 1);
Menu = new List ("Menu", List.IMPLICIT, elements, null);
Menu.addCommand(exitListes);
Menu.setCommandListener(this);
}

public void startApp()
{
    display.setCurrent(Menu);
}

public void pauseApp() { }
public void destroyApp(boolean unconditional) { }

public void commandAction(Command cmd, Displayable s)
{
    if(cmd == exitListes) s.setTitle(Menu.getString(Menu.getSelectedIndex()));
}
}
```

Nous sauvons notre MIDlet dans un répertoire portant le même nom que le package :



Nous pouvons ensuite compiler, empaqueter et installer comme pour la première MIDlet.
Un appui sur la touche "exit" change effectivement le titre de la fenêtre en l'item sélectionné dans la liste :



En fait, un objet List possède une variable membre Command dont la valeur par défaut est :

`public static final Command SELECT_COMMAND`

- celle-ci correspond évidemment à la sélection d'un item dans une liste de type IMPLICIT. Nous pouvons donc compléter la méthode commandAction() par :

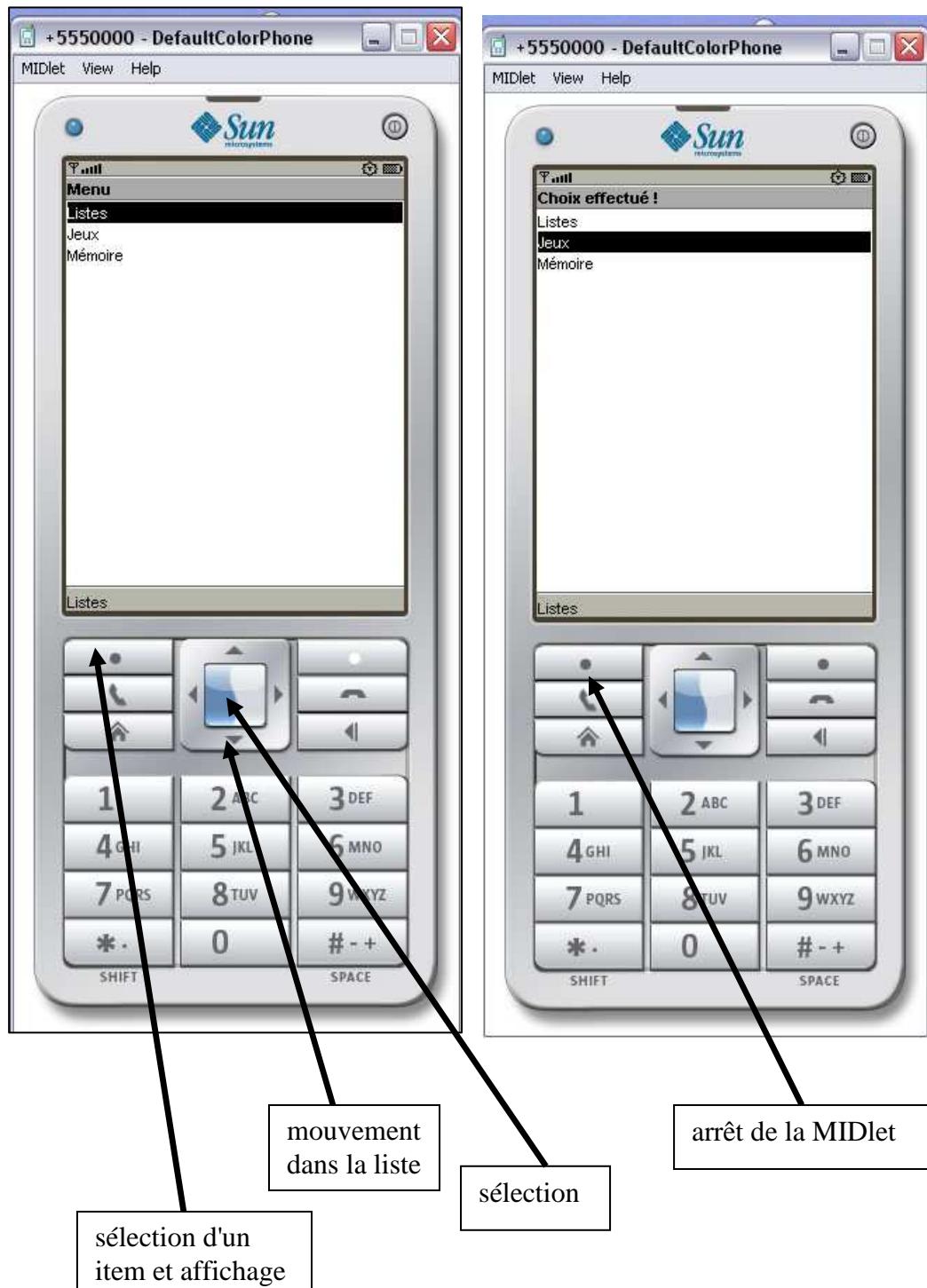
```
public void commandAction(Command cmd, Displayable s)
{
    try
    {
        if(cmd == exitListes) s.setTitle(Menu.getString(Menu.getSelectedIndex()));
    }
}
```

```

if (cmd == List.SELECT_COMMAND) s.setTitle("Choix effectué !");
}
catch ( Exception e) { System.out.println(e); }
}

```

ce qui donne bien :



Evidemment, le résultat n'est pas extraordinaire – allons plus loin. Mais d'abord, passons à une EDI de luxe ;-)

10. Netbeans et son Mobility Pack

10.1 Génération d'une MIDlet basique

A l'heure actuelle, NetBeans 6.* est devenu un énorme environnement de développement qui comporte de multiples modules, dont le "Mobility Pack". NetBeans 5.* peut d'ailleurs être complété de ce même module.



Le pack complémentaire en question comporte les APIs nécessaires au développement d'une application J2ME, soit CLDC 1.1, MIDP 2.0 et portable ToolKit 2.2.

Concrètement, créer une application pour mobile démarre en créant un projet dédié aux mobiles :

New Project

Steps

1. Choose Project
2. ...

Choose Project

Categories:

- Java
- Java Web
- Java EE
- Java ME**
- PHP
- Ruby
- Groovy
- C/C++
- SOA
- NetBeans Modules
- Examples

Projects:

- Mobile Application
- Mobile Class Library
- Mobile Project with Existing MIDP Sources
- Import Wireless Toolkit Project
- CDC Application
- CDC Class Library
- Import CDC Pack 5.5 Project
- Import CDC Toolkit Project
- Mobile Designer Components

New Mobile Application

Steps

1. Choose Project
2. **Name and Location**
3. Default Platform Selection
4. More Configurations Selection

Name and Location

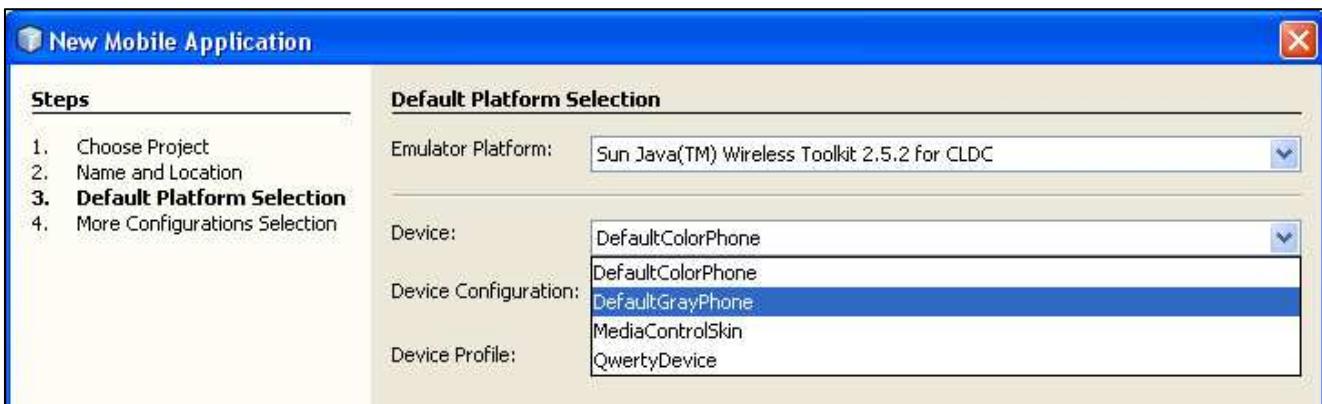
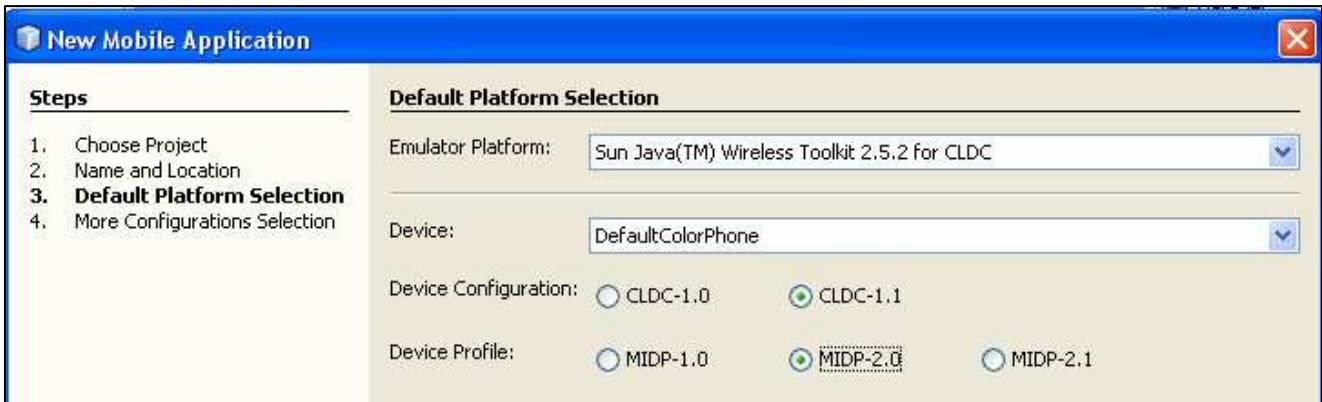
Project Name:

Project Location:

Project Folder:

Set as Main Project

Create Hello MIDlet



Le résultat généré par l'environnement de développement est une MIDlet qui dit bonjour (avec Netbeans 5.5.* , après cependant expurgé le code de quelques accolades de trop ;-) , comme on peut le vérifier en l'exécutant au moyen de l'émulateur sélectionné :



Voyons le code généré :

HelloMIDlet.java (code généré)

```
package hello;

import javax.microedition.MIDlet.*;
import javax.microedition.lcdui.*;

/**
 * @author Vilvens
 */
public class HelloMIDlet extends MIDlet implements CommandListener
{
    private boolean MIDletPaused = false;

    private Command exitCommand;
    private Form form;
    private StringItem stringItem;

    public HelloMIDlet() {}

    private void initialize()
    {
        // write pre-initialize user code here

        // write post-initialize user code here
    }

    public void startMIDlet()
    {
        // write pre-action user code here
        switchDisplayable(null, getForm());
        // write post-action user code here
    }

    public void resumeMIDlet()
    {
        // write pre-action user code here

        // write post-action user code here
    }

    public void switchDisplayable(Alert alert, Displayable nextDisplayable)
    {
        Display display = getDisplay();
        if (alert == null) { display.setCurrent(nextDisplayable); }
        else { display.setCurrent(alert, nextDisplayable); }
    }
}
```

```
public void commandAction(Command command, Displayable displayable)
{
    if (displayable == form) {
        if (command == exitCommand) { exitMIDlet(); }
    }
}

public Command getExitCommand()
{
    if (exitCommand == null)
    { exitCommand = new Command("Exit", Command.EXIT, 0); }
    return exitCommand;
}

public Form getForm()
{
    if (form == null)
    {
        form = new Form("Welcome", new Item[] { getStringItem() });
        form.addCommand(getExitCommand());
        form.setCommandListener(this);
    }
    return form;
}

public StringItem getStringItem()
{
    if (stringItem == null) { stringItem = new StringItem("Hello", "Hello, World!"); }
    return stringItem;
}

public Display getDisplay () { return Display.getDisplay(this); }

public void exitMIDlet()
{
    switchDisplayable (null, null);
    destroyApp(true);
    notifyDestroyed();
}

public void startApp()
{
    if (MIDletPaused) { resumeMIDlet (); }
    else
    {
        initialize ();
        startMIDlet ();
    }
    MIDletPaused = false;
}
```

```

public void pauseApp()
{
    MIDletPaused = true;
}

public void destroyApp(boolean unconditional) { }

}

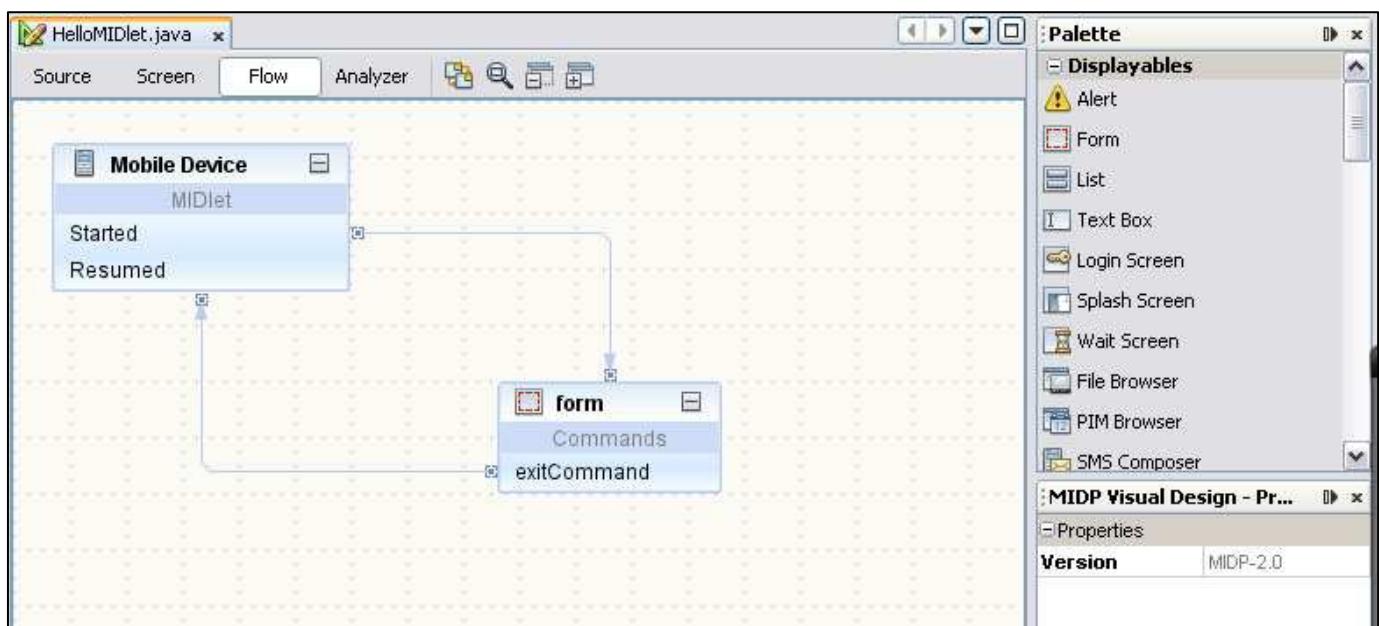
```

On en peut pas dire que ce soit un code difficile – il est seulement compliqué ;-) ... mais en fait parfaitement compréhensible avec nos connaissances actuelles. On remarquera les nombreuses méthodes propres à l'EDI, comme startMIDlet(), resumeMIDlet() et exitMIDlet() ainsi que la vision très "Java Bean" avec les méthodes getXXX() utilisées à outrance ;-)

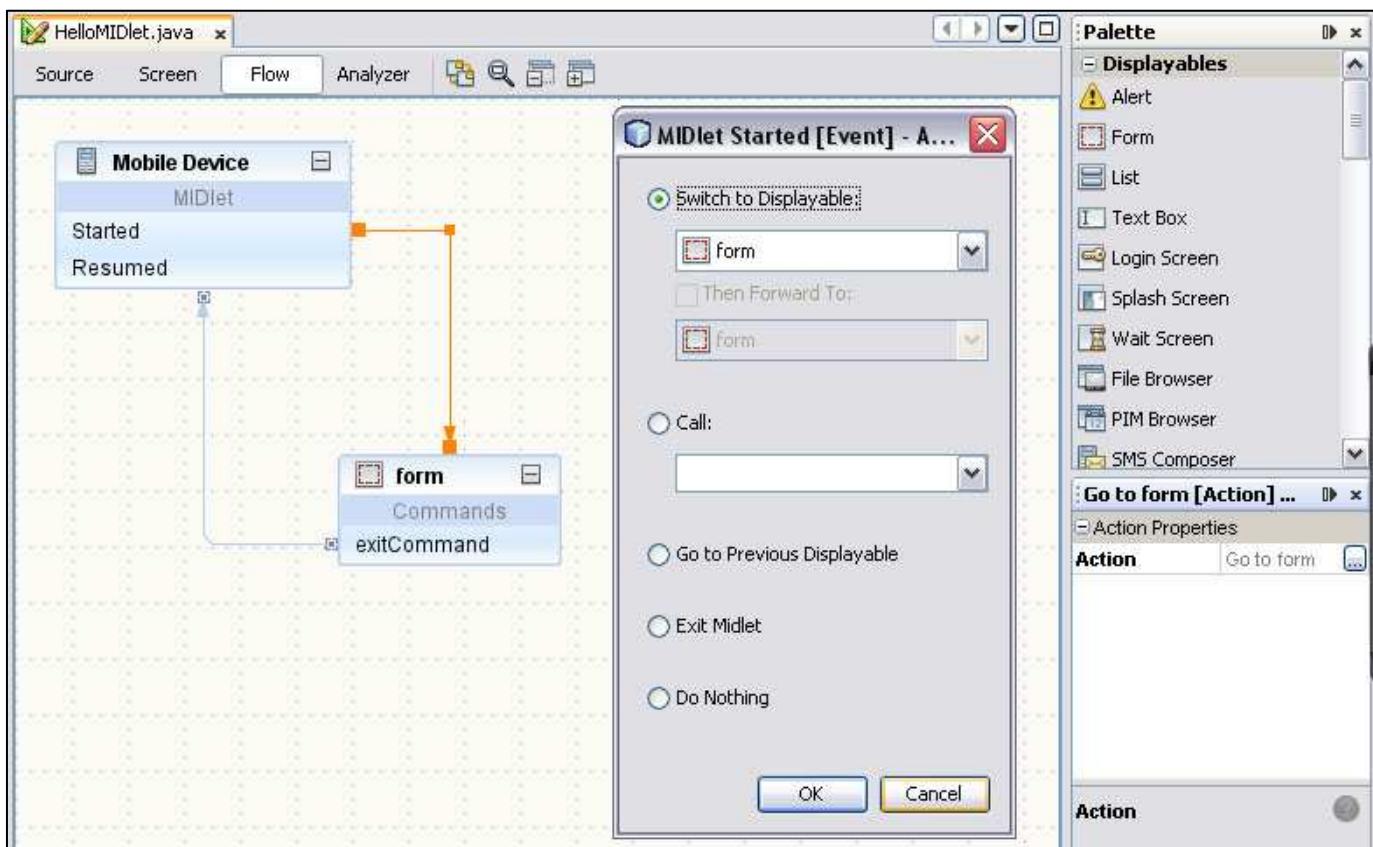
10.2 Utilisation des outils de développement

Mais ce qui frappe avant tout, c'est le décor de l'environnement de développement J2ME proposé. En effet, outre l'onglet Source, on dispose aussi d'autres panels.

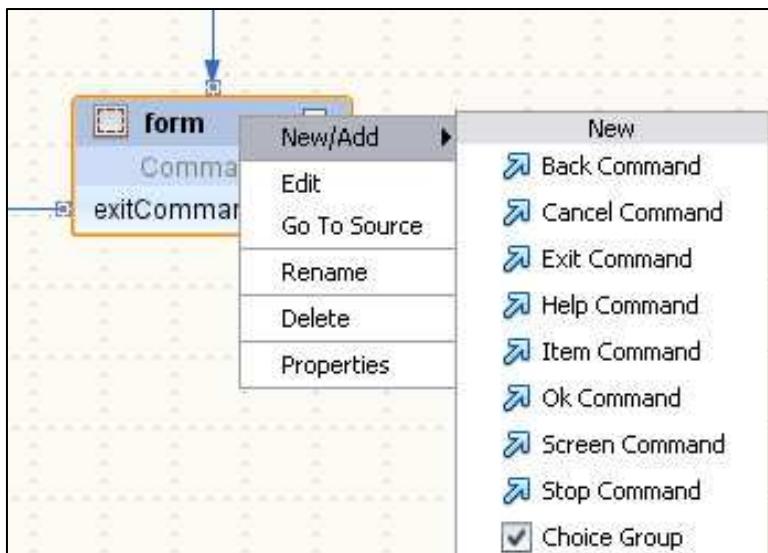
A. L'onglet Flow :



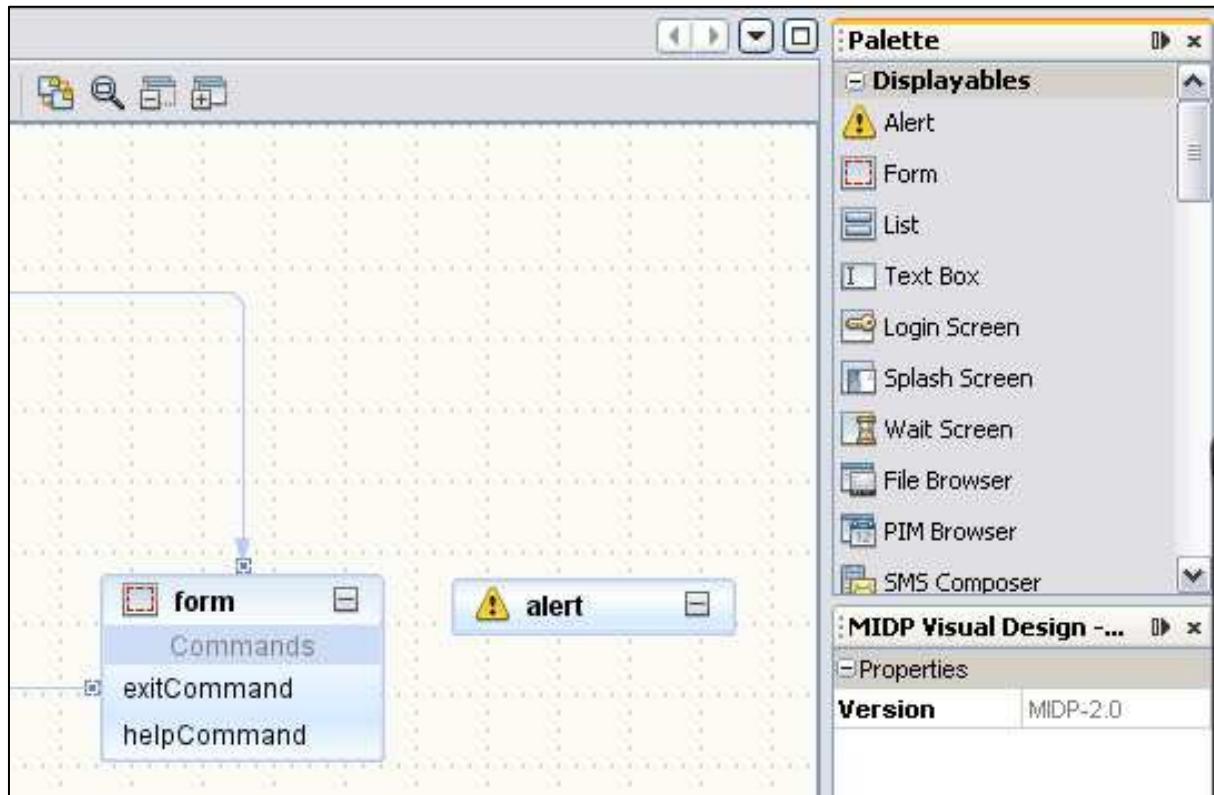
Cet onglet sert à définir les réactions de la MIDlet (représentée par le "Mobile Device" qui la fait fonctionner) aux commandes reçues. Ainsi, la sélection de la flèche associée à "Started" suivie d'un clic sur l'ellipse ("...") accolée à l'action donne :



On constate que l'on pourrait donc, par exemple, forcer l'appel à une méthode ("Call"). On peut imaginer aussi d'ajouter le traitement d'une autre commande, par exemple Help – un simple clic droit:



Pour préparer la réponse à cette commande, plaçons un composant de plus extrait de la palette, soit un objet Alert :



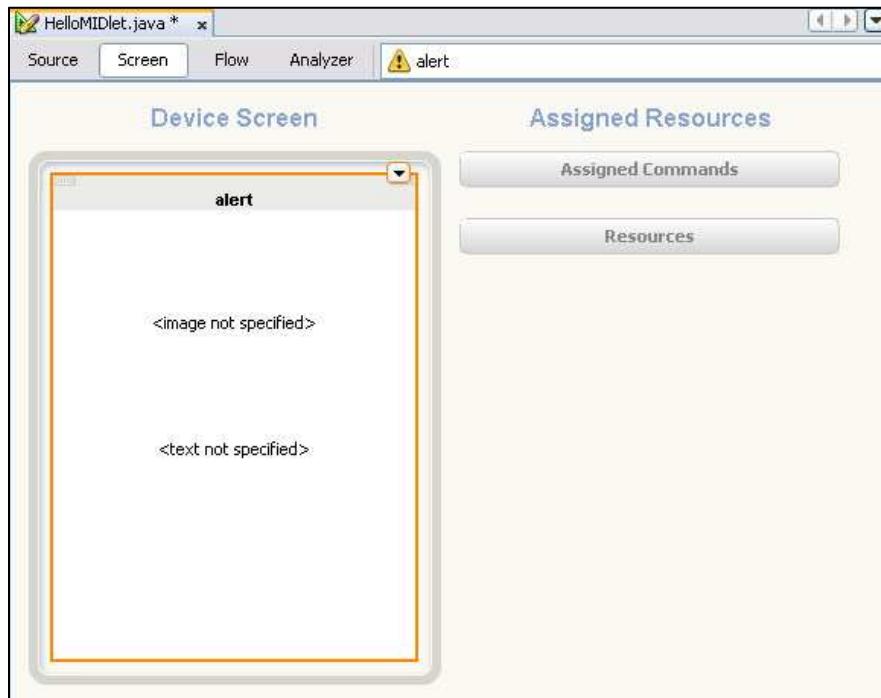
Nous l'associons très intuitivement à notre commande Help :



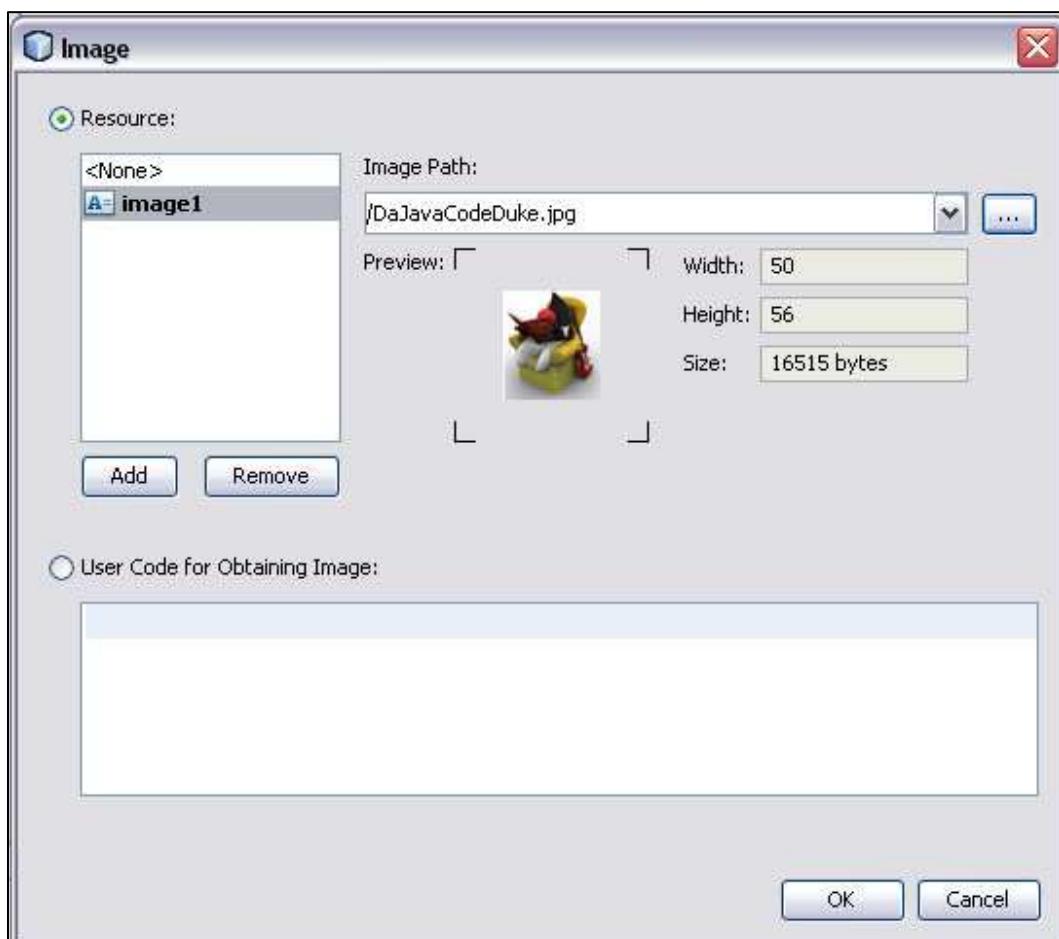
Pour configurer notre Alert, un double clic dessus nous mène à un autre onglet :

B. L'onglet Screen :

Nous nous retrouvons ainsi devant :



Bien sûr, il s'agit ici avant tout de fournir le texte affiché par la boîte de dialogue d'alerte. On peut aussi ajouter une image :



Résultat :



et une simple exécution donne :

Au niveau du code généré, on peut remarquer

- ♦ l'apparition de nouvelles variables membres :

```
private Command helpCommand;  
private Image image1;  
private Alert alert;
```

- la classe Image possède une factory

```
public static Image createImage (String name) throws IOException
```

permettant de créer un objet image à partir d'une ressource nommée, donc ici un fichier.;

- ♦ la modification de la méthode de listener de commande :

```
public void commandAction(Command command, Displayable displayable)  
{  
    if (displayable == form)  
    {  
        if (command == exitCommand)  
        {  
            exitMIDlet();  
        }  
        else if (command == helpCommand)  
        {  
            switchDisplayable(null, getAlert());  
        }  
    }  
}
```

avec

```
public Alert getAlert()  
{  
    if (alert == null)  
    {  
        alert = new Alert("alert", "Aide toi, le ciel t'aidera ...", getImage1(), null);  
        alert.setTimeout(Alert.FOREVER);  
    }  
    return alert;  
}
```

et

```
public Image getImage1()  
{  
    if (image1 == null)  
    {
```

```
try
{
    image1 = Image.createImage("/DaJavaCodeDuke.jpg");
}
catch (java.io.IOException e)
{
    e.printStackTrace();
}
return image1;
}
```

On remarquera la méthode de la classe Alert

```
public void setTimeout(int time)
```

qui permet de limiter l'apparition de la boîte à un temps exprimé en millisecondes, à moins que l'on utilise la constante de classe

```
public static final int FOREVER
```

qui aura comme effet de maintenir l'affichage jusqu'à l'appui sur un bouton.

Du point de vue déploiement, on peut trouver comme d'habitude dans le répertoire dist un jar dont le manifeste contient sans surprise :

MANIFEST.MF

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.0
Created-By: 1.5.0_06-b05 (Sun Microsystems Inc.)
MIDlet-1: HelloMIDlet, , hello.HelloMIDlet
MIDlet-Vendor: Vendor
MIDlet-Version: 1.0
MIDlet-Name: MideltBigMenu
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.1
```

10.3 Utilisation de la MIDlet générée avec le Sun Java Wireless Toolkit

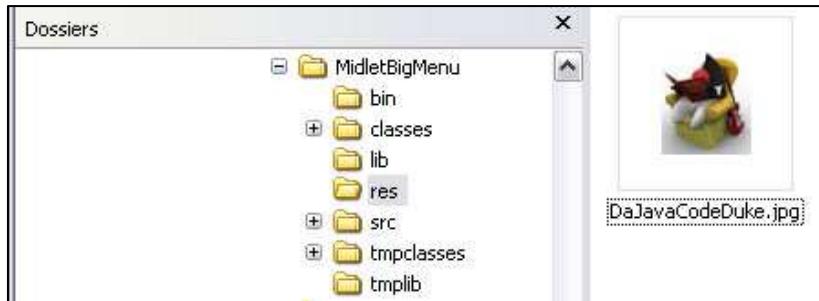
Nous allons à présent récupérer notre MIDlet pour la faire fonctionner dans l'outil Wireless Toolkit puis sur notre palmOne.

a) Après avoir créé un projet dans SJWT, en donnant comme nom de MIDlet son nom complet (hello.HelloMIDlet), nous pouvons tenter d'utiliser le jar produit par Netbeans. Cependant, si l'application veut bien démarrer, l'image quant à elle ne s'affiche pas :

```
Running in the identified _third_party security domain
java.io.IOException
    at javax.microedition.lcdui.ImmutableImage.getImageFromStream(+15)
    at javax.microedition.lcdui.ImmutableImage.<init>(+20)
```

```
at javax.microedition.lcdui.Image.createImage(+8)
at hello.HelloMIDlet.getImage1(+13)
at hello.HelloMIDlet.getAlert(+20)
at hello.HelloMIDlet.commandAction(+37)
at javax.microedition.lcdui.Display$DisplayAccessor.commandAction(+282)
at javax.microedition.lcdui.Display$DisplayManagerImpl.commandAction(+10)
at com.sun.midp.lcdui.DefaultEventHandler.commandEvent(+68)
at com.sun.midp.lcdui.AutomatedEventHandler.commandEvent(+47)
at com.sun.midp.lcdui.DefaultEventHandler$QueuedEventHandler.handleVmEvent(+186)
at com.sun.midp.lcdui.DefaultEventHandler$QueuedEventHandler.run(+57)
```

En fait, l'image doit se trouver pour SJWT dans le répertoire res du projet :



Créons le package correspondant – il a pour manifeste :

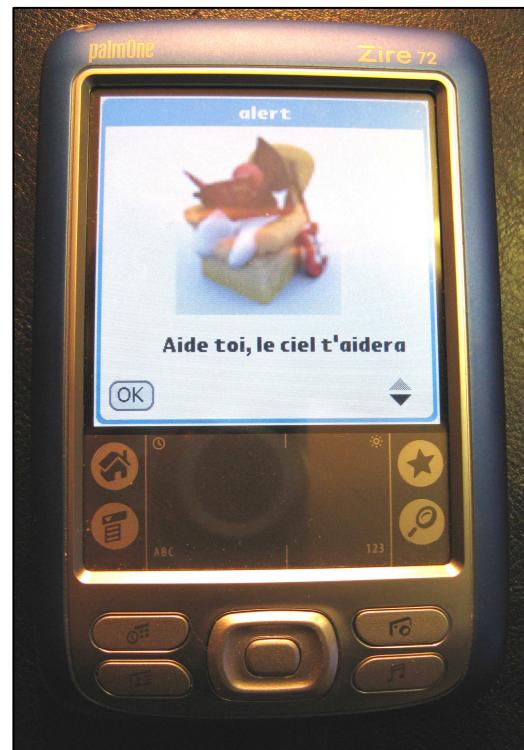
MANIFEST.MF	
Manifest-Version:	1.0
MicroEdition-Configuration:	CLDC-1.1
MIDlet-Name:	MIDletBigMenu
Created-By:	1.5.0_06 (Sun Microsystems Inc.)
MIDlet-Vendor:	Unknown
MIDlet-1:	MIDletBigMenu, MIDletBigMenu.png, hello.HelloMIDlet
MIDlet-Version:	1.0
MicroEdition-Profile:	MIDP-2.1

tandis le jad s'écrit :

MIDletBigMenu.jad

MIDlet-1: MIDletBigMenu, MIDletBigMenu.png, hello.HelloMIDlet
MIDlet-Jar-Size: 9980
MIDlet-Jar-URL: MIDletBigMenu.jar
MIDlet-Name: MIDletBigMenu
MIDlet-Vendor: Unknown
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.1

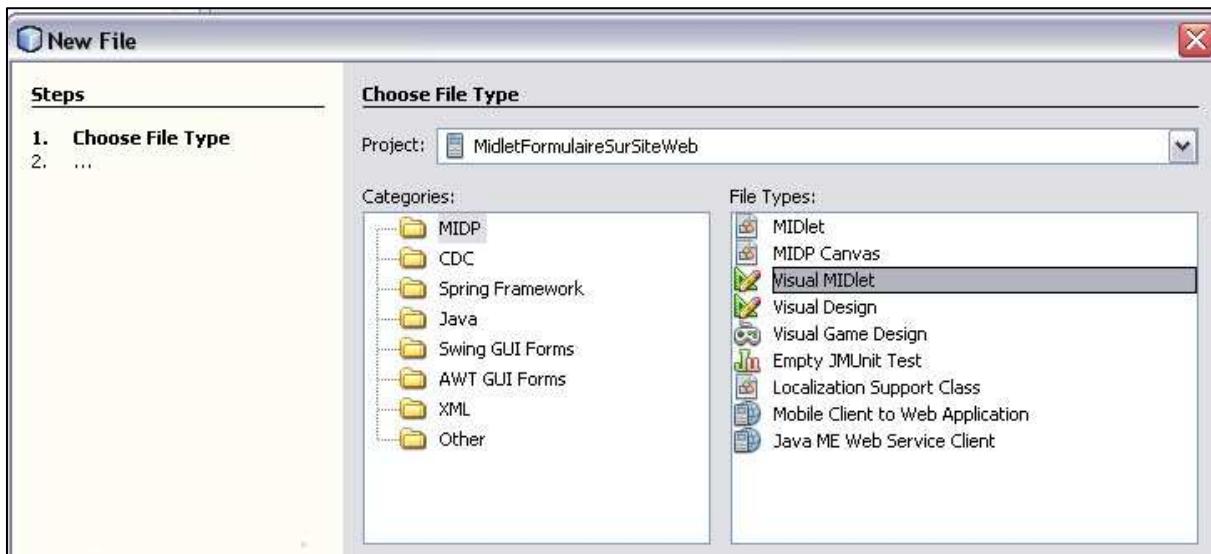
Après génération du fichier prc, nous pouvons effectuer un HotSync. La MIDlet est bien téléchargée sur le palmOne mais, à l'exécution, l'image n'apparaît pas. Quel est le problème ? En fait, nous utilisons une image de format **jpg** : or, notre palmOne ne supporte que les images au format **png** (Portable Network Graphics, un format d'image destiné à remplacer le format gif qui était à l'époque propriétaire). Si donc nous remplaçons notre image jpg par son équivalent en png, tout va fonctionner :



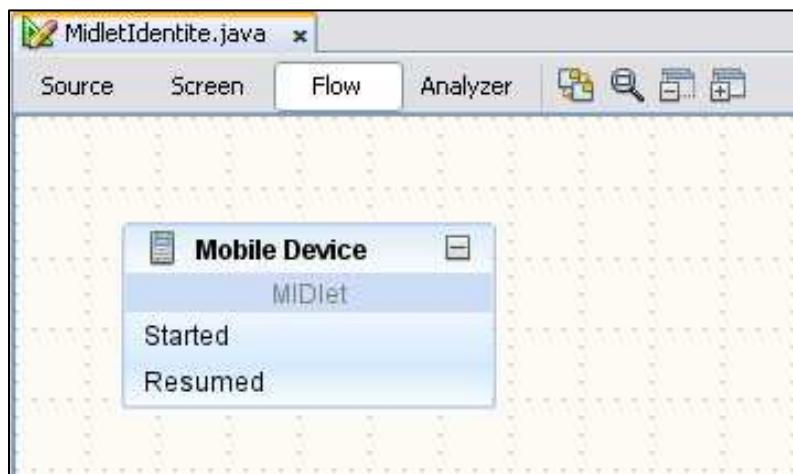
11. Une MIDlet à formulaire élémentaire placée sur un serveur Web

11.1 Une MIDlet à formulaire avec Netbeans

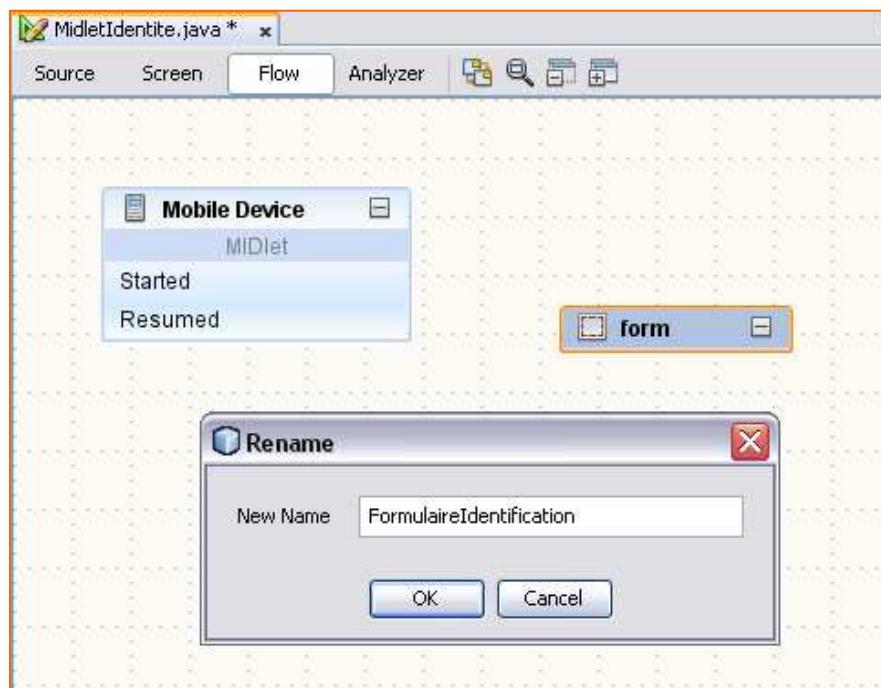
Nous allons donc créer un projet de type Mobility ayant pour nom MIDletFormulaireSurSiteWeb. Nous allons créer notre MIDlet à partir de rien. Donc, nous éliminons le package hello puis nous créons le package MIDletsFormulaires et la MIDlet MIDletIdentite (attention à bien choisir une MIDlet "visuelle") :



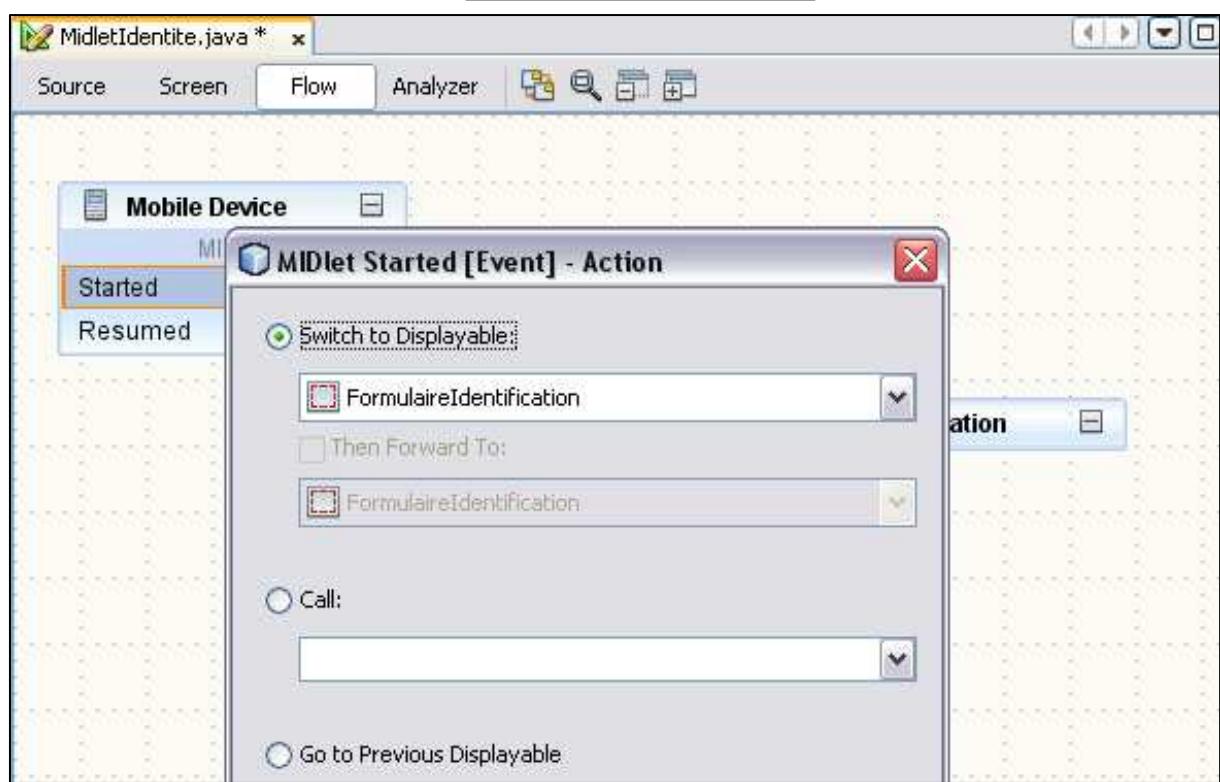
Nous nous retrouvons donc avec une MIDlet qui ne fait encore rien, comme le montre l'onglet Flow :



Ensuite, dans l'onglet Flow, nous plaçons un formulaire que nous renommons en "FormulaireIdentification" : un simple clic droit et le choix Rename permet de réaliser l'opération.



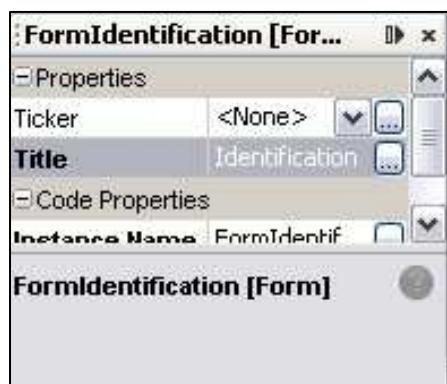
Nous précisons ensuite que, lorsque la MIDlet démarre, c'est ce formulaire qui doit apparaître. Pour ce faire, on sélectionne "Started" et on clique sur l'ellipse ("...") accolée à l'action



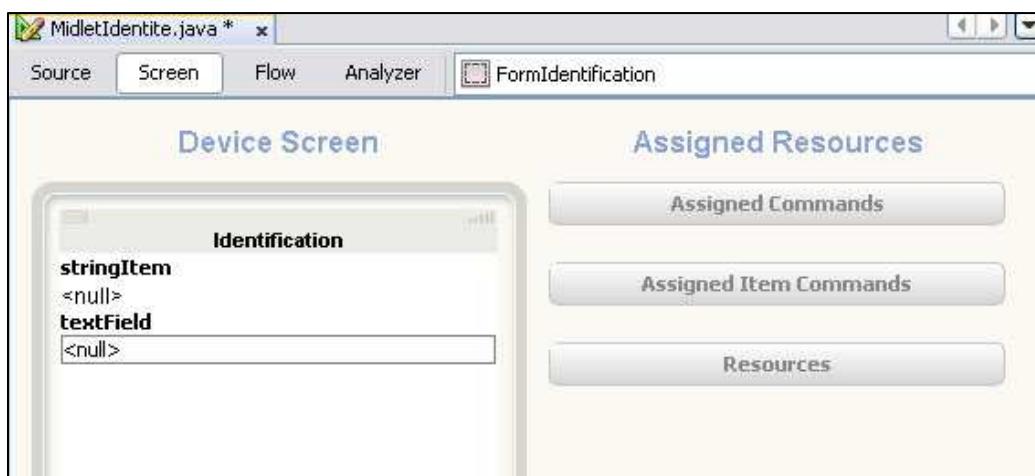
Résultat :



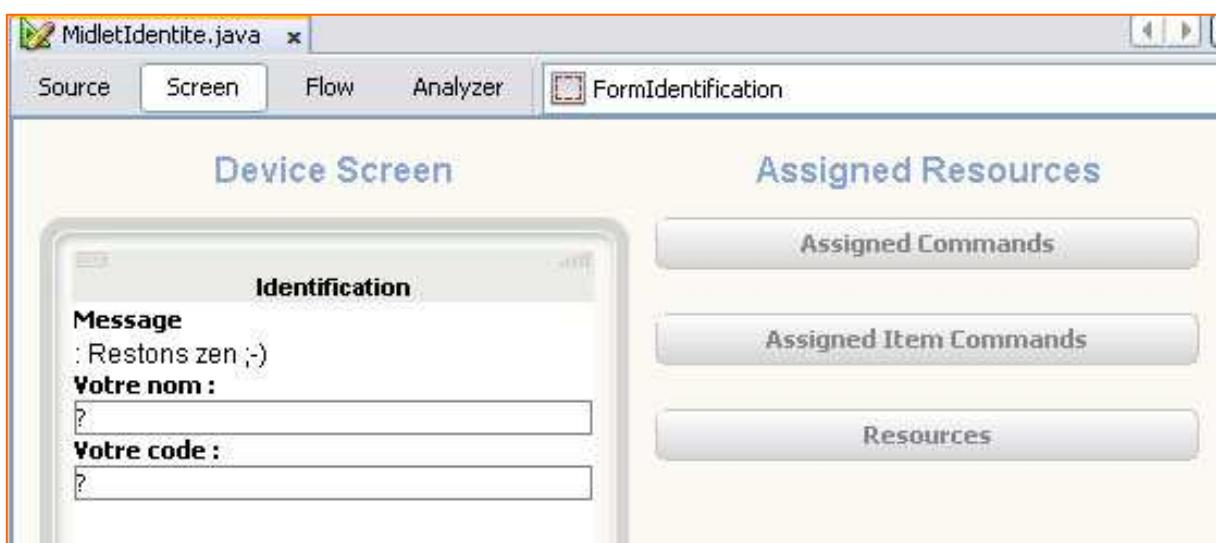
Ensuite, dans l'onglet Screen, nous renommons le formulaire (en Identification) ainsi que son titre qui sera affiché :



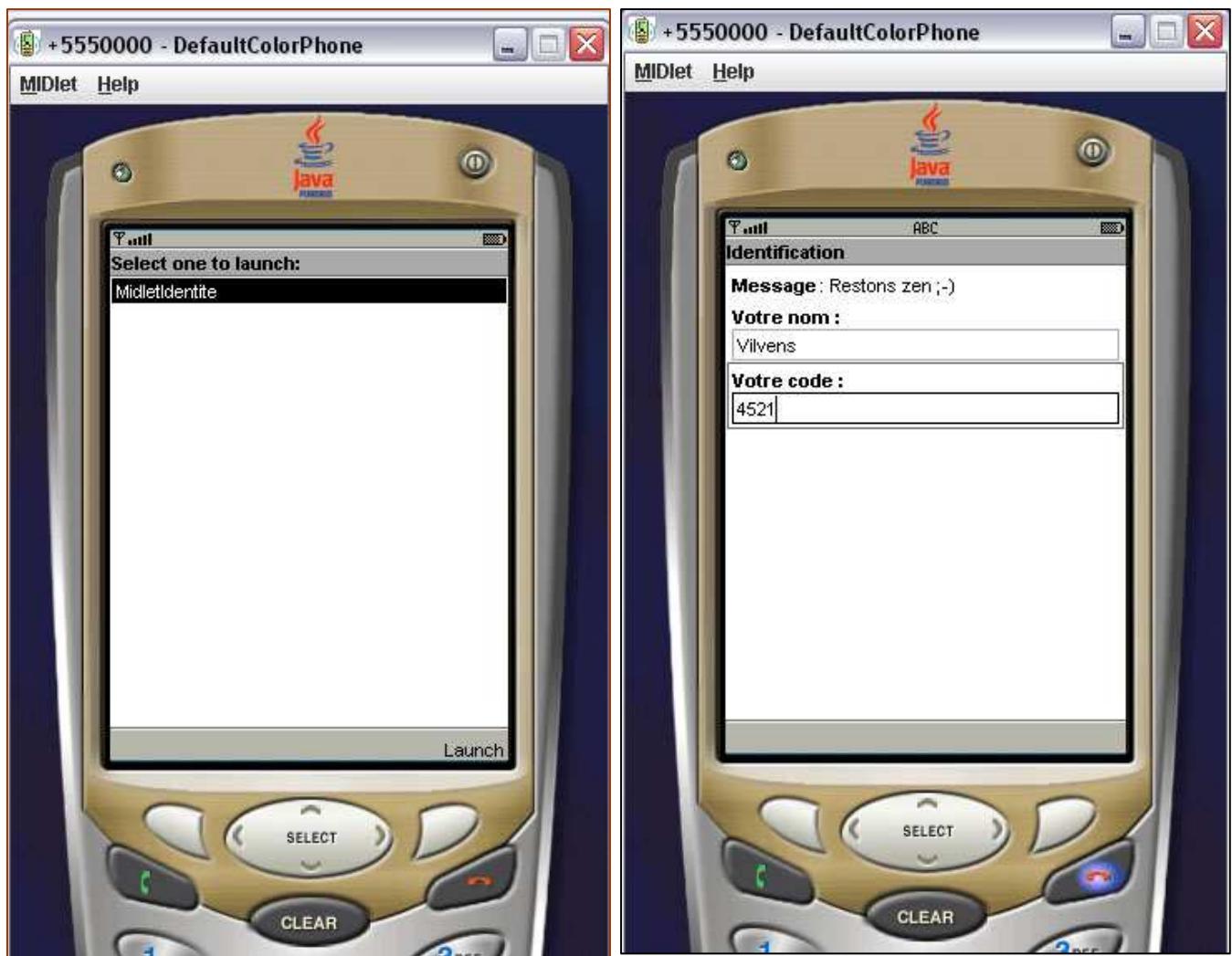
adaptions et enrichissons le formulaire (en glissant des composants de la palette) comme ceci :



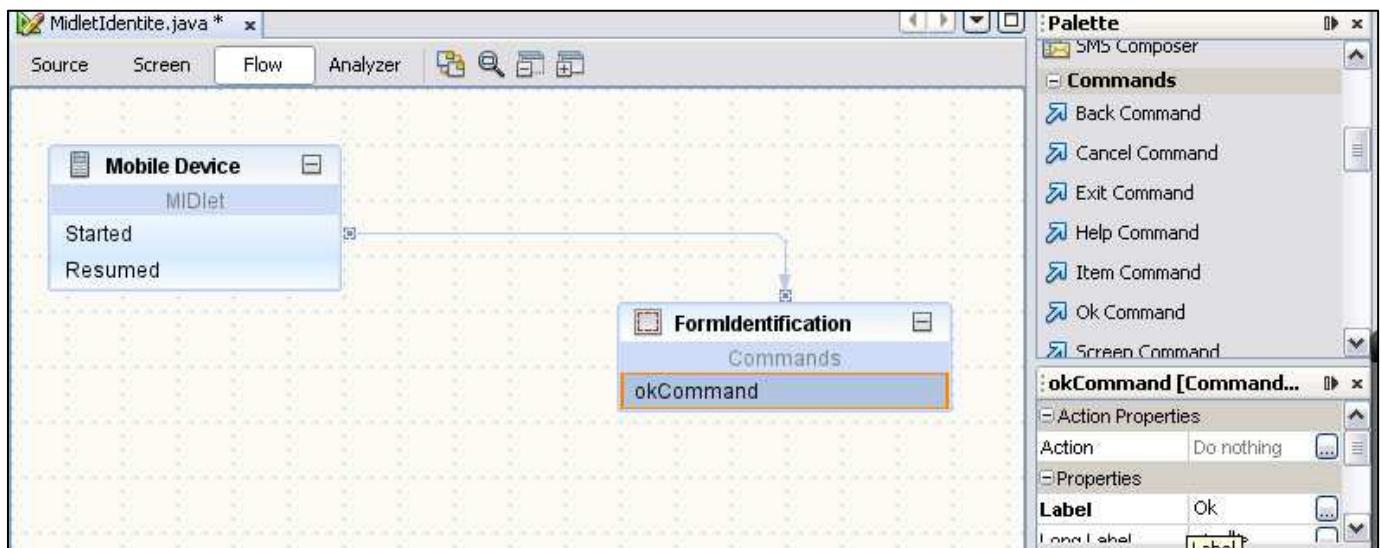
que nous retravaillons en :



A ce stade, l'exécution donne :



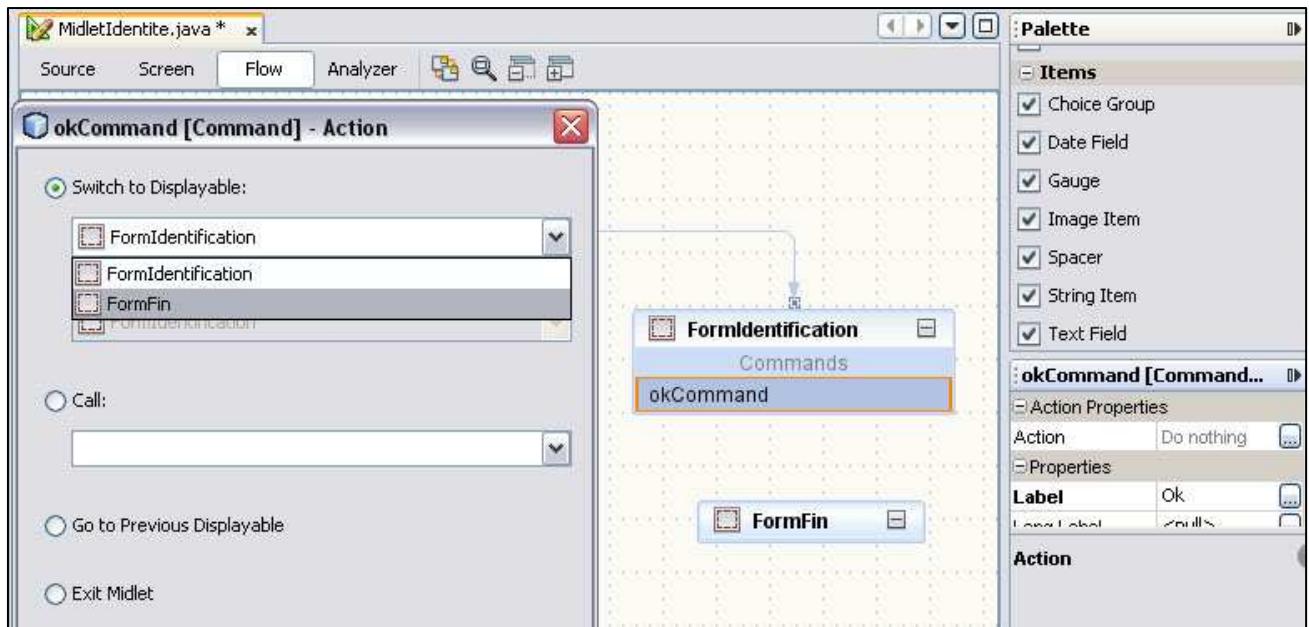
Il nous reste évidemment à ajouter les commandes sur le formulaire et sur ses éléments. Nous ajoutons tout d'abord le mécanisme de traitement de la commande Ok (par simple glissement de la commande Ok sur le formulaire depuis la palette) :



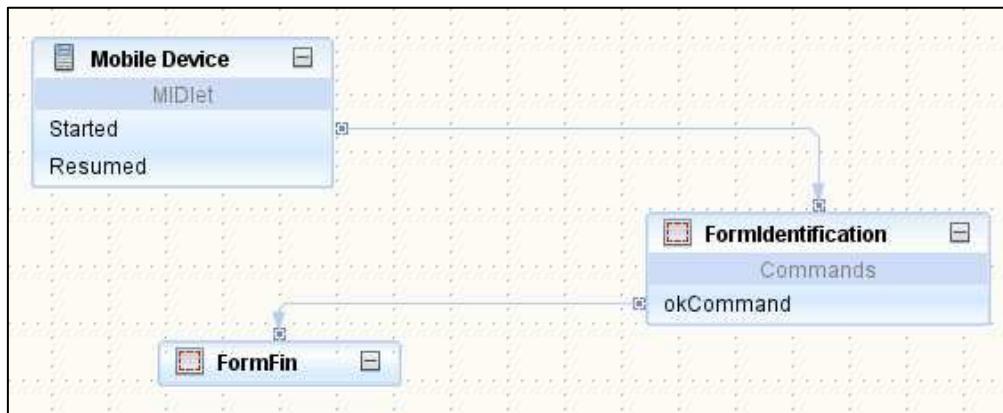
Nous ajoutons un deuxième formulaire de fin d'opérations :



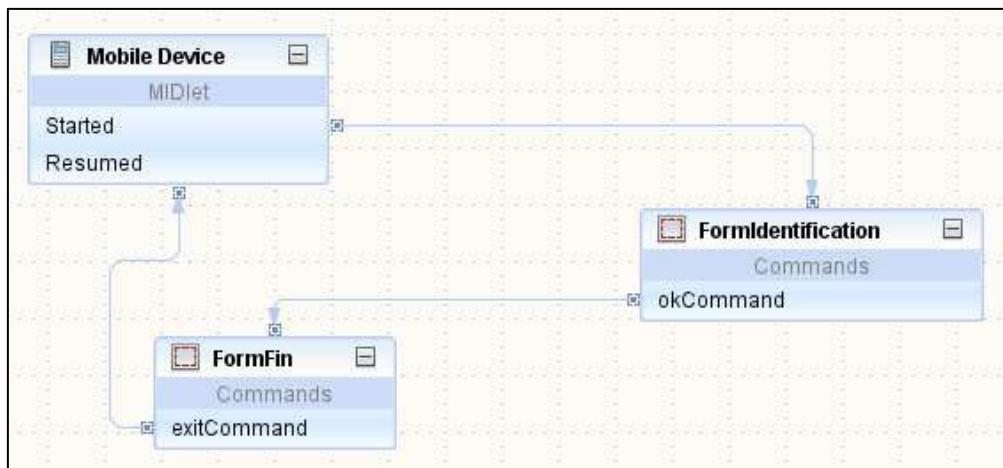
et nous le raccordons à la commande Ok du formulaire principal :



Résultat :



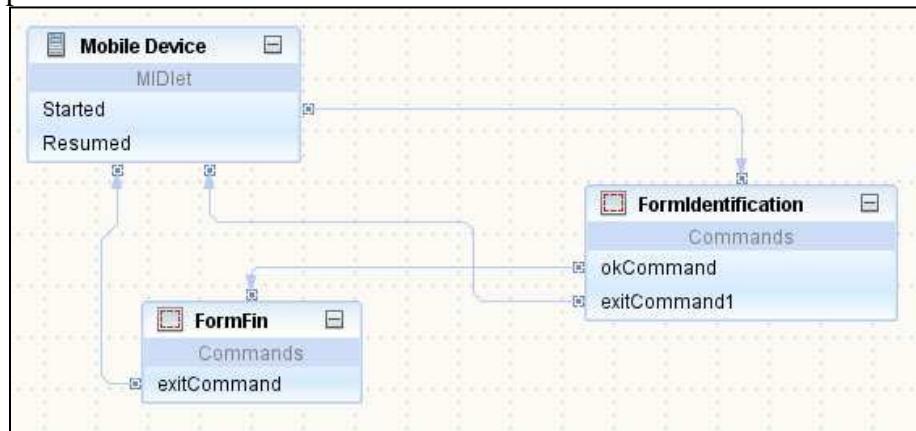
Il reste évidemment à boucler la boucle :



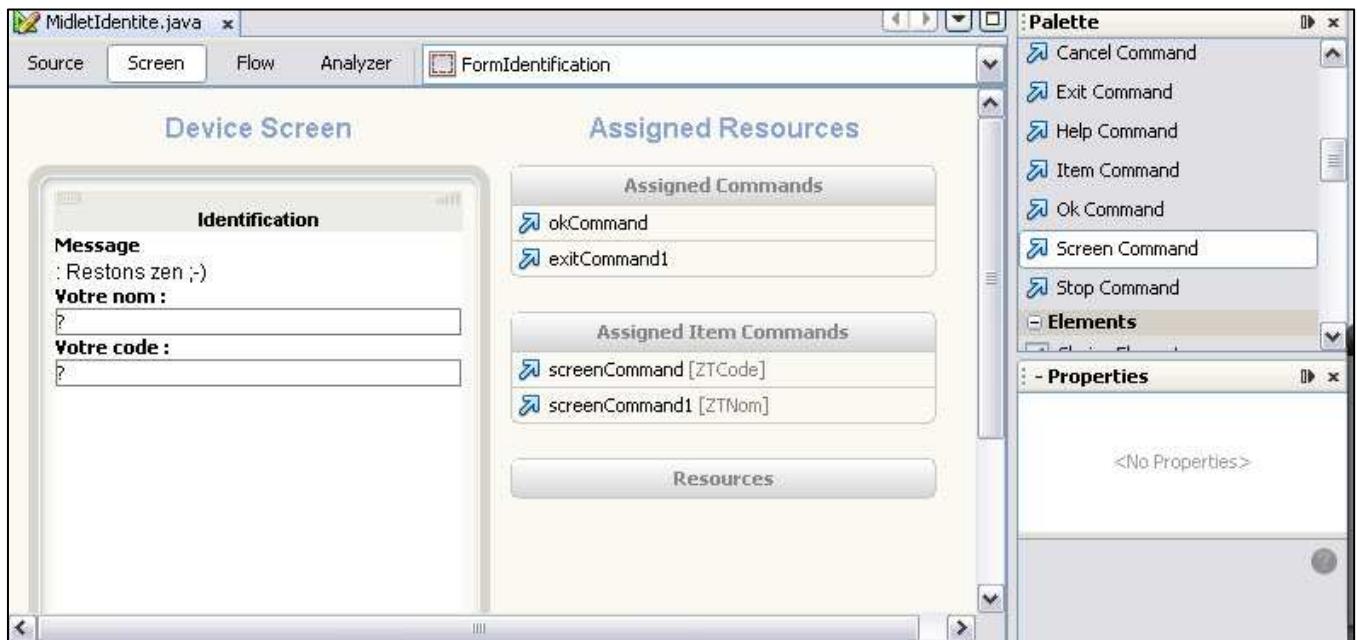
A l'exécution :



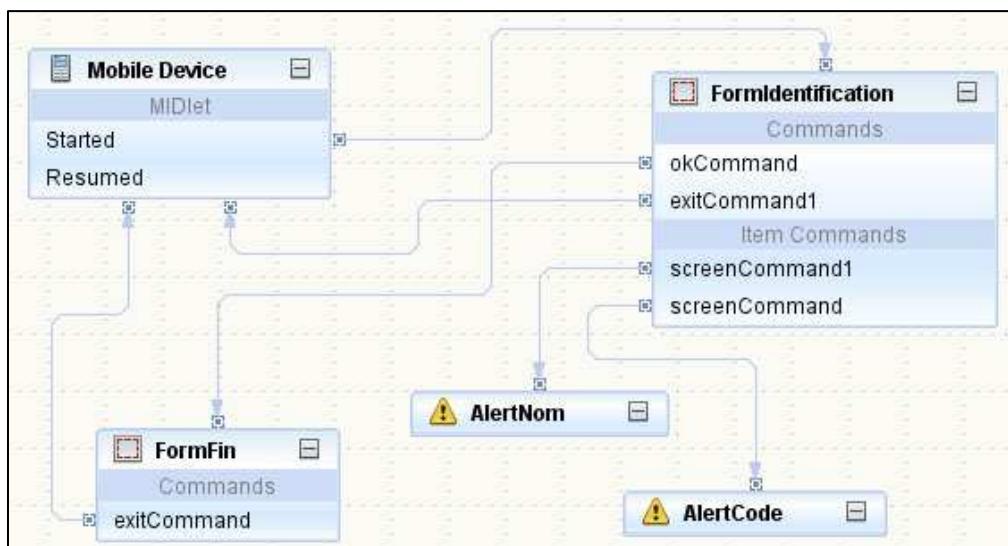
Encore une petite commande exit :



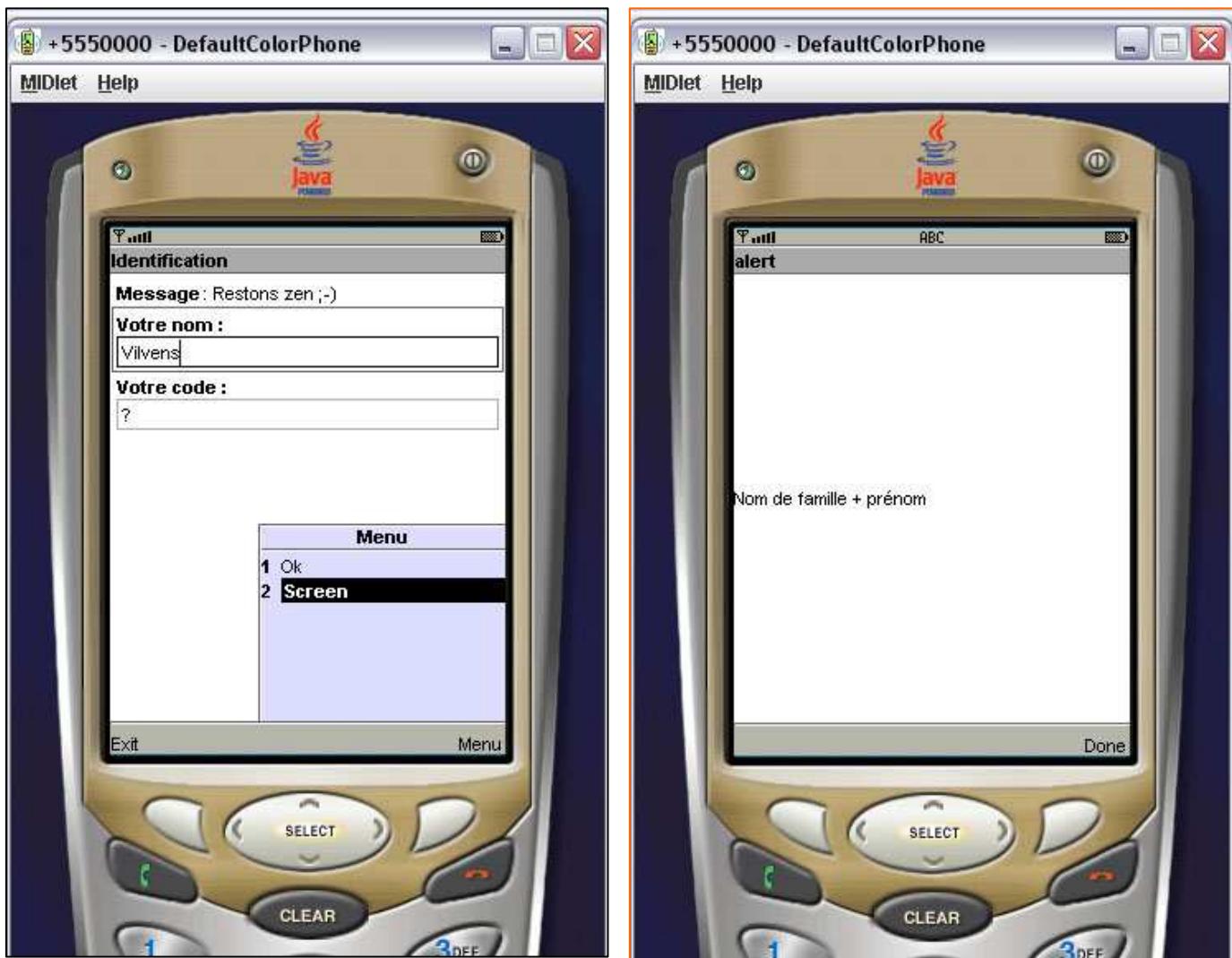
Nous passons pour terminer à des commandes sur les composantes du formulaire : disons que nous affectons la commande screen à chaque zone d'entrée :



et nous associons à chaque commande une boîte d'alerte :



A l'exécution, cela donne, par exemple :



11.2 Une MIDlet sur un serveur Web

Cette modeste MIDlet étant au point, nous allons la placer sur un serveur Web (dans notre cas, nous utiliserons Tomcat 6.0 sur une machine de nom ulyssevil) afin de la rendre accessible et déployable depuis un client Web quelconque (cette opération est souvent désignée sous le nom de *provisioning*). On parle encore de déploiement de **Over-The-Air** (user-initiated) provisioning, en abrégé **OTA**.

Il suffit pour cela

- ◆ de placer le fichier jar (dans notre cas MIDletFormulaireSurSiteWeb.jar) dans le répertoire ROOT (ou dans l'un de ses sous-répertoires);
- ◆ de modifier le fichier jad pour y enregistrer le chemin correct du jar :

MIDletFormulaireSurSiteWeb.jad

MIDlet-1: MIDletIdentite, , MIDletsFormulaires.MIDletIdentite

MIDlet-Jar-Size: 3370

MIDlet-Jar-URL: <http://ulyssevil:8080/MIDletFormulaireSurSiteWeb.jar>

```
MIDlet-Name: MIDletFormulaireSurSiteWeb  
MIDlet-Vendor: Vendor  
MIDlet-Version: 1.0  
MicroEdition-Configuration: CLDC-1.1  
MicroEdition-Profile: MIDP-2.0
```

Il convient encore de vérifier dans le fichier web.xml se trouvant dans le répertoire C:\Program Files\Apache Software Foundation\Tomcat 6.0\conf que les deux mime-mappings suivants sont présents :

web.xml

```
...  
<mime-mapping>  
    <extension>jad</extension>  
    <mime-type>text/vnd.sun.j2me.app-descriptor</mime-type>  
</mime-mapping>  
<mime-mapping>  
    <extension>jar</extension>  
    <mime-type>application/java-archive</mime-type>  
</mime-mapping>  
...
```

Dans le browser, il suffit de taper l'URL

<http://ulyssevil:8080/MIDletFormulaireSurSiteWeb.jad>

pour obtenir tout d'abord :



puis :

Nous allons déployer cette MIDlet sur un mobile qui aura été placé sur le même réseau que le serveur Web. Mais faisons d'abord connaissance avec ce mobile ...

12. Un second exemple de mobile

12.1 Le téléphone tactile Htc Touch



Nous avons donc joint à notre Pocket Pc Palm One un collègue plus récent, le téléphone mobile Htc Touch (du nom de la société asiatique HTC). Techniquement, ce mobile comporte un processeur cadencé à 201 MHz, une ROM de 128 Mo, une SDRAM de 64 Mo, un écran de résolution 240 X 320 pixels avec 65536 couleurs, des connexions Bluetooth 2.0, Wi-Fi IEEE 802.11 b/g et HTC ExtUSB (combiné mini USB et prise audio à 11 broches), un appareil photo CMOS couleur 2 mégapixels.

Il fonctionne sous le système d'exploitation Windows Mobile 6 Professionnel, basé sur un Windows CE 5.2 et plus particulièrement destiné aux Pocket PCs offrant des fonctionnalités de téléphonie mobile (l'édition Classic n'offre pas cette possibilité alors que la version Standard est destinée aux Smartphones).

Ce mobile supporte une machine virtuelle Java Jbed conçue par la société Esmertec pour une configuration CDC et un profil Personal Profile 1.0.

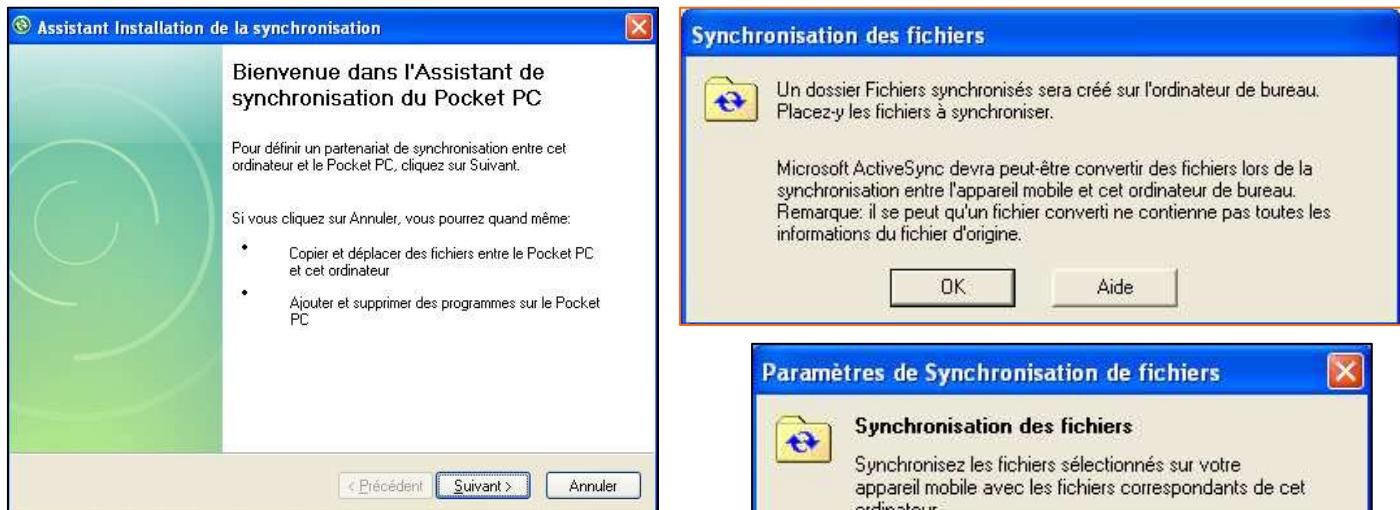


12.2 La synchronisation et Activesync

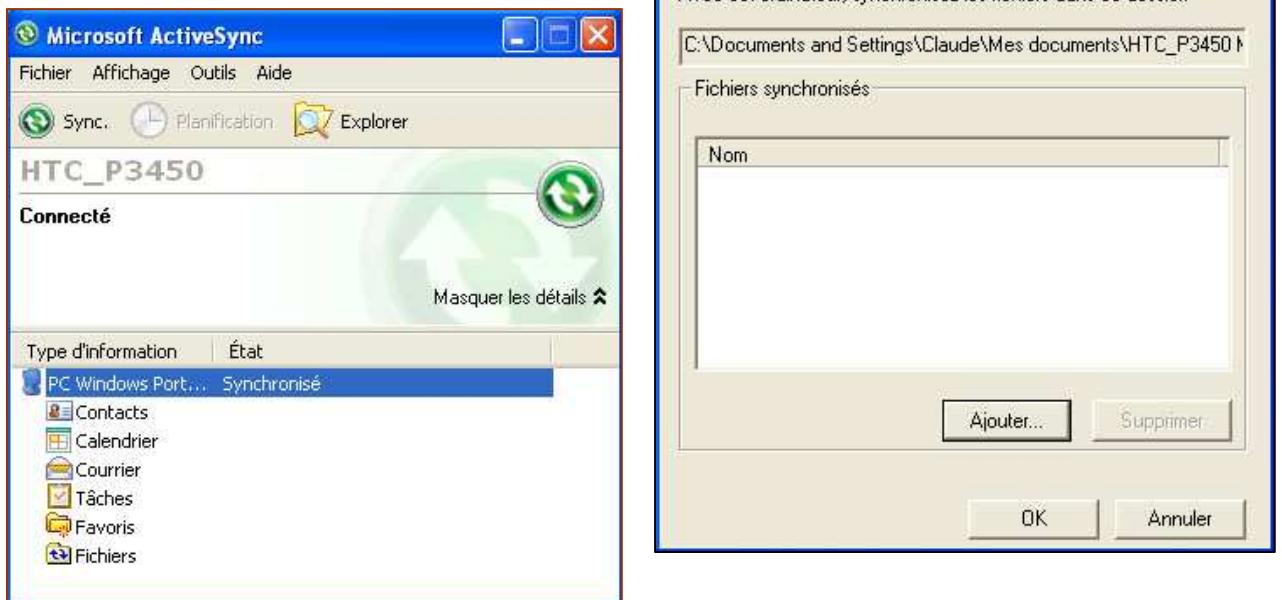
Le programme de synchronisation entre le mobile et le PC doit être ici Microsoft Activesync 4.5 (la très courante version 4.2 est insuffisante) – on peut le télécharger depuis le site de Microsoft (on obtient un fichier setup.msi). Il convient donc de l'installer au préalable sur la machine de développement, ce qui ne pose pas de problème (à remarquer la tentative de localiser Outlook pour les futures opérations de synchronisation et l'étonnement de ne pas le trouver ;-)). Après un redémarrage éventuel, le lancement d'Activesync provoque l'apparition de la fenêtre suivante :



La connexion du mobile va enclencher une seconde phase dont le but est de définir ce qui fera l'objet d'une synchronisation automatique et notamment les répertoires qui contiendront les fichiers à synchroniser :

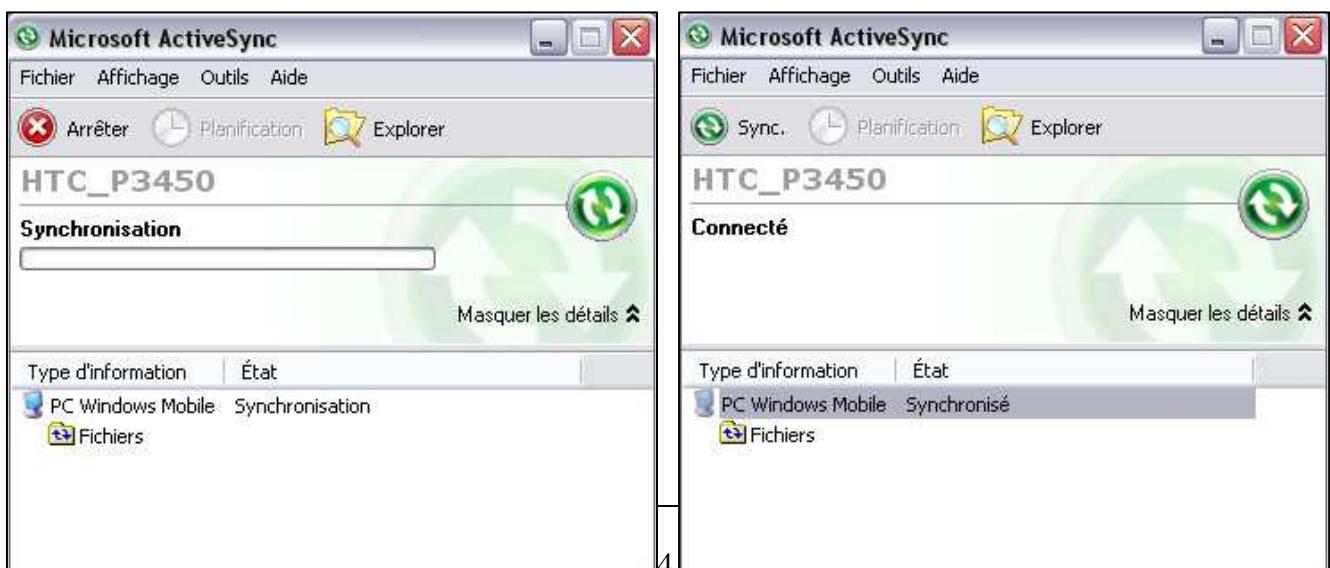


Après une première synchronisation, on obtient :

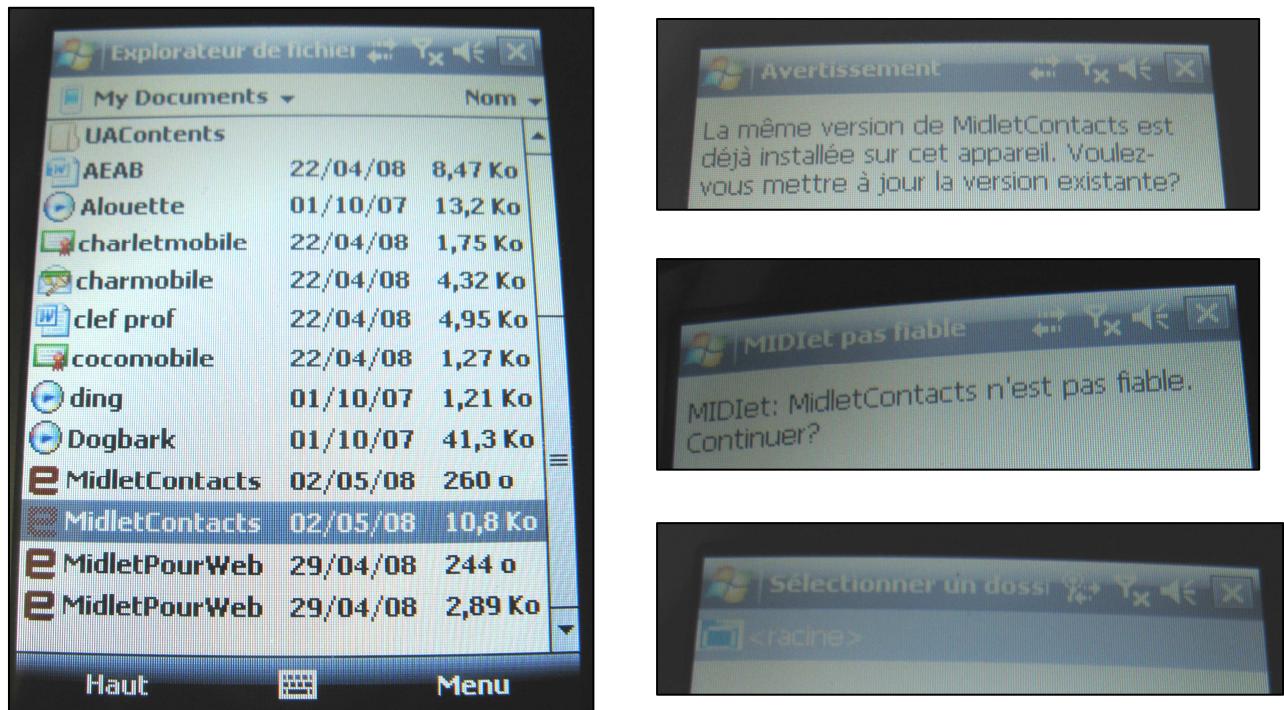


12.3 Le déploiement d'une MIDlet par câble

Il suffit de copier les fichiers jar et jad dans le répertoire associé à "fichiers" (donc, ici C:\Documents and Settings\Claude\Mes documents\HTC_P3450 Mes documents) puis de lancer Activesync :



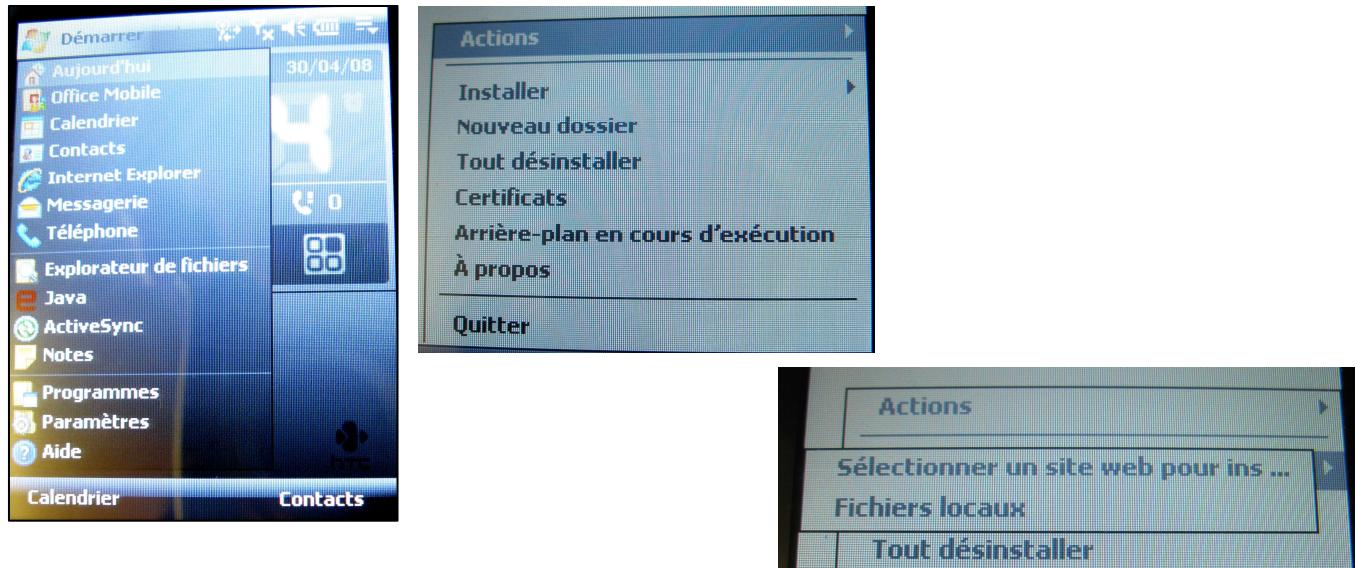
Les fichiers sont donc transférés, mais la MIDlet n'est pas encore installée. Pour cela, il suffit, sur le mobile, d'aller dans l'Explorateur de fichiers et de sélectionner le fichier jar (ici, par exemple, MIDletContacts) :

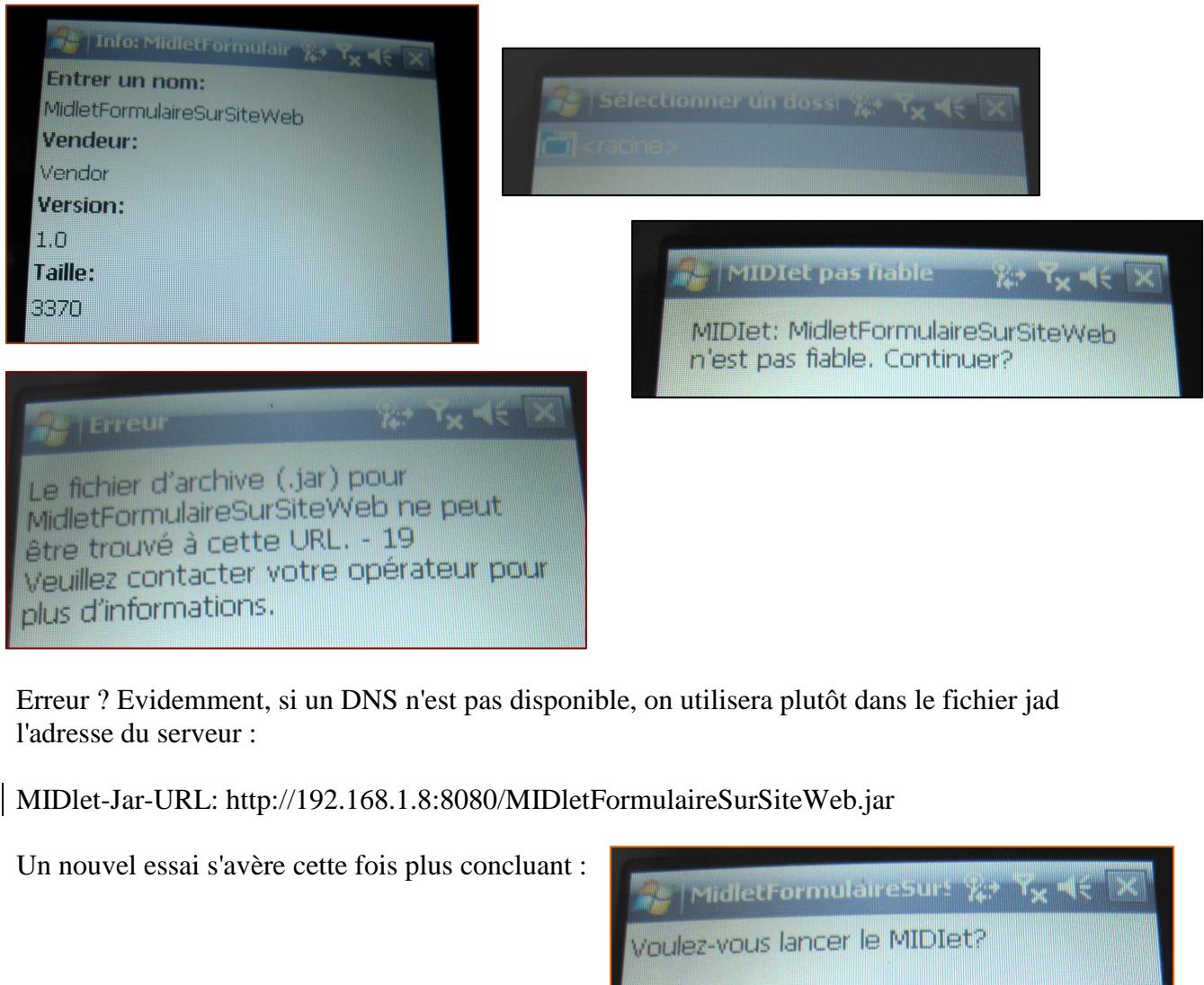


En cas de refus d'installation (erreur 40), il est probable que la MIDlet a été compilée en MIDP 2.1 – le mobile Htc ne supporte que MIDP 2.0. Une fois l'installation, on peut lancer la MIDlet.

12.4 Le déploiement d'une MIDlet par OTA

Notre mobile se trouvant sur le réseau et étant doté d'un browser et d'une machine virtuelle, rien n'empêche le déploiement d'une MIDlet par OTA suivie de son exécution. Dans le menu principal, nous trouvons un item "Java" :

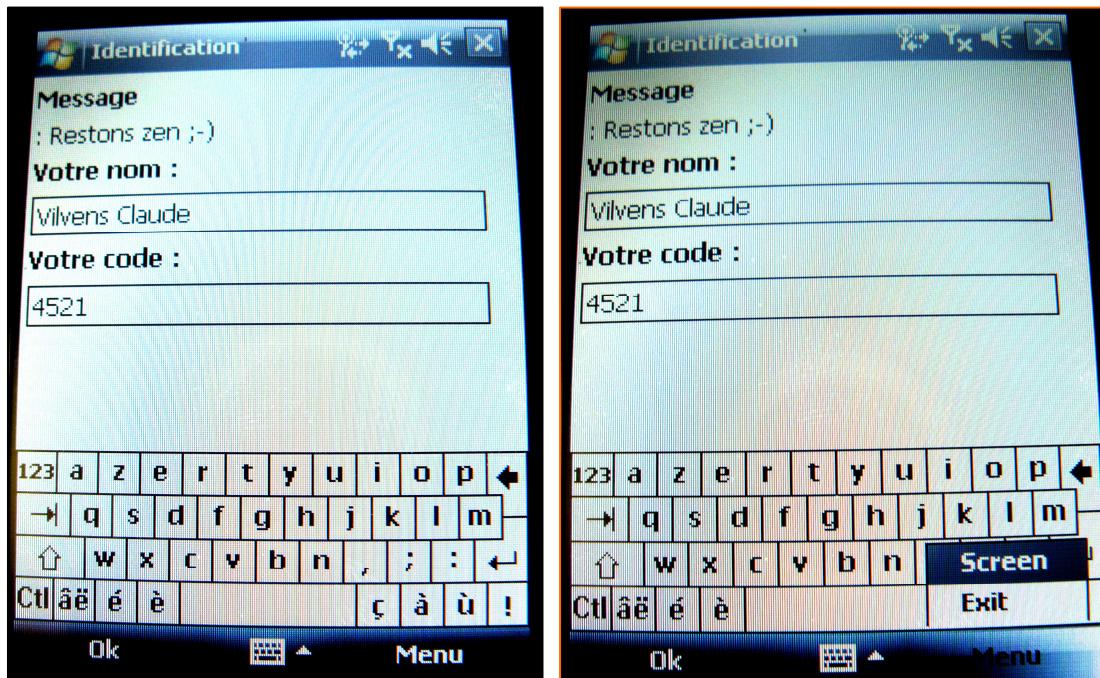


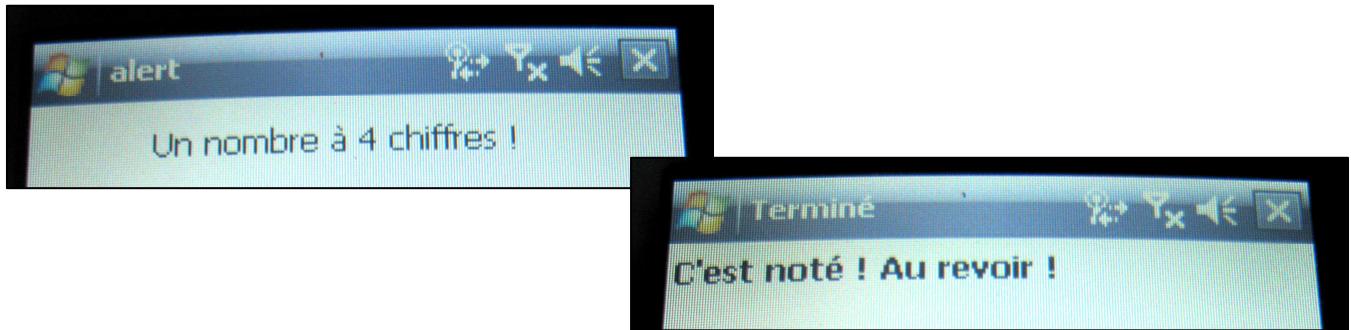


Erreur ? Evidemment, si un DNS n'est pas disponible, on utilisera plutôt dans le fichier jad l'adresse du serveur :

| MIDlet-Jar-URL: <http://192.168.1.8:8080/MIDletFormulaireSurSiteWeb.jar>

Un nouvel essai s'avère cette fois plus concluant :





13. Un formulaire plus élaboré

13.1 La construction du formulaire

Nous allons revoir notre MIDlet à menu en y plaçant, dans le formulaire qui sert de container, trois listes proposant des items prédefinis : ce seront des instances de **ChoiceGroup** (deux zones d'entrée de textes **TextField** compléteront le GUI). Un objet ChoiceGroup s'instancie en utilisant le plus souvent le constructeur

```
public ChoiceGroup (String label, int choiceType, String[] stringElements,  
Image[] imageElements)
```

qui rappelle évidemment un constructeur de la classe List, avec pour le troisième paramètre la possibilité supplémentaire

```
public static final int POPUP
```

à la fonction évidente. Comme pour les TextBox, le dernier paramètre permet de spécifier une contrainte sur les entrées en utilisant les mêmes constantes de classe que celles qu'utilisent les TextBox.

Nous pouvons donc compléter notre MIDlet comme ceci :

MenuMIDletHeavy.java (2)

```
/*  
 * MenuMIDlet.java  
 * Created on 6 octobre 2005, 10:33  
 */  
  
package hello;  
  
import javax.microedition.MIDlet.*;  
import javax.microedition.lcdui.*;  
  
/**  
 * @author Vilvens  
 * @version  
 */  
public class MenuMIDletHeavy extends MIDlet implements CommandListener  
{  
    private Display display;  
    private List Menu;  
    private Command exitListes;
```

```

private Form GUIListes;
private ChoiceGroup Categorie, Niveau, Epoque;
private TextField TfNom, TfTelephone ;

public MenuMIDlet()
{
    display = Display.getDisplay(this);
    String[] elements = { "Listes", "Jeux", "Mémoire" };
    exitListes = new Command("Listes", Command.EXIT, 1);

    Menu = new List ("Menu", List.IMPLICIT, elements, null);
    Menu.addCommand(exitListes);
    Menu.setCommandListener(this);
}

public void startApp() { display.setCurrent(Menu); }
public void pauseApp() { }
public void destroyApp(boolean unconditional) { }

public void commandAction(Command cmd, Displayable s)
{
    try
    {
        if(cmd == exitListes) s.setTitle(Menu.getString(Menu.getSelectedIndex()));
        if (cmd == List.SELECT_COMMAND) s.setTitle("Choix effectué !");
        {
            System.out.println("Item de Menu Selected");
            GUIListes = new Form("Listes");

            String[] ElementsCategorie = { "Privé", "Professionnel", "Distractions" };
            Categorie = new ChoiceGroup("Catégories", Choice.EXCLUSIVE,
                ElementsCategorie,null);

            String[] ElementsSecurite = { "A partager", "Réservé", "Top secret" };
            Niveau = new ChoiceGroup("Sécurité", Choice.POPUP,ElementsSecurite ,null);

            String[] ElementsEpoque = { "Vacances", "Fin d'année", "Anniversaires",
                "Déprime" };
            Epoque = new ChoiceGroup("Epoques", Choice.MULTIPLE,ElementsEpoque
                ,null);

            TfNom = new TextField ("Nom :","",30,TextField.ANY);
            TfTelephone = new TextField ("Téléphone :","",30,TextField.ANY);

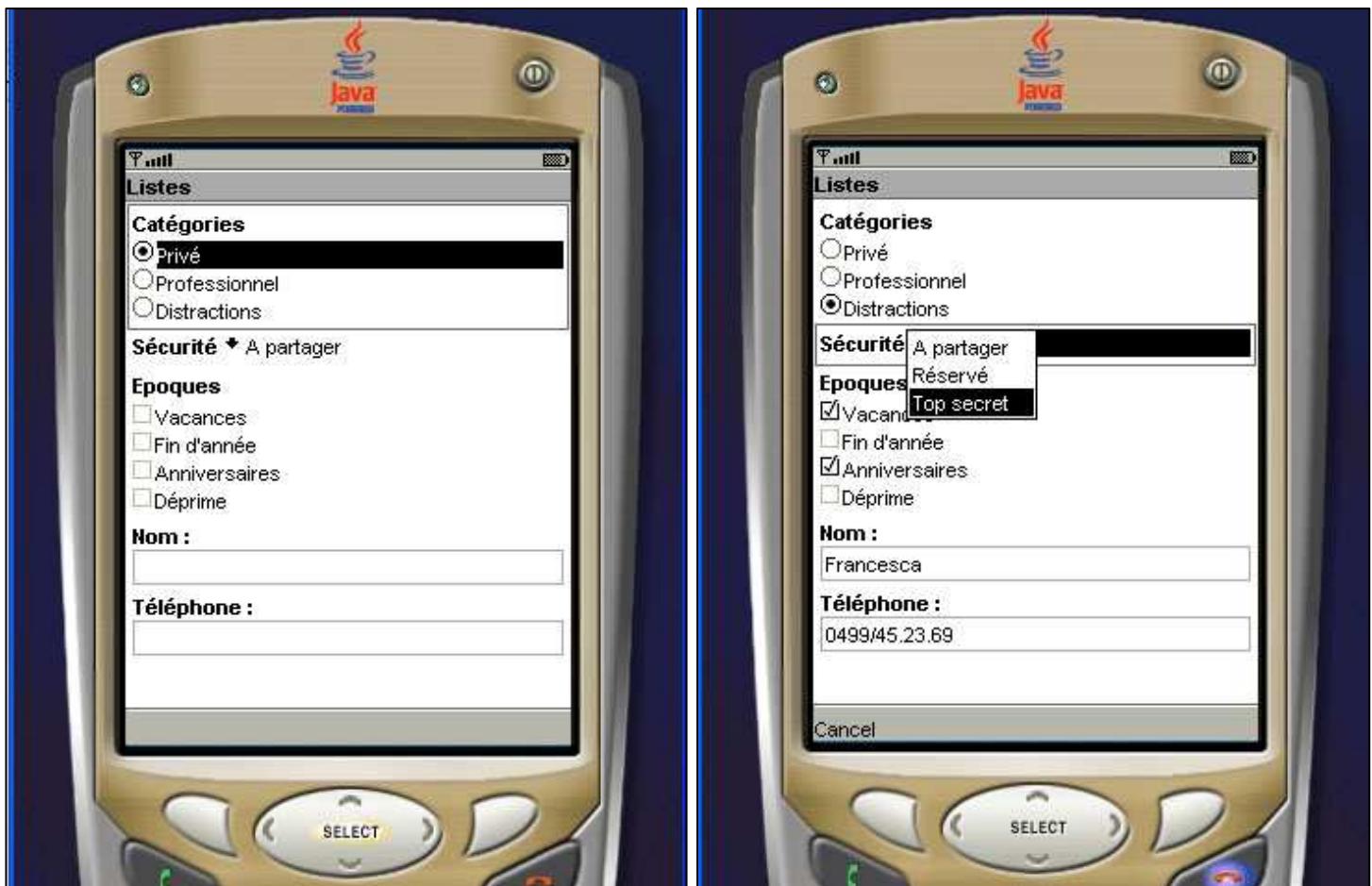
            GUIListes.append(Categorie); GUIListes.append(Niveau);
            GUIListes.append(Epoque);
            GUIListes.append(TfNom); GUIListes.append(TfTelephone);

            GUIListes.setCommandListener(this); //Enregistrement aux Listener.
        }
    }
}

```

```
        display.setCurrent(GUIListes);           //Affichage du formulaire.  
    }  
}  
catch ( Exception e) {  System.out.println(e);  }  
}  
}
```

avec pour résultat :



13.2 L'interaction avec le formulaire

Toute modification de l'état d'un formulaire, par interaction avec l'un de ses composants, provoque la génération d'un événement de type JavaBean. Celui qui veut réagir à cette intervention peut être un objet quelconque implémentant l'interface **ItemStateListener**, qui comporte la méthode :

```
public void itemStateChanged (Item item)
```

Le plus souvent, à nouveau, c'est l'application elle-même qui est le listener :

```
public class HelloMIDlet extends MIDlet implements ItemStateListener  
{ ... }
```

à condition, comme toujours, de s'enregistrer comme tel au moyen de la méthode de Form :

public void **setItemStateListener** (ItemStateListener l)

Donc ici :

GUIListes.**setItemStateListener**(this);

Il reste dès lors à effectivement doter notre midlet de la méthode itemStateChanged() : ici, elle se contentera de recopier la sélection réalisée dans une zone de texte additionnelle (appelons ce TextField "TfOperation"). Cependant, quand on y réfléchit, on se rend compte que le ChoiceGroup "Epoque" est à sélection multiple : il se pourrait donc que plusieurs items soient sélectionnés ... Comment les obtenir ? En fait, la classe ChoiceGroup (comme d'ailleurs la classe List à laquelle elle est sémantiquement apparentée) possède une méthode :

public int **getSelectedFlags** (boolean[] selectedArray_return)

qui met à true les booléens occupant une position correspondant à un item sectionné dans le ChoiceGroup – il suffit donc de les tester ...

MenuMidlet.java (3)

```
/*
 * MenuMidlet.java
 */

package hello;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

/**
 * @author Vilvens
 */

public class MenuMidlet extends MIDlet
    implements CommandListener, ItemStateListener
{
    private Display display;
    private List Menu;
    private Command exitListes;
    private Form GUIListes;
    private ChoiceGroup Categorie, Niveau, Epoque;
    private TextField TfNom, TfTelephone, TfOperation ;

    public MenuMidlet() { ... }
    public void startApp() { ... }
    ...

    public void commandAction(Command cmd, Displayable s)
    {
```

```

try
{
    if(cmd == exitListes)
        s.setTitle(Menu.getString(Menu.getSelectedIndex()));
    if (cmd == List.SELECT_COMMAND) s.setTitle("Choix effectué !");
    {
        GUIListes = new Form("Listes");
        String[] ElementsCategorie = { "Privé", "Professionnel", "Distractions" };
        Categorie = new ChoiceGroup ("Catégories",
            Choice.EXCLUSIVE,ElementsCategorie,null);
        ...
        TfOperation = new TextField("Dernière sélection :","",30,TextField.ANY);

        GUIListes.append(Categorie); ... GUIListes.append(TfOperation);

        GUIListes.setItemStateListener(this);
        display.setCurrent(GUIListes);      }
    }
    catch ( Exception e) { System.out.println(e);  }
}

public void itemStateChanged (Item item)
{
    String s = new String();

    if (item == Categorie)
        s = "Catégorie = "+ Categorie.getString(Categorie.getSelectedIndex());
    else if (item == Niveau)
        s = "Niveau = "+ Niveau.getString(Niveau.getSelectedIndex());
    else
    {
        //s = "Epoque = "+ Epoque.getString(Epoque.getSelectedIndex()); // NON :-(

        boolean flagsItemsSelectionnes[] = new boolean[Epoque.size()];
        Epoque.getSelectedFlags(flagsItemsSelectionnes);
        s = "Epoques = ";
        for (int i = 0; i < Epoque.size(); i++)
            if(flagsItemsSelectionnes[i]) s += " "+Epoque.getString(i);
    }

    TfOperation.setString(s);
}
}

```

On remarquera la méthode de ChoiceGroup :

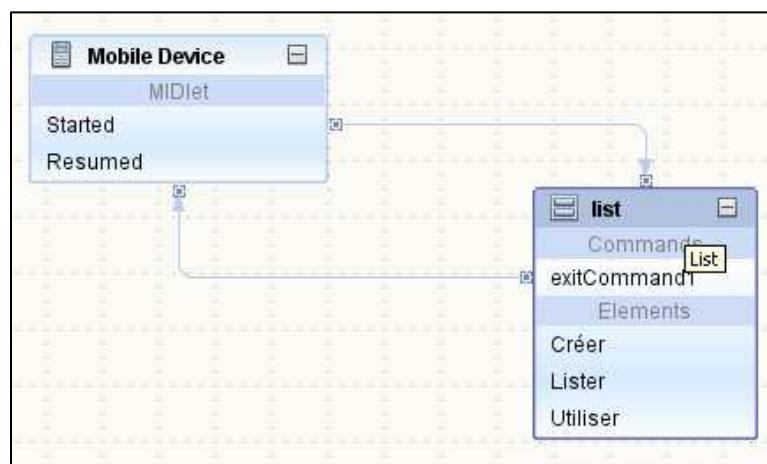
`public int size()`

qui fournit le nombre total d'items du ChoiceGroup. On aura donc alors, par exemple :

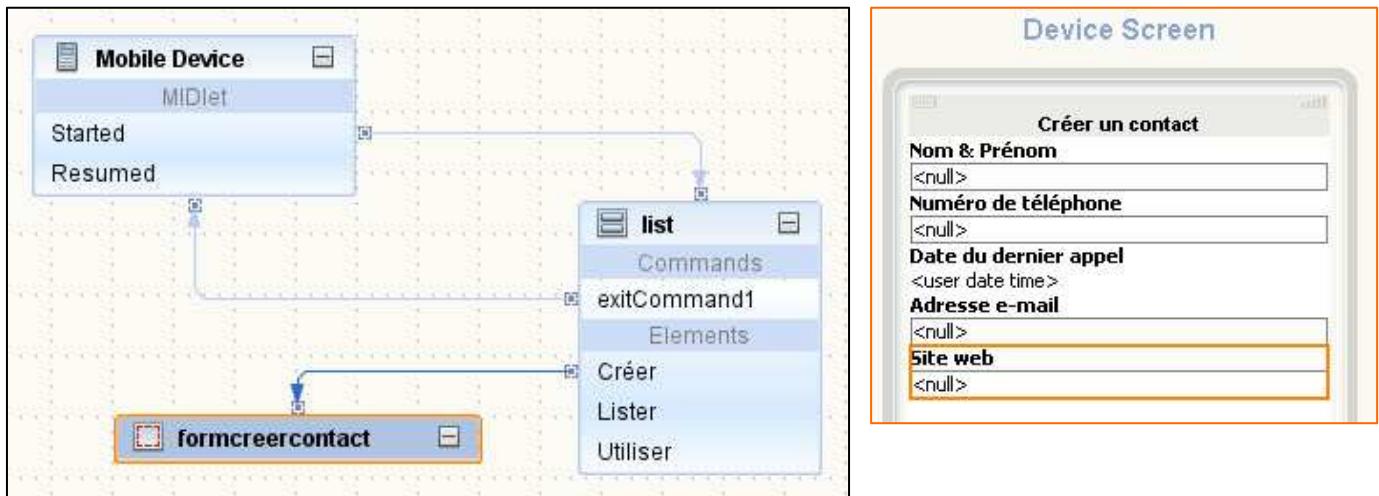


14. Une application de gestion de contacts : GUI de création

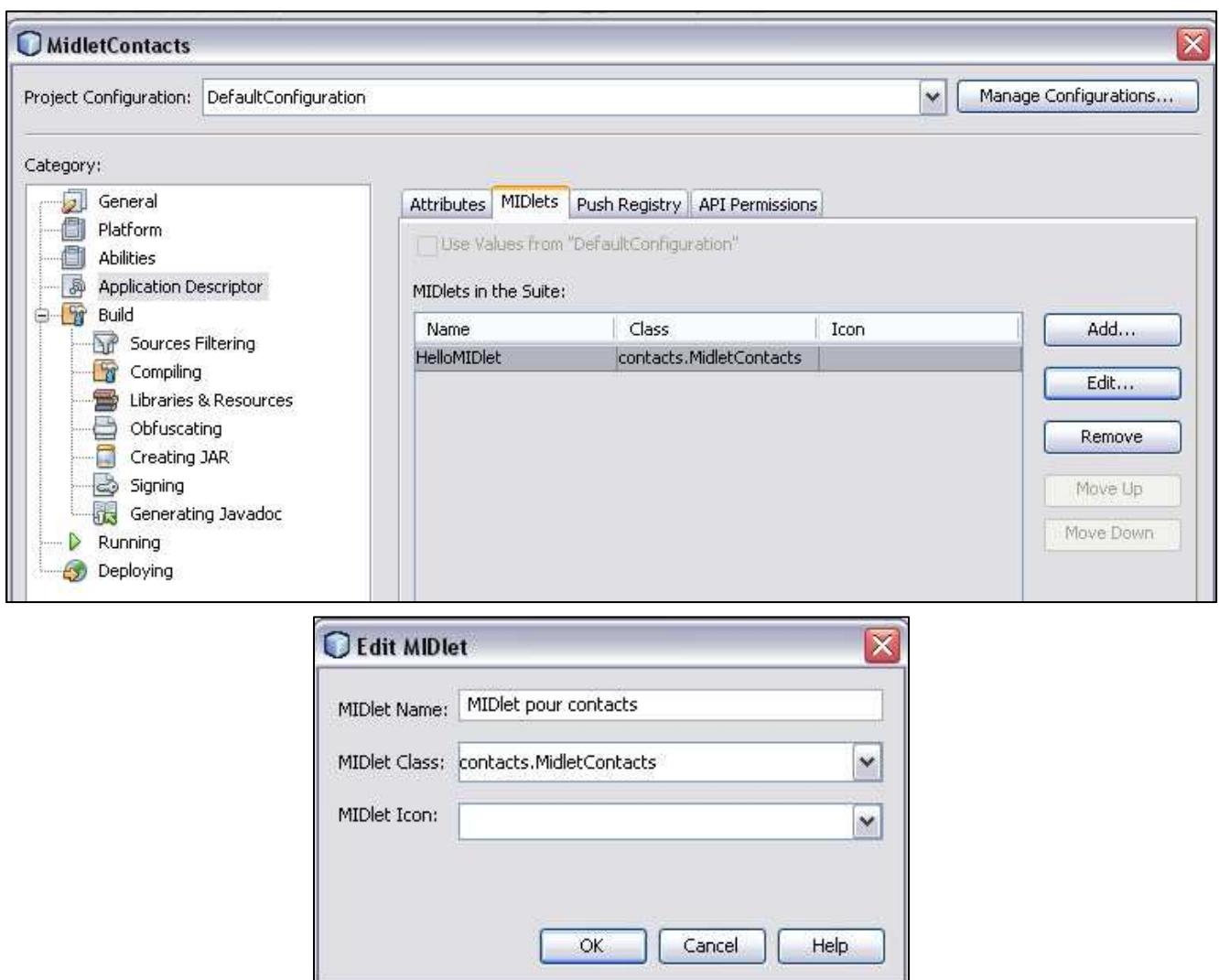
Attaquons-nous à présent à une application mobile qui va manipuler des données persistantes. Une fois le projet créé, nous commençons par remplacer le formulaire par défaut par une liste :



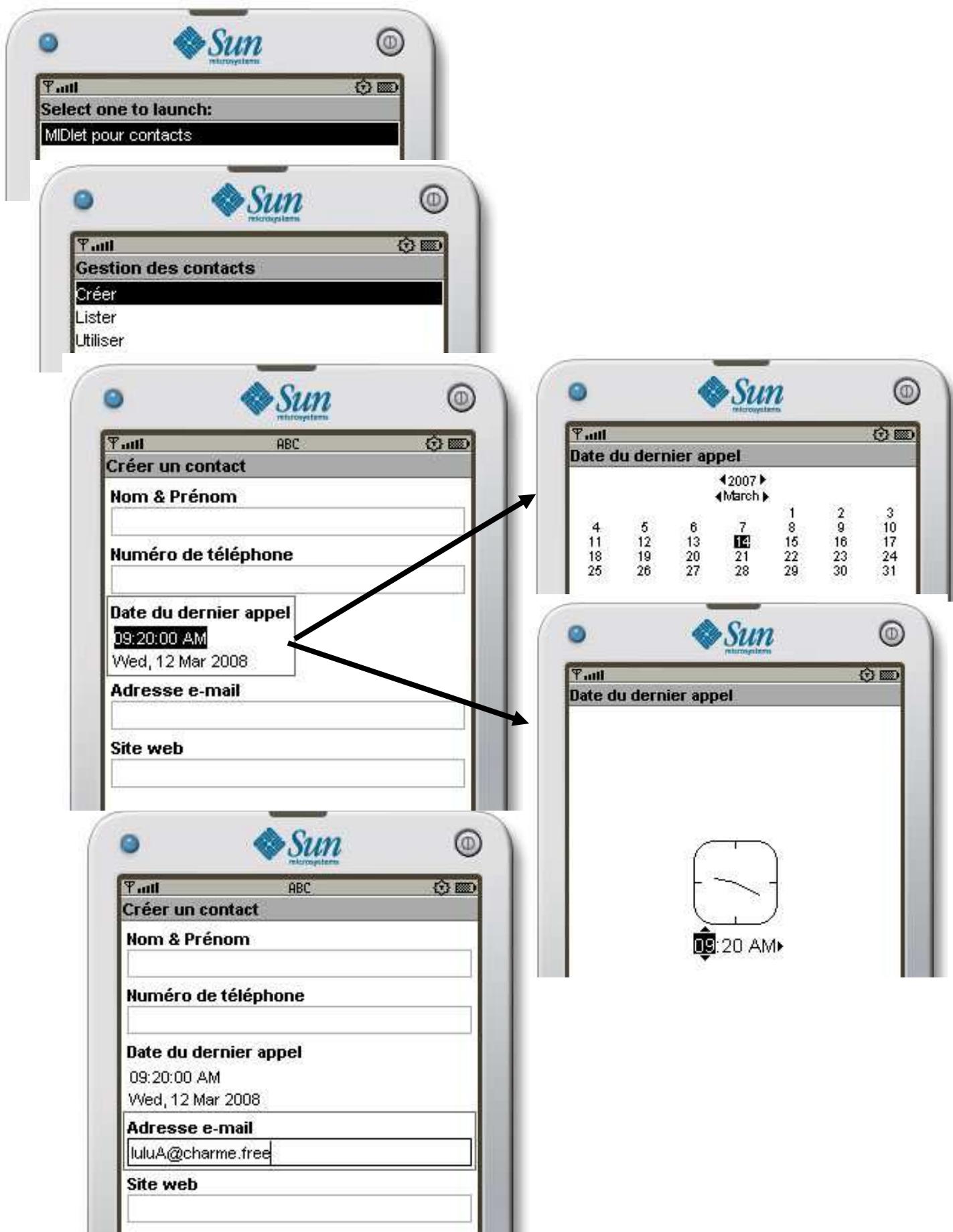
Nous associons à notre item de liste un formulaire d'entrée des données :



Nous pouvons aussi modifier le nom afficher sur le mobile pour repérer notre MIDlet (son "descripteur d'application"). Il suffit d'édition les propriétés du projet :



Un exemple d'exécution montre que tout se passe bien :



Cependant, notre curiosité naturelle nous pousse à inspecter le code généré. Nous pouvons y relever (en nous concentrant seulement sur l'essentiel) :

MIDletContacts.java

```
package contacts;

import javax.microedition.MIDlet.*;
import javax.microedition.lcdui.*;

/**
 * @author Vilvens
 */

public class MIDletContacts extends MIDlet implements CommandListener
{

    public void startMIDlet()
    {
        switchDisplayable(null, getList());
    }

    public List getList()
    {
        if (list == null)
        {
            list = new List("Gestion des contacts", Choice.IMPLICIT);
            list.append("Cr\u00e9er", null);
            list.append("Lister", null); list.append("Utiliser", null);
            list.addCommand(getExitCommand1());
            list.setCommandListener(this);
            list.setSelectedFlags(new boolean[] { false, false, false });
        }
        return list;
    }

    public Command getExitCommand1()
    {
        if (exitCommand1 == null)
        {
            exitCommand1 = new Command("Exit", Command.EXIT, 0);
        }
        return exitCommand1;
    }

    public void commandAction(Command command, Displayable displayable)
    {
        if (displayable == formcreercontact)
        {
            if (command == exitCommand2) {}
```

```

        }
    else if (displayable == list)
    {
        if (command == List.SELECT_COMMAND)
        {
            listAction();
        }
        else if (command == exitCommand1)
        {
            exitMIDlet();
        }
    }
}

public void listAction()
{
    String __selectedString = getList().getString(getList().getSelectedIndex());
    if (__selectedString != null)
    {
        if (__selectedString.equals("Cr\u00e9er"))
        {
            switchDisplayable(null, getFormcreercontact());
        }
        else if (__selectedString.equals("Lister")) {}
        else if (__selectedString.equals("Utiliser")) {}{}
    }
}

public Form getFormcreercontact()
{
    if (formcreercontact == null)
    {
        formcreercontact = new Form("Cr\u00e9er un contact",
            new Item[] { getTextField(), getTextField1(), getDateField(), getTextField2(),
                getTextField3() });
        formcreercontact.addCommand(getExitCommand2());
        formcreercontact.setCommandListener(this);
    }
    return formcreercontact;
}

```

On remarquera la méthode

`public void setSelectedFlags(boolean[] selectedArray)`

qui permet d'initialiser un vecteur d'état des items de la liste – on pourra par la suite les consulter par :

`public int getSelectedFlags(boolean[] selectedArray_return)`

On s'en doute, rien ne sert d'encoder des contacts si ce n'est pas pour les mémoriser de manière persistante. Nous pouvons toujours créer une méthode **ajouteContacts()** : mais comment procéder ? Nous avons clairement besoin ici d'un système de fichiers sur le mobile lui-même, dont la nature est pourtant fort variable.

15. Une application de gestion de contacts : enregistrement des contacts

15.1 Le stockage des données en J2ME et la classe RecordStore

Il se fait que le package **javax.microedition.rms** propose des APIs permettent le sauvegarde permanent de données : il s'agit d'un système de fichiers nommé **RMS** (Record Management System), articulé à partir de la classe **RecordStore** (package **javax.microedition.rms**), et dont les APIs sont de niveau suffisamment élevé pour être indépendantes du type de mobile.

Dans ce contexte, un fichier classique à enregistrements (les obsédés de SQL diront "une table") est une instance de la classe **RecordStore**. *Les données y sont stockées sous forme de bytes*. Chaque enregistrement est numéroté (à partir de 1) par un "recordId" qui est son identifiant (les obsédés de SQL diront qu'il s'agit de la "clé primaire").

Un objet **RecordStore** est ouvert au moyen de la méthode de classe polymorphe **open()**, dont nous utiliserons la version :

```
public static RecordStore openRecordStore  
    (String recordStoreName, boolean createIfNecessary)  
    throws RecordStoreException, RecordStoreFullException,  
          RecordStoreNotFoundException
```

dont on aura compris que le deuxième paramètre détermine si on crée **RecordStore** qui n'existe pas encore ou pas. On remarquera les exceptions éventuellement lancées : la classe **RecordStoreException** (dérivée de **Exception** au sein du package **java.microedition.rms**), assez générale, possède quelques classes dérivées plus spécialement dédiées à des problèmes particuliers (par exemple, un espace de stockage plein).

Le **RecordStore** ainsi créé (ou ouvert) est en fait lié à la MIDlet par l'intermédiaire du nom fourni. Une MIDlet ne peut donc accéder qu'à des **RecordStore** créés par elle. Elle peut obtenir la liste des **RecordStore** disponibles au moyen de la méthode de classe

```
public static String[] listRecordStores()
```

On s'en doute, on referme ce que l'on a ouvert au moyen de la méthode

```
public void closeRecordStore()  
    throws RecordStoreNotOpenException, RecordStoreException
```

Cependant, il convient d'être prudent pour éviter les exceptions : le nombre de **close()** exécutés doit correspondre exactement au nombre d'**open**; comme il n'est pas possible de savoir si un **RecordStore** est ouvert ou non, le plus simple est d'ouvrir-agir-fermer ...

15.2 L'enregistrement dans un RecordStore

Dans le cadre de notre modeste application, il est clair que nous allons mémoriser des contacts simplissimes, limités à un nom, un numéro de téléphone, etc. Cependant, même pour atteindre cet objectif limité, il faut réaliser que :

- ◆ une classe **Contact** sera sans doute la bienvenue afin de matérialiser le futur enregistrement à mémoriser; les variables membres vont de soi, si ce n'est que ...
- ◆ il ne faut pas oublier qu'un record doit posséder un **recordId**; on peut pour cela utiliser la méthode de RecordStore :

```
public int getNextRecordID()
    throws RecordStoreNotOpenException, RecordStoreException
```

qui fournit évidemment le prochain numéro de recordId utilisable; ce numéro n'est valide que pour un RecordStore ouvert et jusqu'à la prochaine écriture;

- ◆ il ne faut pas non plus oublier non plus que les données sont stockées sous forme de **bytes**, pas sous forme d'objets sérialisés; il sera donc judicieux de prévoir une méthode fournissant les bytes représentant un objet contact (appelons-la finement `getBytes()` par analogie avec la classe `String`) et, à l'inverse, un constructeur créant un objet `Contact` à partir d'un tableau de tels bytes.

Toutes ces réflexions nous mènent à la classe suivante :

Contact.java

```
package contacts;

/**
 * @author Vilvens
 */

import java.io.*;

public class Contact
{
    private String nom, numTelephone, adresseEMail, nomSiteWeb;
    private int id;

    public Contact (String n, String nt, String aem, String nsw, int i)
    {
        nom = n; numTelephone = nt;
        adresseEMail = aem; nomSiteWeb = nsw;
        id = i;
    }

    public Contact (byte[] b)
    {
        try
        {
            ByteArrayInputStream bis = new ByteArrayInputStream(b);
            DataInputStream dis = new DataInputStream(bis);

            nom = dis.readUTF(); numTelephone = dis.readUTF();
            nomSiteWeb = dis.readUTF(); adresseEMail = dis.readUTF();
        }
    }
}
```

```
    id = dis.readInt();
    dis.close();
}

catch(Exception e)
{
    System.out.println("Erreur dans la lecture de bytes pour Contact");
}
}

public String toString()
{
    return getNom() + " : " + getNumTelephone()
        + " -- " + this.getAdresseEMail() + " / " + this.getNomSiteWeb()
        + " (" + getId() + ")";
}

public byte[] getBytes()
{
    try
    {
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(bos);

        dos.writeUTF(getNom()); dos.writeUTF(getNumTelephone());
        dos.writeUTF(nomSiteWeb); dos.writeUTF(adresseEMail);
        dos.writeInt(getId());
        dos.close();

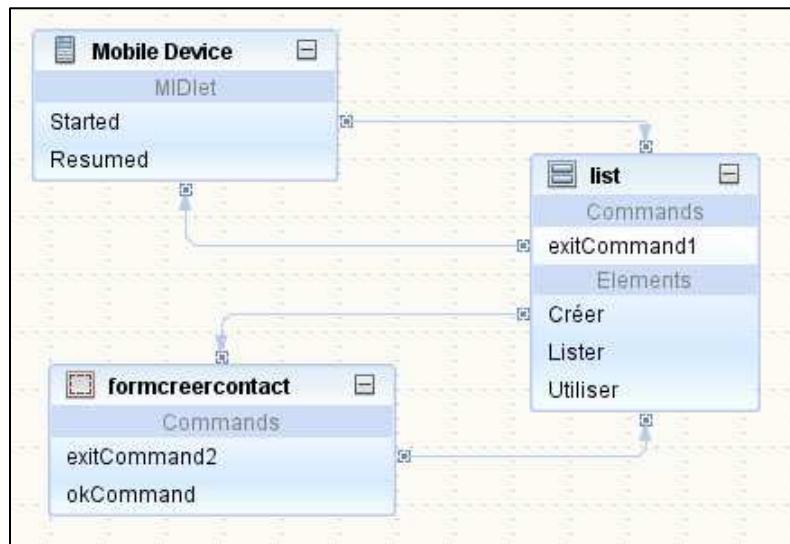
        return bos.toByteArray();
    }
    catch(Exception e)
    { return new byte[] { }; }
}

public String getNomSiteWeb() { return nomSiteWeb; }
public void setNomSiteWeb(String nomSiteWeb)
{ this.nomSiteWeb = nomSiteWeb; }
public String getAdresseEMail() { return adresseEMail; }
public void setAdresseEMail(String adresseEMail)
{ this.adresseEMail = adresseEMail; }
public String getNom() { return nom; }
public void setNom(String nom)
{ this.nom = nom; }
public String getNumTelephone() { return numTelephone; }
public void setNumTelephone(String numTelephone)
{ this.numTelephone = numTelephone; }
public int getId() { return id; }
public void setId(int id) { this.id = id; }
}
```

L'écriture proprement dite du record dans le RecordStore s'effectuera avec la méthode

```
public int addRecord (byte[] data, int offset, int numBytes)
throws RecordStoreNotOpenException, RecordStoreException,
RecordStoreFullException
```

où tout semble clair. Nous pouvons donc doter notre MIDlet de la gestion d'une commande Ok qui permettra de provoquer l'enregistrement des données entrées dans le GUI par appel d'une méthode ajouteContacts(), la commande Exit provoquant le retour à la liste générale :



avec

MenuMIDlet.java (4)

```
...
import javax.microedition.rms.*;
...
public class MenuMIDlet extends MIDlet implements CommandListener, ItemStateListener
{ ...
    private RecordStore fichierContacts; ...

    public void commandAction(Command command, Displayable displayable)
    {
        if (displayable == formcreercontact)
        {
            if (command == exitCommand2)
            {
                switchDisplayable(null, getList());
            }
            else if (command == okCommand)
            {
                ajouteContacts();
            }
        }
    }
}
```

```

public void ajouteContacts()
{
    try
    {
        fichierContacts = RecordStore.openRecordStore("Contacts", true);

        Contact c = new Contact(ZTNom.getString(), ZTTelephone.getString(),
                               ZTemail.getString(), ZTsiteweb.getString(), fichierContacts.getNextRecordID());

        fichierContacts.addRecord(c.getBytes(), 0, c.getBytes().length);
        fichierContacts.closeRecordStore();
        System.out.println("Enregistrement effectué");
        System.out.println(" --> " + c.toString());
    }
    catch (RecordStoreException e)
    {
        System.out.println("Erreur RMS : " + e.getMessage());
    }
}
}

```

Lors de l'exécution avec l'émulateur, on peut vérifier que l'enregistrement est bien effectué.
Mouais ... Il faudrait peut-être vérifier !

16. Une application de gestion de contacts : liste des contacts

16.1 La lecture des enregistrements d'un RecordStore

Bien logiquement, nous allons à présent relire nos contacts mémorisés. Pour ce faire, on dispose de la méthode de RecordStore :

```

public RecordEnumeration enumerateRecords (RecordFilter filter,
                                         RecordComparator comparator, boolean keepUpdated)
                                         throws RecordStoreNotOpenException

```

qui fournit une liste de records sous forme d'un objet implémentant l'interface **RecordEnumeration** (c'est l'équivalent d'un Resultset ou d'un curseur de SGBD – on peut voir aussi une analogie avec les itérateurs de containers) Dans ce prototype assez complexe, les deux premiers paramètres implémentent les interfaces qui correspondent respectivement à la sélection de certains records seulement et l'ordre dans lequel on souhaite obtenir ces records – ces deux références peuvent être à null). Le troisième paramètre, un booléen, précise si les modifications apportées au RecordStore depuis la constitution de la liste doivent être pris en compte ou non – autrement dit, "snapshot or not snapshot".

Le parcours de l'objet énumération obtenu se programme sans difficulté au moyen de la méthode de RecordEnumeration :

```

public byte[] nextRecord()
throws InvalidRecordIDException, RecordStoreNotOpenException, RecordStoreException

```

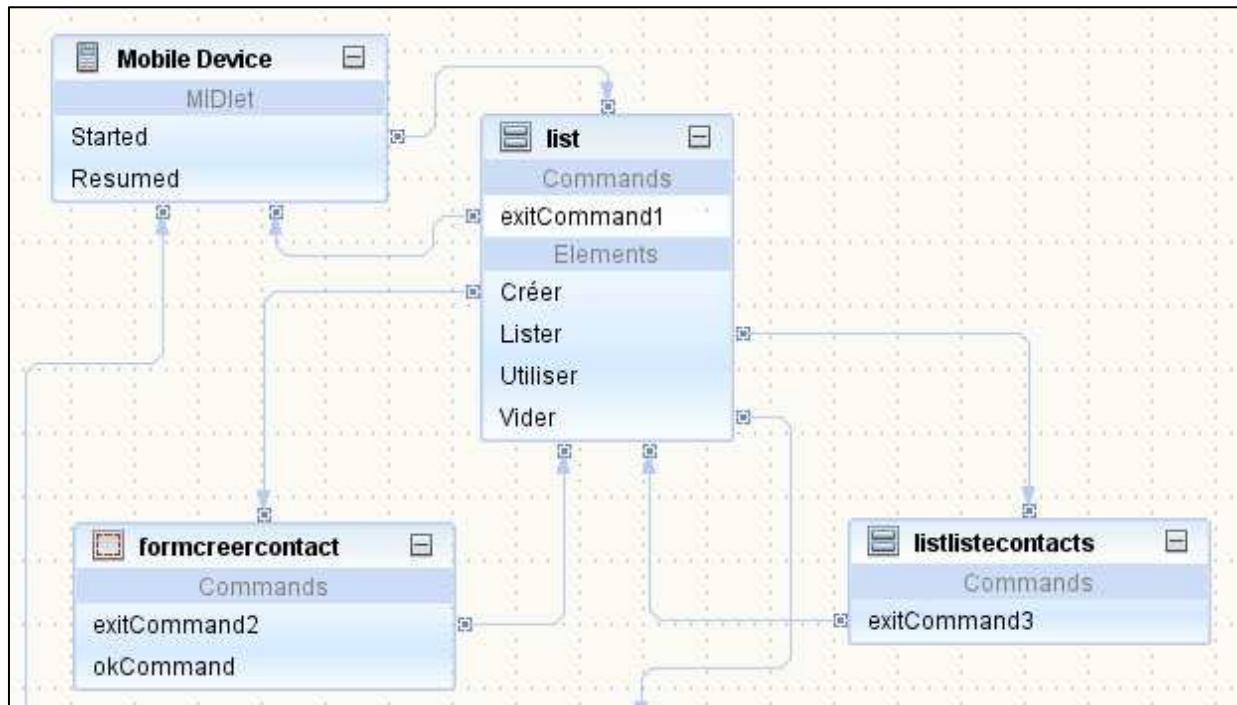
où la notion de "record suivant" étant définie sur la base éventuelle des filtre et comparateur utilisés si ils ont été définis, et de l'autre méthode

`public boolean hasNextElement()`

Bien sûr, les méthodes symétriques remplaçant "next" par "previous" sont également disponibles.

16.2 L'affichage des contacts

Nous ajoutons à notre application un objet List :



et la fonctionnalité de listing des contacts sera donc :

MenuMIDlet.java (5)

```

...
public class MenuMIDlet extends MIDlet implements CommandListener
{
    ...
    public void commandAction(Command command, Displayable displayable)
    {
        if (displayable == formcreercontact)
        { ... }
        else if (displayable == list)
        {
            if (command == List.SELECT_COMMAND)
            {
                listAction();
            }
        }
    }
}
  
```

```

public void listAction()
{
    String __selectedString = getList().getString(getList().getSelectedIndex());
    if (__selectedString != null)
    {
        if (__selectedString.equals("Cr\u00e9er"))
        {
            switchDisplayable(null, getFormcreercontact());
        }
        else if (__selectedString.equals("Lister"))
        {
            switchDisplayable(null, getListlistecontacts());
        }
    }
}

public List getListlistecontacts()
{
    if (listlistecontacts == null)
    {
        listlistecontacts = new List("Liste des contacts enregistr\u00e9s",
            Choice.IMPLICIT);
        listlistecontacts.addCommand(getExitCommand3());
        listlistecontacts.setCommandListener(this);
        listerContacts(listlistecontacts); // write post-init user code here
    }
    return listlistecontacts;
}

public void listerContacts(List li)
{
    try
    {
        fichierContacts = RecordStore.openRecordStore("Contacts", true);
        RecordEnumeration re = fichierContacts.enumerateRecords(null, null, true);
        int nRec = re.numRecords();
        if(nRec != 0)
            for(int i = 0 ; re.hasNextElement() ; i++)
            {
                byte[] contactEnBytes = re.nextRecord();
                Contact c = new Contact(contactEnBytes);
                li.append(c.getNom(), null);
                System.out.println(c.toString());
            }
        fichierContacts.closeRecordStore();
    }
    catch (RecordStoreException e)
    {
        System.out.println("Erreur RMS : " + e.getMessage());
    }
}

```

Si donc on a créé quelques contacts, par exemple :



avec dans la console :

Enregistrement effectué
-->Lulu Dupont : 0800696969 -- luluA@charme.free / www.distractions.lu (3)

on obtient ensuite :



Il serait évidemment logique de prévoir une interaction sur la liste – à faire ;-)

17. Une application de gestion de contacts : effacer tous les contacts

On peut toujours supprimer un RecordStore au moyen de la méthode

```
public void deleteRecord(int recordId)  
throws RecordStoreNotOpenException, InvalidRecordIDException, RecordStoreException
```

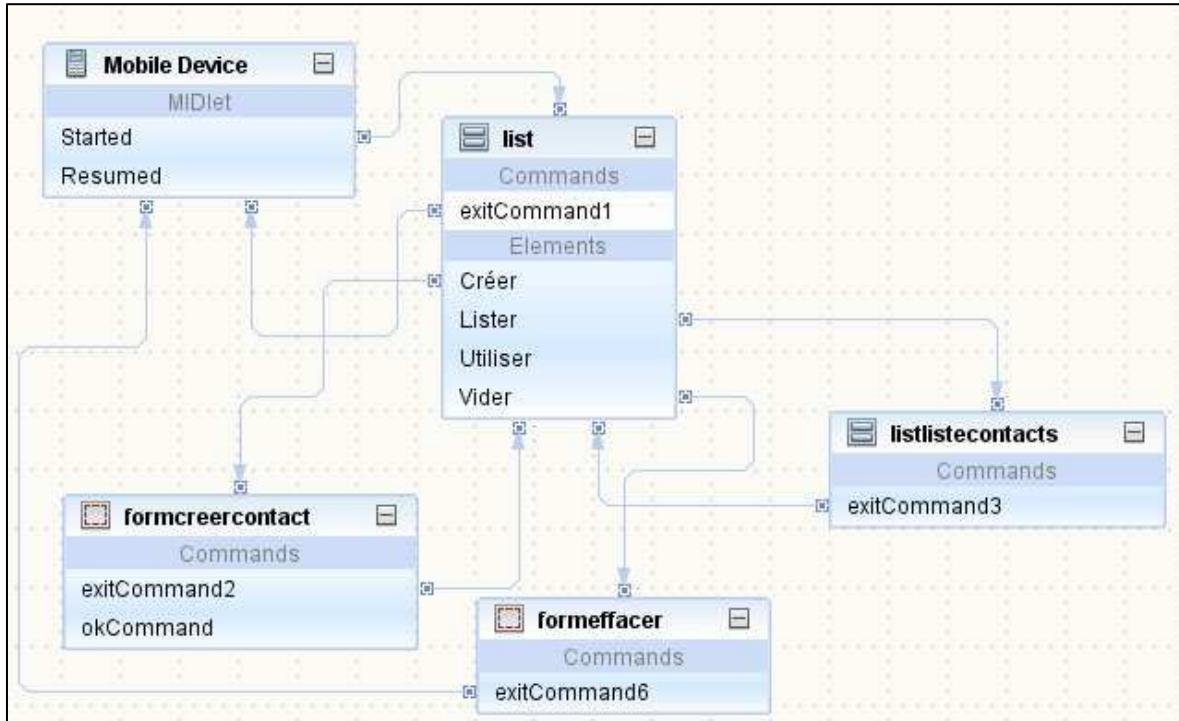
ou de la méthode de classe

```
public static void deleteRecordStore (String recordStoreName)  
throws RecordStoreException, RecordStoreNotFoundException
```

On peut supprimer l'entièreté du RecordStore avec

```
public static void deleteRecordStore(String recordStoreName)
    throws RecordStoreException, RecordStoreNotFoundException
```

Très simplement, nous allons ajouter cette fonctionnalité à notre application avec l'item "Vider" :



avec

MenuMIDlet.java (6)

```
...
public void listAction()
{
    String __selectedString = getList().getString(getList().getSelectedIndex());
    if (__selectedString != null)
    {
        if (__selectedString.equals("Cr\u00e9er")) {...}
        ...
        else if (__selectedString.equals("Vider"))
        {
            switchDisplayable(null, getFormeffacer());
        }
    }
}

public Form getFormeffacer()
{
    if (formeffacer == null)
    {
        formeffacer = new Form("Effacement du RecordStore");
        formeffacer.addCommand(getExitCommand6());
```

```

        formeEffacer.setCommandListener(this);
        effacerTousContacts();
    }
    return formeEffacer;
}

public void effacerTousContacts()
{
    try
    {
        RecordStore.deleteRecordStore("Contacts");
    }
    catch (RecordStoreNotFoundException e)
    {
        System.out.println("Erreur RMS : " + e.getMessage());
    }
    catch (RecordStoreException e)
    {
        System.out.println("Erreur RMS : " + e.getMessage());
    }
}
}

```

18. Les communications réseaux

18.1 Le framework GCF

Le profil MIDP fournit des fonctionnalités réseau spécifiquement orientées vers le monde des mobiles : il s'agit du **GCF** (**G**eneric **C**onnection **F**ramework). L'idée est de fournir un ensemble de classes permettant des communications réseaux classiques (TCP, UDP, HTTP)

- ◆ simples : des objets **Connector** et **Connection** permettent d'obtenir très facilement les flux de communication;
- ◆ uniformisées : tous les acteurs du réseau sont identifiés au moyen d'**URLs**;
- ◆ génériques : un système d'interfaces et de factories assure l'adaptabilité aux différents types de mobiles et à leurs possibilités.

Prévu initialement pour la seule configuration CLDC, le GCF a été étendu au CDC et même ... à J2SE car sa simplicité d'emploi a séduit plus d'un développeur. Le schéma de programmation est en effet simple :

- ◆ on obtient un objet matérialisant la connexion au destinataire visé en utilisant la méthode **open()** de la classe **Connector**;
- ◆ l'objet connexion obtenu est en fait une implémentation d'interfaces divers (tous dérivés de **Connection**) que la factory **open()** aura construite en fonction du format de l'url (notamment du protocole qui y apparaît);
- ◆ cet objet **Connexion** fournit des flux d'entrée et de sortie (donc des implémentations de **InputStream** et **OutputStream**) sur lesquels il n'y a plus qu'à lire ou écrire les bytes constituant les données.

18.2 La méthode open de Connector

La classe **Connector** (du package javax.microedition.io) ne contient en fait que des versions polymorphes de la méthode statique **open()**, qui est donc une factory destinée à produire l'objet **Connection** le plus approprié :

```
public static Connection open (String url) throws IOException
```

accompagnées de méthodes statiques réalisant le même travail mais fournissant immédiatement un flux d'entrée ou de sortie, bas niveau ou haut niveau, comme par exemple :

```
public static DataInputStream openDataInputStream (String url) throws IOException  
public static InputStream openInputStream (String url) throws IOException
```

...

Le paramètre d'**open()** répond à une syntaxe assez précise :

```
<protocole>://[<user>:<password>@]<adresse ou nom machine>:<port>/<chemin>[;<paramètres>]
```

où le protocole peut être, par exemple, http ou socket. Si on souhaite par exemple accéder à un fichier texte par l'intermédiaire d'un serveur Web, on écrira quelque chose du genre:

```
conn = (HttpConnection) Connector.open("http://localhost:8083/newsOfDay.txt");
```

On peut mentionner, parmi les diverses possibilités, les protocoles suivant :

nom url	protocole correspondant	classe(s) Connection
http	HTTP	HttpConnection
https	HTTP sur SSL	HttpsConnection
file	Système de fichiers local	FileConnection, InputConnection
socket / serversocket	TCP	SocketConnection / ServerSocketConnection
datagram	UDP	UDPDatagramConnection
apdu / jcrmi	APDU	APDUConnection / JavaCardRMIConnection
btl2cap	Bluetooth	L2CAPConnection
comm	ligne série	CommConnection

En fait, seule l'implémentation des deux protocoles HTTP et HTTPS est assurée dans le MIDP de base. Les autres sont présents selon les possibilités du mobile utilisé.

18.3 Les interfaces Connection de base

L'interface **Connection** ne déclare en fait que la méthode

```
public void close() throws IOException
```

C'est compréhensible, puisque la méthode **open()** de **Connector** est en fait la factory qui construira l'objet implémentant **Connection** et correspondant au protocole et au mobile considéré.

Bien sûr, un tel objet Connection ne présente guère d'intérêt tel quel : il doit être capable de fournir des flux d'entrée et de sortie. On trouve donc des interfaces dérivés de Connection qui apportent cette potentialité :

- ◆ **InputConnection** avec

```
public InputStream openInputStream() throws IOException  
public DataInputStream openDataInputStream() throws IOException
```

- ◆ **OutputConnection** avec

```
public OutputStream openOutputStream() throws IOException  
public DataOutputStream openDataOutputStream() throws IOException
```

- ◆ **StreamConnection** qui se contente de rassembler les deux précédents en en dérivant – c'est la "pointe du diamant".

18.4 Les connexions HTTP

Si nous pensons plus précisément à des communications réseaux basées HTTP ou HTTPS, nous entrons dans le domaine des protocoles qui négocient les contenus (c'est l'essence même du protocole HTTP). Nous allons donc trouver deux interfaces définissant des connexions de ce type : **HttpConnection** et son descendant **HttpsConnection**. Cependant, toujours prudents, les concepteurs de J2ME ont glissé entre les connexions basiques et les connexions http-like un interface écran, **ContentConnection**, qui déclare les 3 méthodes:

```
public String getType()  
    référence à peine voilée au header content-type de HTTP;  
public String getEncoding()  
    pour le header content-encoding;  
public long getLength()  
    pour le header content-length.
```

18.5 Les connexions TCP

L'interface associé est **SocketConnection**, dérivé de StreamConnection : il apporte des méthodes prévisibles, bien logiques pour un "pipe TCP" :

```
public String getLocalAddress() throws IOException  
public int getLocalPort() throws IOException  
public String getAddress() throws IOException  
public int getPort() throws IOException
```

et aussi une méthode de paramétrage de socket :

```
public void setSocketOption(byte option, int value) throws IllegalArgumentException,  
IOException
```

dont le premier paramètre peut valoir l'une de constants de classe déclarée dans l'interface :

```
public static final byte DELAY  
public static final byte LINGER
```

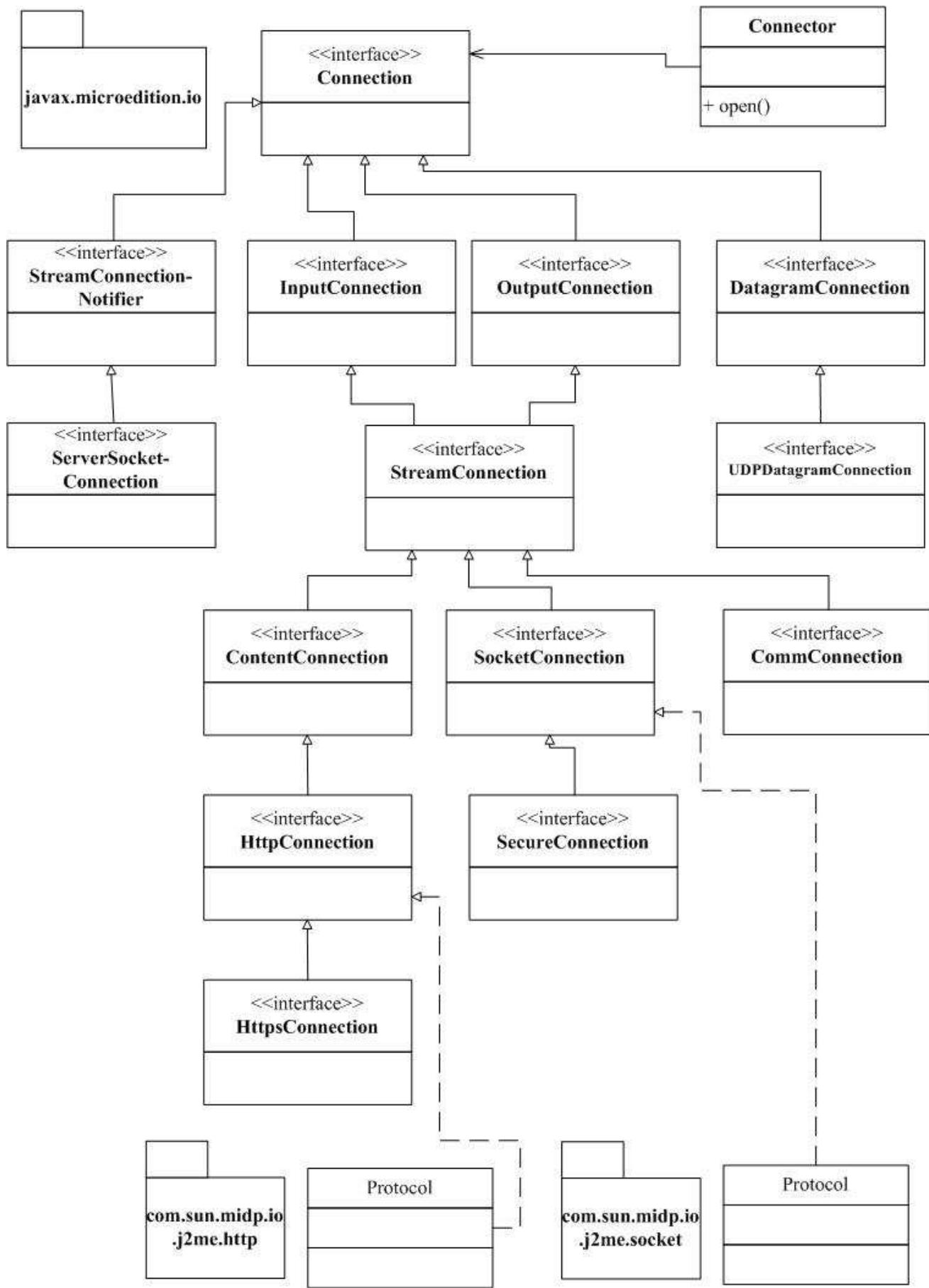
```
public static final byte KEEPALIVE
public static final byte RCVBUF
public static final byte SNDBUF
```

- le deuxième paramètre en fixe évidemment la valeur.

On peut encore citer l'interface dérivé de SocketConnection nommé **SecureConnection**, qui est la version sécurisée du précédent, basé sur SSL.

18.6 Le diagramme de classe UML des interfaces et classes réseaux

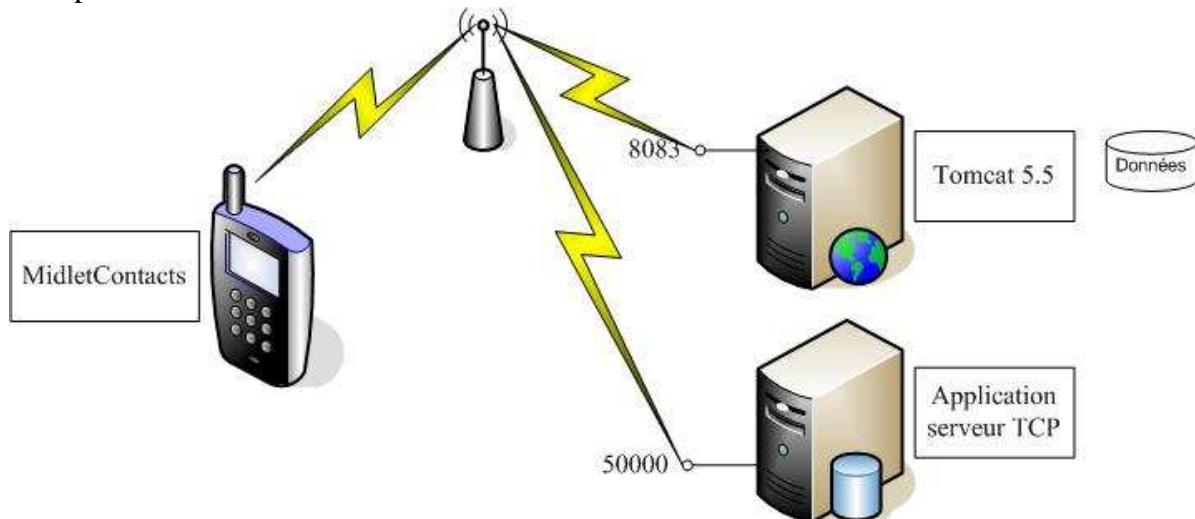
Le diagramme suivant propose la synthèse des possibilités :



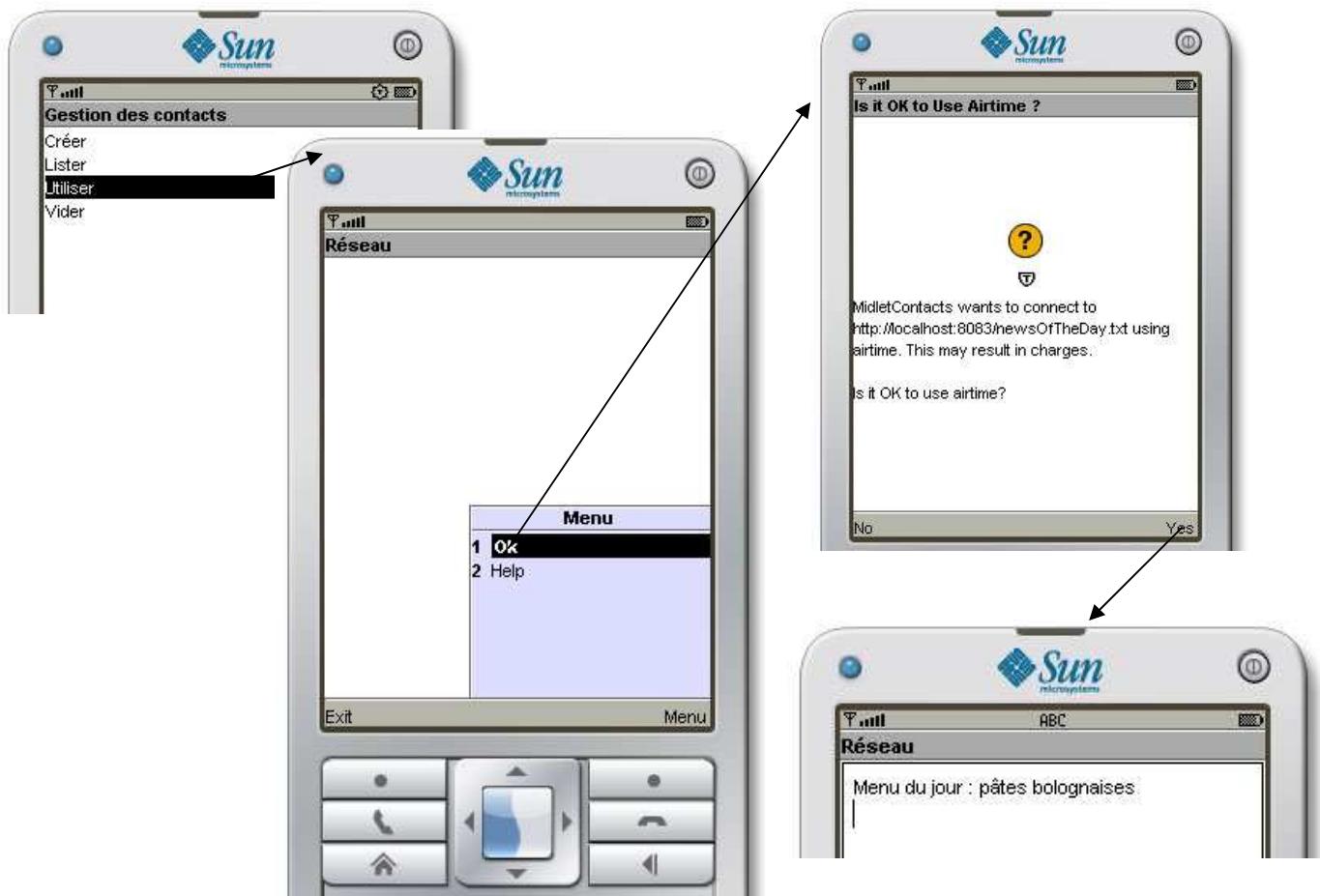
19. Une application réseau J2ME-J2SE basée TCP et HTTP

19.1 Description de l'application

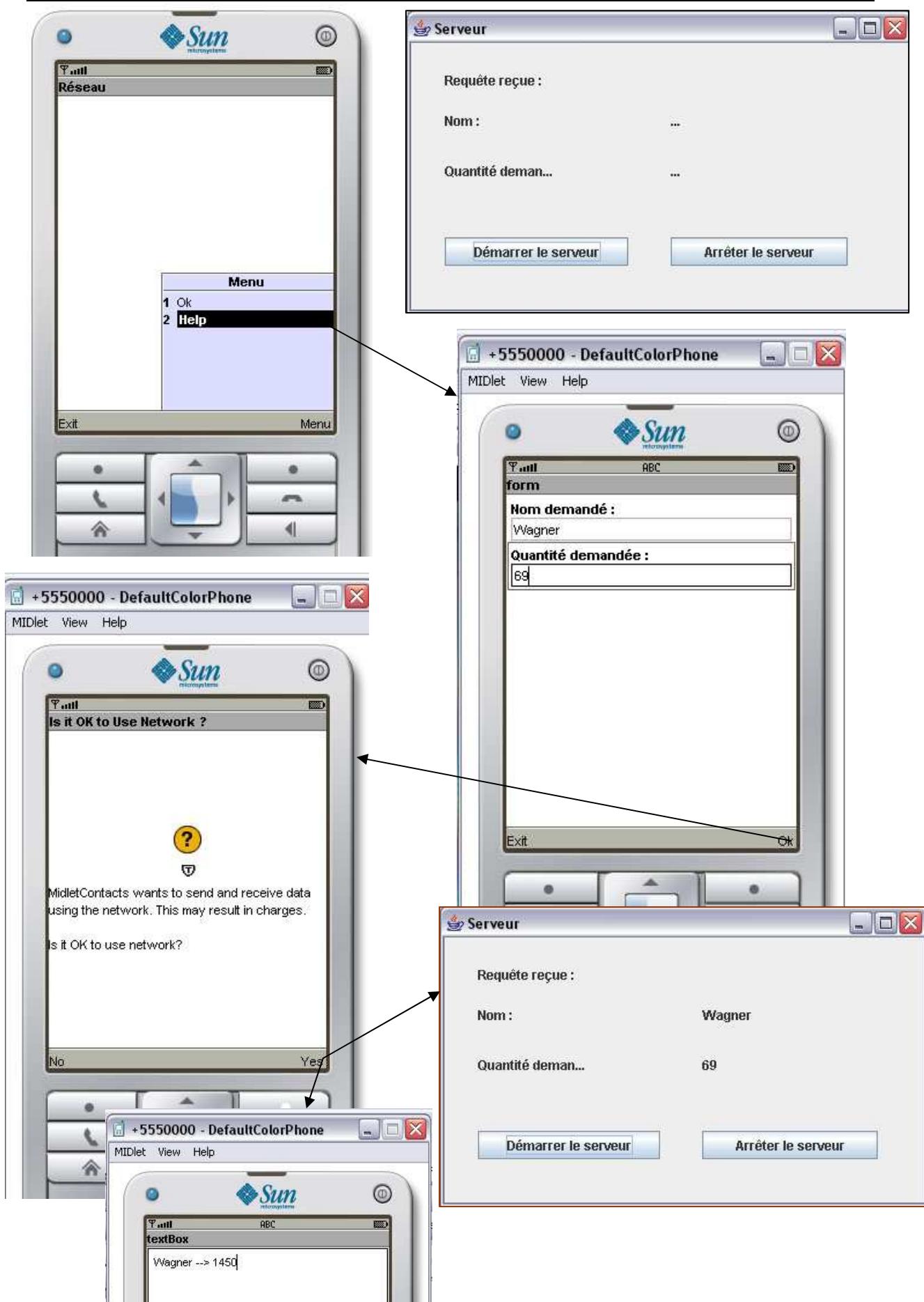
Afin d'illustrer ces concepts, nous allons imaginer une petite application réseau que l'on pourrait schématiser ainsi :



L'effet attendu est celui-ci (présentation sur le simulateur) : dans la liste proposée par notre MIDlet MidletContacts, le choix d'Utiliser fait apparaître un menu où Ok donne :

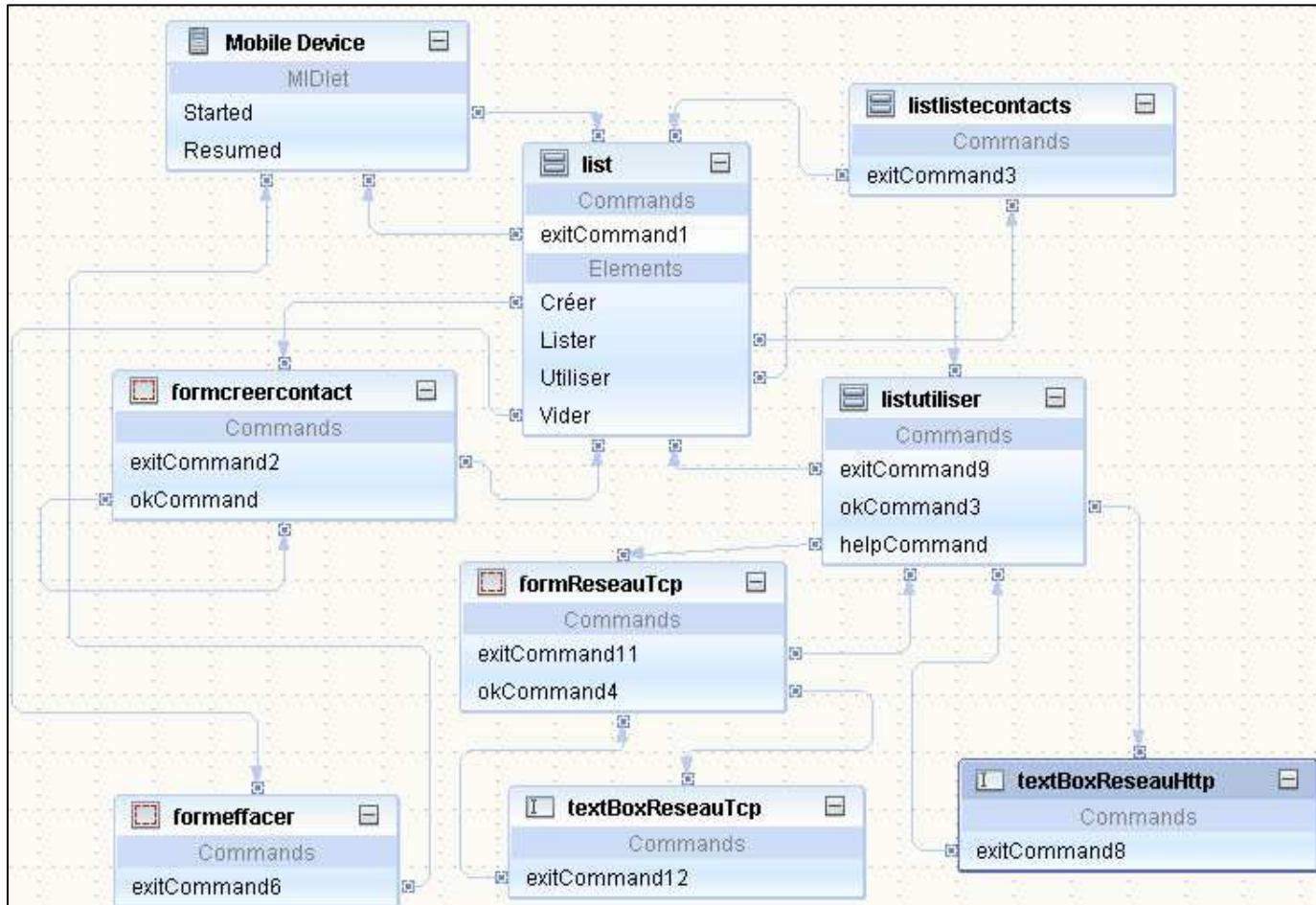


- le menu du jour est en fait un fichier texte qui se trouve sur le serveur Web; le choix de Help permet d'interagir avec un serveur FenServeurSocket (son air devrait dire quelque chose au lecteur qui a lu Java II ;-)) :



19.2 Analyse de l'application

Le schéma conceptuel sera celui-ci :



On remarquera le choix sur l'item de liste listutiliser :

- ◆ la commande Ok conduira à une boîte de texte pour la réponse à la requête HTPP;
- ◆ la commande Help conduira au formulaire formReseauTcp qui permettra l'encodage des données de la requête (nom et quantité demandée), une commande Ok conduisant à son tour à une boîte de texte pour la réponse.

Bien sûr, nous créerions deux threads, chacun chargé d'une mission réseau : ils recevront la référence d'une TextBox où écrire leurs résultats ainsi que l'URL du serveur à atteindre ...
L'aide des packages nous le conseille d'ailleurs :

"Warning: To avoid potential deadlock, operations that may block, such as networking, should be performed in a different thread than the commandAction() handler"

Le code de ces deux threads n'est pas très compliqué

19.3 Le thread pour HTTP

ThreadReseau.java

```
package contacts;

import java.io.IOException;
import java.io.InputStream;
import javax.microedition.io.Connector;
import javax.microedition.io.HttpConnection;
import javax.microedition.lcdui.TextBox;

/**
* @author Vilvens
* /

public class ThreadReseau extends Thread
{
    private TextBox afficheurThread;
    private String adresseReseau;

    public ThreadReseau(TextBox d, String url)
    {
        super();
        afficheurThread = d;
        adresseReseau = url;
    }
    public void run()
    {
        HttpConnection conn = null;
        InputStream is = null;
        byte[] chaineLue = null;
        String urlServeur = adresseReseau;
        try
        {
            conn = (HttpConnection) Connector.open(urlServeur);
            System.out.println("Classe de connexion : " + conn.getClass().getName());
            conn.setRequestMethod(HttpConnection.GET);
            if (conn.getResponseCode() == HttpConnection.HTTP_OK)
            {
                is = conn.openInputStream();
                int longueurData = (int) conn.getLength();
                if (longueurData > 0)
                {
                    chaineLue = new byte[longueurData];
                    is.read(chaineLue);
                }
            }
            if (is != null) is.close();
            if (conn != null) conn.close();
        }
```

```

        }
        catch (IOException ex)
        {
            System.out.println("Erreur : " + ex.getMessage());
        }
        String chaineAsciiLue = new String(chaineLue);
        System.out.println("Chaine reçue = " + chaineAsciiLue);
        afficheurThread.setString(chaineAsciiLue);
    }
}

```

19.4 Le thread pour TCP

ThreadReseauTcp.java

```

package contacts;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import javax.microedition.io.Connector;
import javax.microedition.io.SocketConnection;
import javax.microedition.lcdui.StringItem;
import javax.microedition.lcdui.TextBox;

/**
 * @author Vilvens
 */
public class ThreadReseauTcp extends Thread
{
    private TextBox afficheurThread;
    private String adresseReseau;
    private String outNom;
    private int outQuantite;

    public ThreadReseauTcp(String n, int q, TextBox d, String url)
    {
        super();
        outNom = n;
        outQuantite = q;
        afficheurThread = d;
        adresseReseau = url;
    }
    public void run()
    {
        SocketConnection conn = null;
        DataInputStream dis = null;
        DataOutputStream dos = null;
        byte[] chaineLue = null;
        String urlServeur = adresseReseau;

```

```

String reponse = null;
int inQuantiteRestante = 0;

try
{
    conn = (SocketConnection) Connector.open(urlServeur);
    System.out.println("Classe de connexion : " + conn.getClass().getName());
    System.out.println(conn.getAddress() + " / " + conn.getLocalAddress());
    System.out.println(conn.getPort() + " / " + conn.getLocalPort());
    if (conn!=null)
    {
        dis = conn.openDataInputStream();
        dos = conn.openDataOutputStream();

        dos.writeUTF(outNom);
        dos.writeInt(outQuantite);
        reponse = dis.readUTF();
        inQuantiteRestante = dis.readInt();
        System.out.println("Réponse obtenue pour " + outNom + " = "
                           + reponse + " --> " + inQuantiteRestante);
        String reponseComplete = reponse + " --> " + inQuantiteRestante;
        afficheurThread.setString(reponseComplete);
    }
    if (dis!=null) dis.close();
    if (conn!=null) conn.close();
}
catch (IOException ex)
{
    System.out.println("Erreur : " + ex.getMessage());
}
}
}

```

19.5 Le code réseau de la MIDlet

A cause du travail du générateur de Netbeans, il est un peu compliqué de retrouver les appels de nos threads. En fait :

MIDletContacts.java

```

package contacts;

import java.io.IOException;
import java.io.InputStream;
import javax.microedition.io.Connector;
import javax.microedition.io.HttpConnection;
import javax.microedition.MIDlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.rms.*;

```

```
/***
 * @author Vilvens
 */

public class MIDletContacts extends MIDlet implements CommandListener
{
...
    public void commandAction(Command command, Displayable displayable)
    {
        // write pre-action user code here
        if (displayable == formReseauTcp)
        {
            if (command == exitCommand11)
            {
                // write pre-action user code here
                switchDisplayable(null, getListutiliser());
                // write post-action user code here
            }
            else if (command == okCommand4)
            {
                // write pre-action user code here
                switchDisplayable(null, getTextBoxReseauTcp());
                // write post-action user code here
            }
        }
...
        else if (displayable == listutiliser)
        {
            if (command == List.SELECT_COMMAND)
            {
                // write pre-action user code here
                listutiliserAction();
                // write post-action user code here
            }
            else if (command == exitCommand9)
            {
                // write pre-action user code here
                switchDisplayable(null, getList());
                // write post-action user code here
            }
            else if (command == helpCommand)
            {
                // write pre-action user code here
                switchDisplayable(null, getFormReseauTcp());
                // write post-action user code here
            }
            else if (command == okCommand3)
            {
                // write pre-action user code here
                switchDisplayable(null, getTextBoxReseauHttp());
            }
        }
    }
}
```

```

        // write post-action user code here
    }
}
else if (displayable == textBoxReseauHttp)
{
    if (command == exitCommand8)
    {
        // write pre-action user code here
        switchDisplayable(null, getListutiliser());
        // write post-action user code here
    }
}
else if (displayable == textBoxReseauTcp)
{
    if (command == exitCommand12)
    {
        // write pre-action user code here
        switchDisplayable(null, getFormReseauTcp());
        // write post-action user code here
    }
}
// write post-action user code here
}

public TextBox getTextBoxReseauHttp()
{
    if (textBoxReseauHttp == null)
    {
        // write pre-init user code here
        textBoxReseauHttp = new TextBox("R\u00e9seau", null, 100, TextField.ANY);
        textBoxReseauHttp.addCommand(getExitCommand8());
        textBoxReseauHttp.setCommandListener(this);

        ThreadReseau tr = new ThreadReseau(textBoxReseauHttp,
                                         "http://192.168.1.8:8083/newsOfDay.txt");
        tr.start();
        //textBox.setString(new String(lireSurReseau()));
        // write post-init user code here
    }
    return textBoxReseauHttp;
}

public TextBox getTextBoxReseauTcp()
{
    if (textBoxReseauTcp == null)
    {
        // write pre-init user code here
        textBoxReseauTcp = new TextBox("textBox", null, 100, TextField.ANY);
        textBoxReseauTcp.addCommand(getExitCommand12());
        textBoxReseauTcp.setCommandListener(this);
    }
}

```

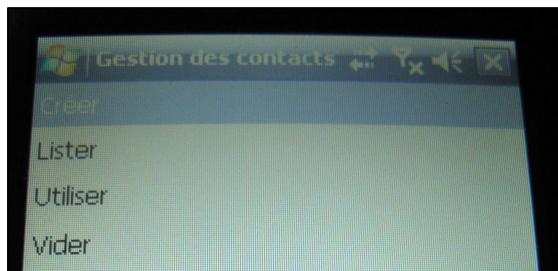
```
// write post-init user code here

ThreadReseauTcp tr = new ThreadReseauTcp(  
    ZTNomDemande.getString(),  
    Integer.parseInt(ZTQuantite.getString()),  
    textBoxReseauTcp,  
    "socket://192.168.1.8:50000");  
    tr.start();  
}  
return textBoxReseauTcp;  
}  
}
```

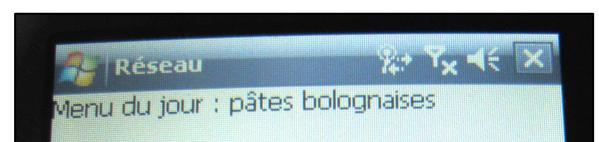
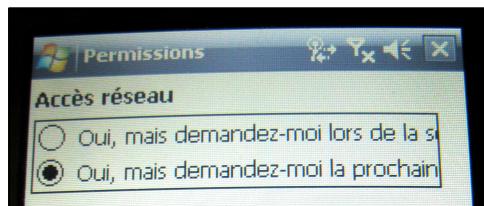
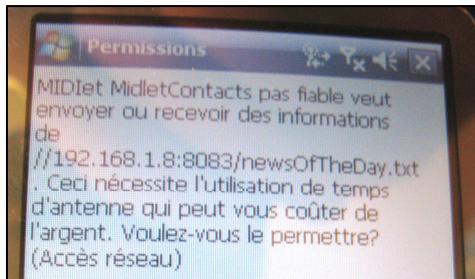
Ouf ! Lors de l'exécution dans le simulateur, on peut constater que la classe effective de la classe de connexion est **com.sun.midp.io.j2me.http.Protocol**.

19.6 L'exécution de la MIDlet sur le mobile

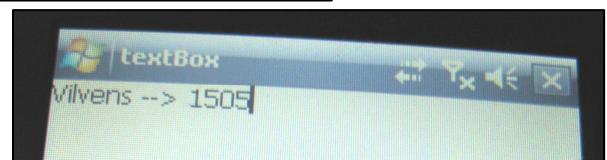
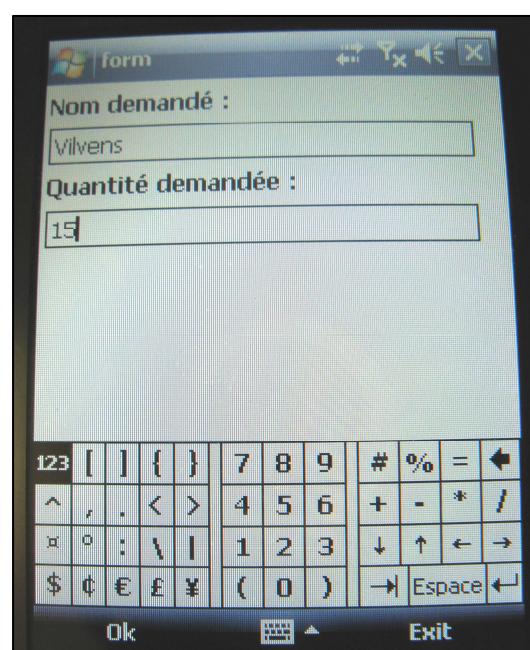
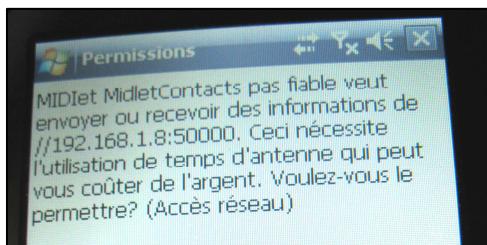
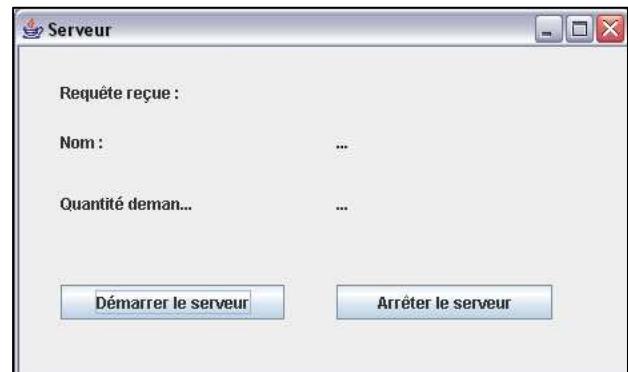
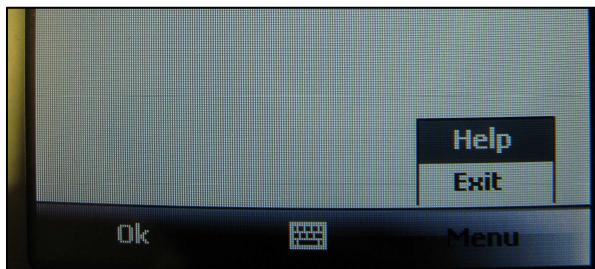
Après transfert de la MIDlet sur le mobile Htc, lancement de Tomcat et lancement du serveur TCP, on obtient enfin :



Ok

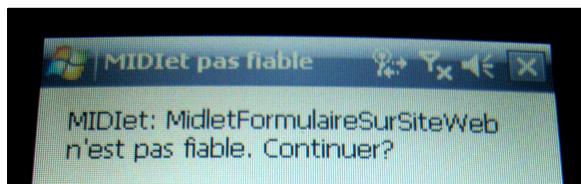


Menu



20. Une MIDlet authentifiée

Jusqu'à présent, nous avons superbement ignoré les avertissements apparaissant lorsque nous téléchargeons nos applications J2ME sur le mobile et nous signalant que notre code n'est pas fiable, c'est-à-dire authentifié par une autorité que nous reconnaissons :



20.1 Le mode opératoire

Nous allons donc à présent envisager de signer nos MIDlets. Pour cela, supposons disposer d'un keystore comme celui-ci (il s'agit du keystore keystore-simpleminds.p12 créé par programmation dans "Java (IV) : Programmation des protocoles applicatifs et sécurité logicielle", chapitre XXVI, paragraphe 3) :

Private Key (keypair) & Trusted Certificate Entries:

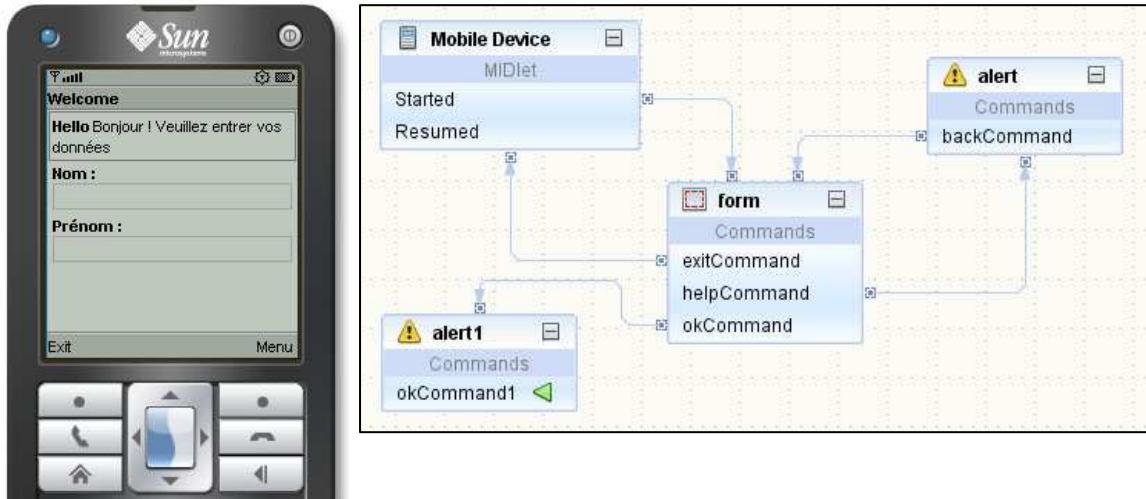
Alias	Entry	Valid Date ?	Self-Signed ?	Trusted C.A. ?	Key Size	Cert. Type	Cert. Sig. Algo.	Modified Date
Claude		✓	-	-	2048 bits	X.509	SHA1WithRSAE...	16-août-2009
HEPL		✓	✓	-	2048 bits	X.509	SHA1WithRSAE...	16-août-2009
HEPL-root		✓	✓	-	2048 bits	X.509	SHA1WithRSAE...	16-août-2009

qui contient des clés permettant de signer. Plus précisément, nous allons :

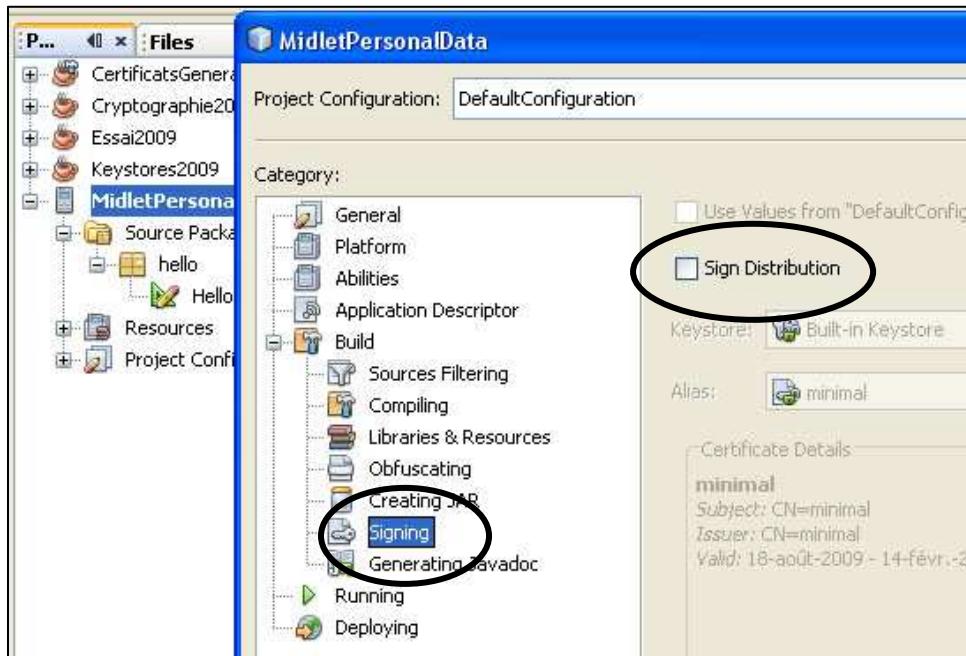
- ◆ construire une MIDlet basique et la signer au moyen de la clé privée de la Key Entry "HEPL-root";
- ◆ exporter le certificat contenant la clé publique associée, certificat se trouvant dans la Trusted Certificate Entry "HEPL";
- ◆ installer ce certificat sur le mobile HTC;
- ◆ installer notre MIDlet signée et constater qu'elle est considérée comme fiable.

20.2 La signature de la MIDlet

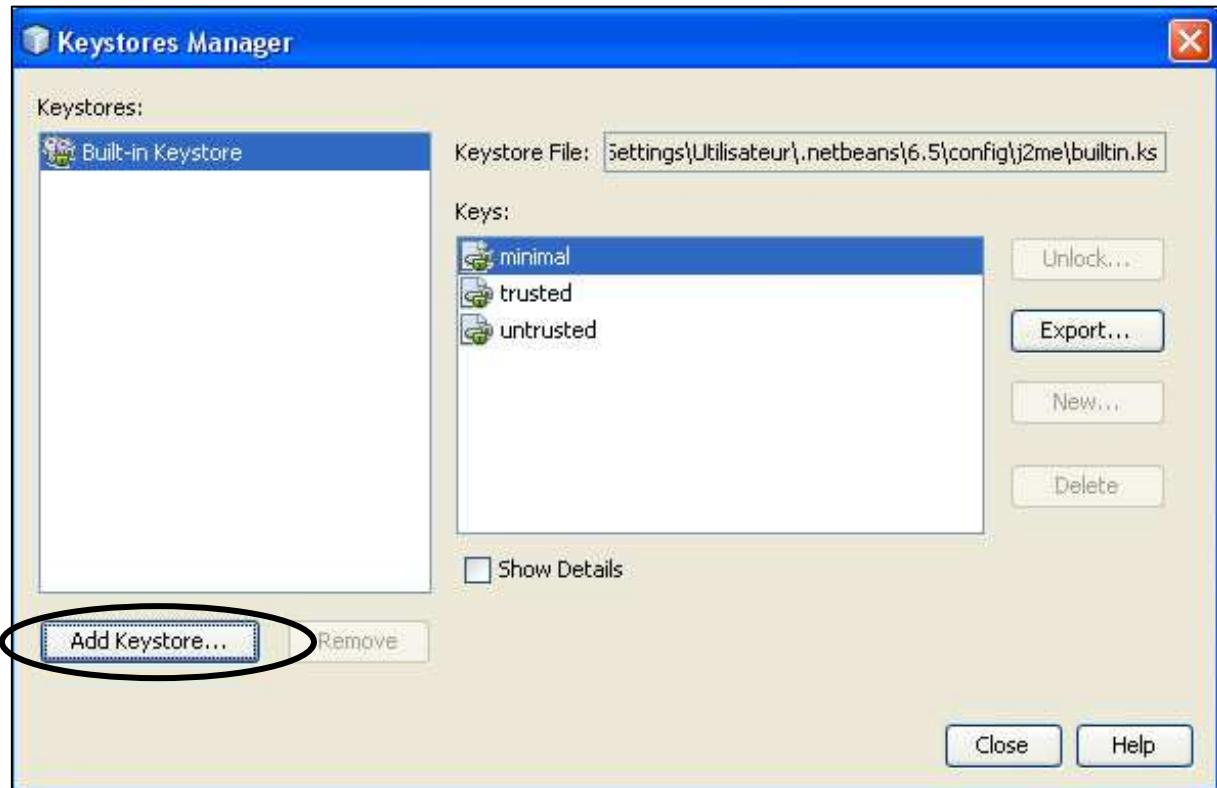
Considérons une application J2ME très simple comme celle-ci, développée sous Netbeans 6.5.1 :



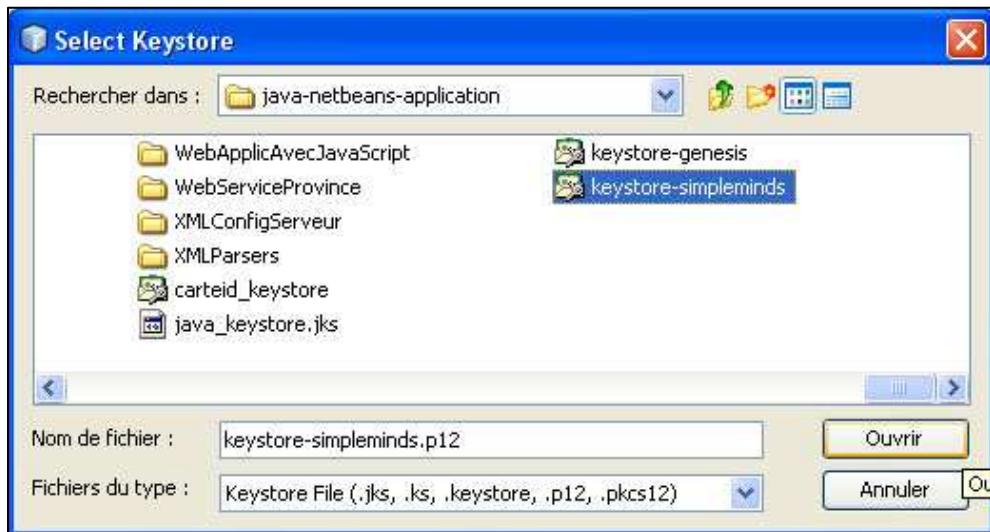
Le processus de signature est pris en charge par Netbeans 6.5.* , si on le demande dans les propriétés du projet :



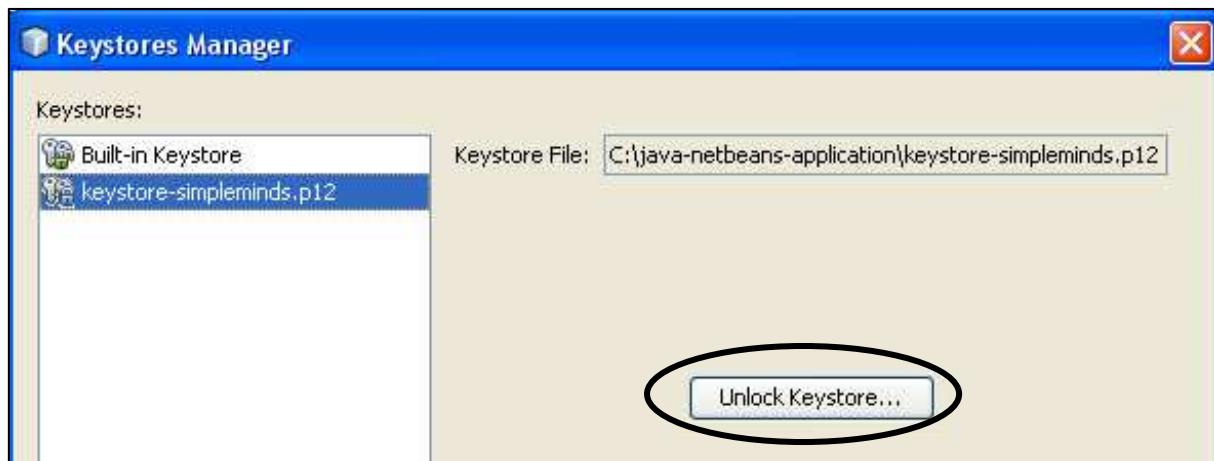
Le fait de cocher la case "Sign Distribution" a pour effet d'activer un bouton "Open Keystores Manager ..." qui va bien entendu nous permettre, une fois sollicité, de sélectionner le keystore et la Key Entry à utiliser. On obtient :



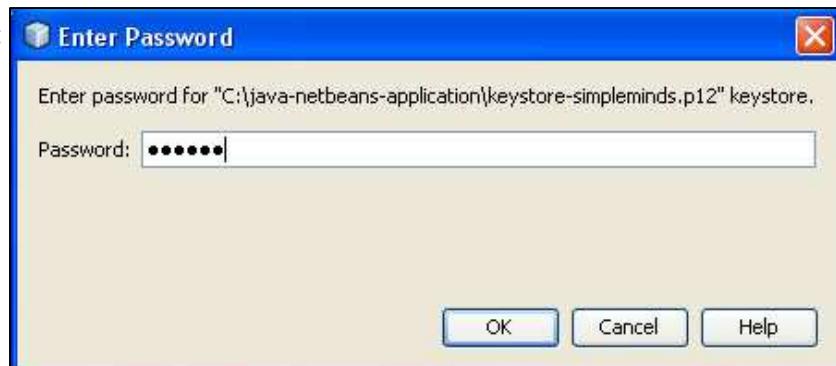
L'appui sur le bouton "Add Keystore ..." permet de choisir le keystore en question :



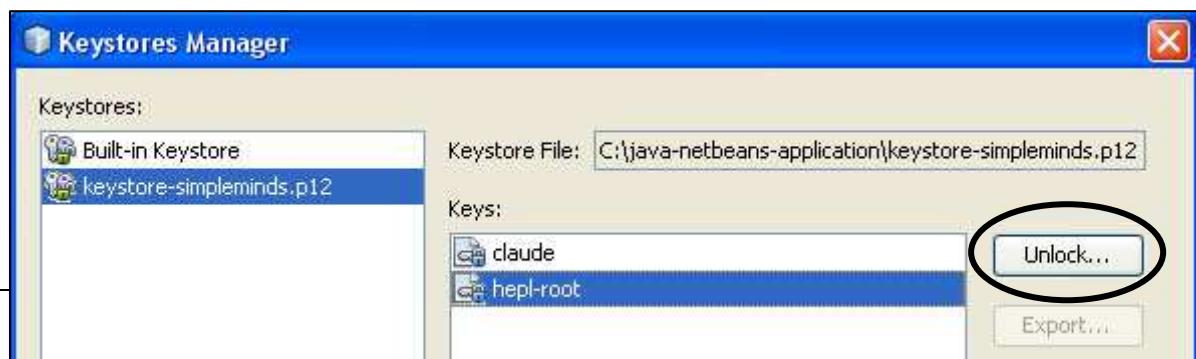
Evidemment, le keystore en question est protégé par un mot de passe : l'appui sur le bouton "Unlock keystore"



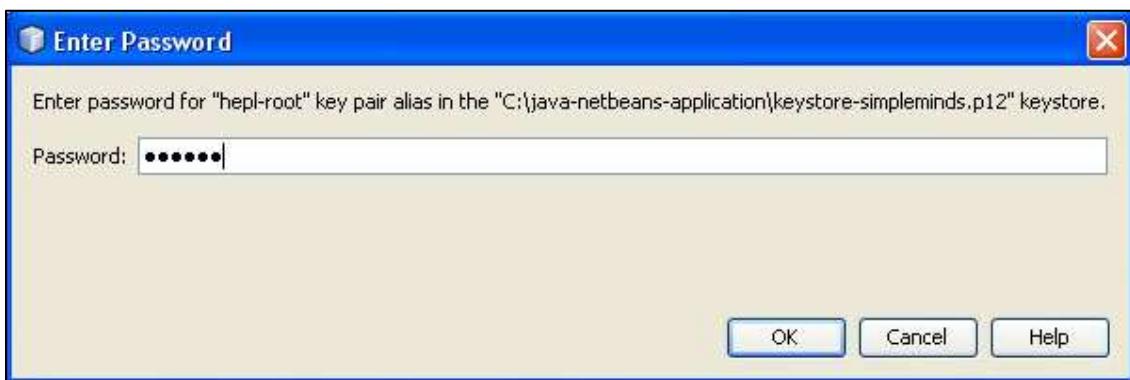
permet d'introduire celui-ci :



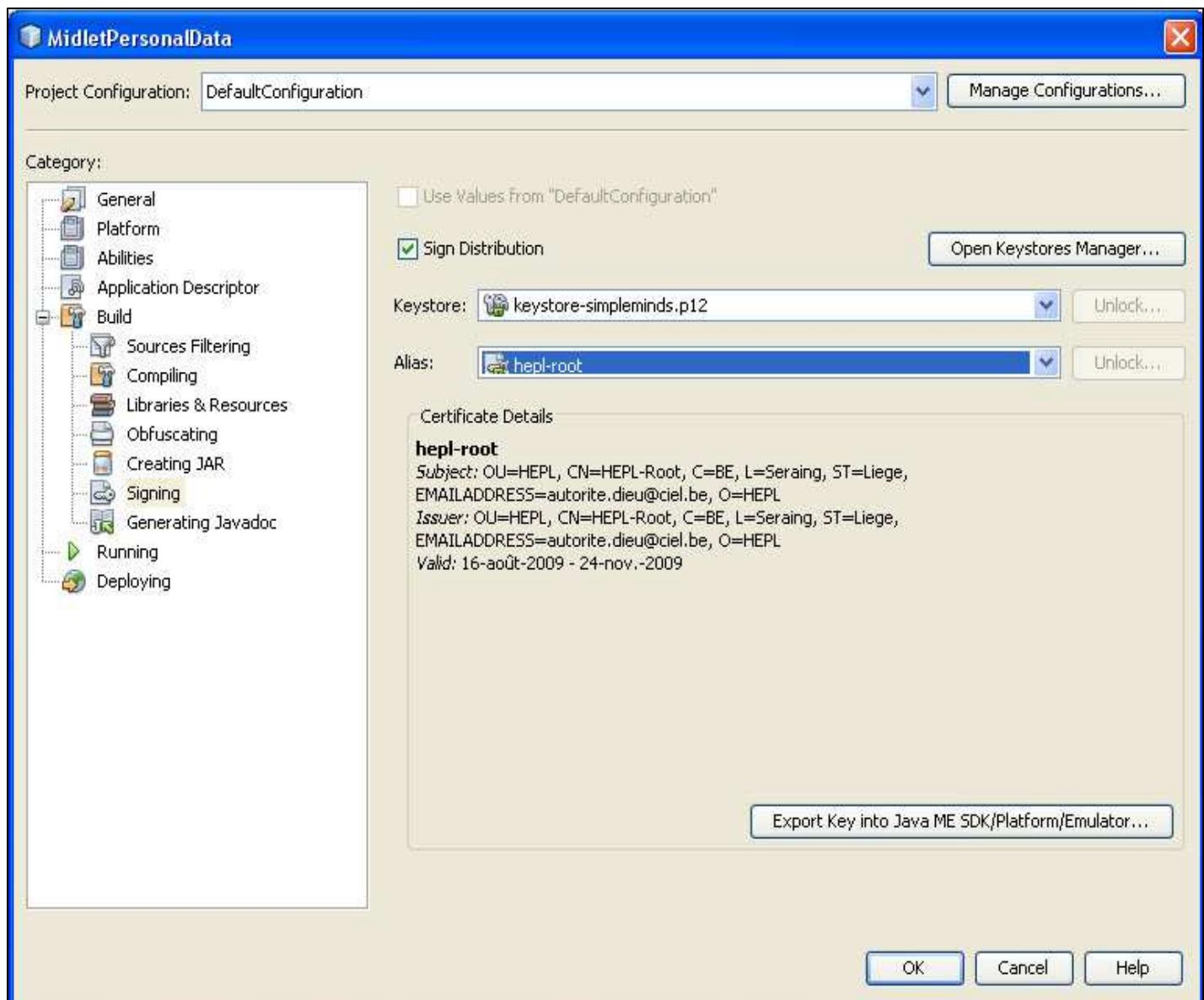
On obtient ainsi la liste des Key Entry disponibles :



mais un mot de passe sera à nouveau nécessaire :



Le projet est ainsi configuré :



Le lancement du "Build all" du projet va déclencher le processus de signature :

```
Building jar: C:\java-netbeans-application\MidletPersonalData\dist\MidletPersonalData.jar
...
update-jad:
Updating application descriptor: C:\java-netbeans-application\MidletPersonalData\dist\
MidletPersonalData.jad
Generated "C:\java-netbeans-application\MidletPersonalData\dist\MidletPersonalData.jar" is
3213 bytes.
Signing Jar
Adding MIDlet-Jar-RSA-SHA1 attribute.
ricoh-init-dalp:
...
override-jad:
semc-ppro-arm:
...
BUILD SUCCESSFUL (total time: 1 second)
```

Que s'est-il passé ? Contrairement à ce qui se passe pour une applet signée, où le fichier manifeste se fait escorter de deux autres fichiers DSA et SF, c'est ici le fichier jad qui va incorporer la signature et le certificat permettant la vérification (les deux sont codés en base64) :

MidletPersonalData.jad

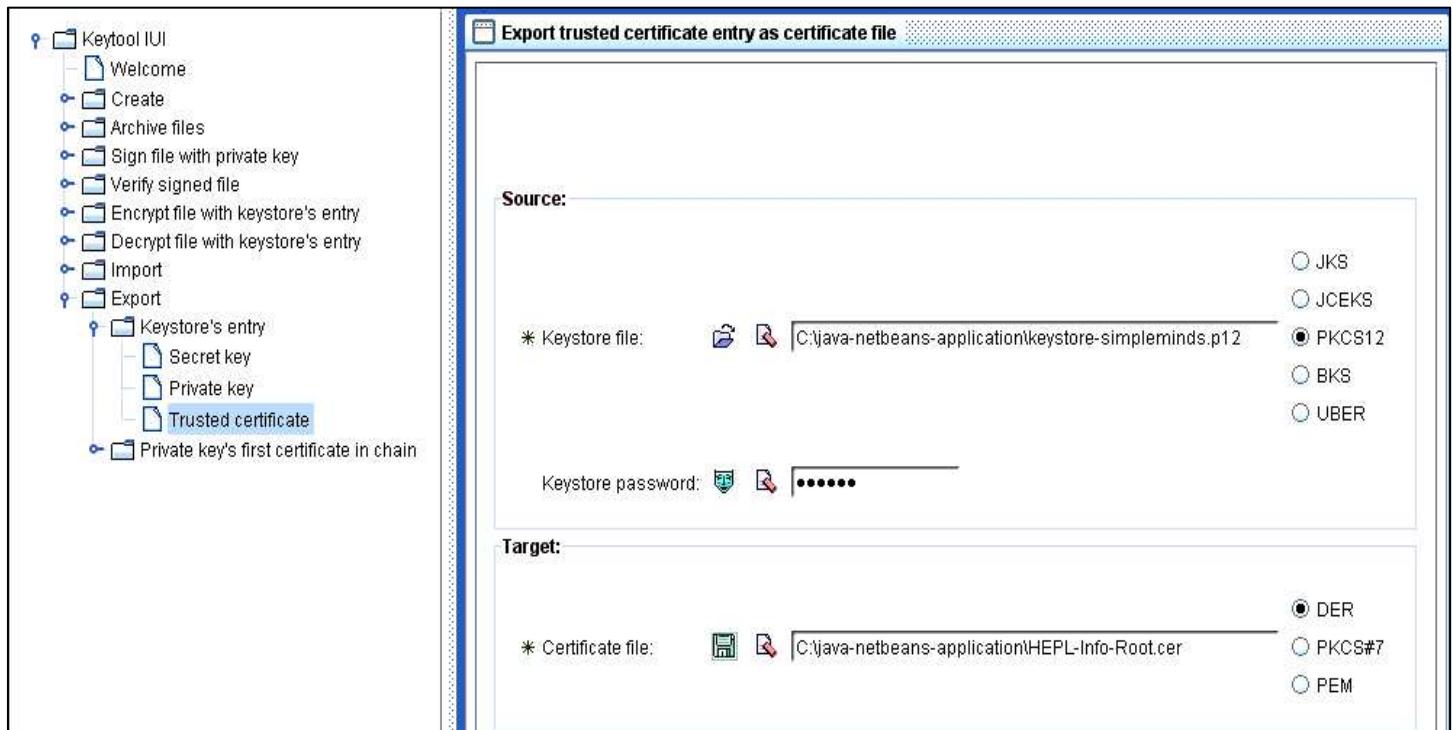
```
MIDlet-1: HelloMIDlet, , hello.HelloMIDlet
MIDlet-Certificate-1-1: MIIDjjCCAnagAwIB...3LYpXmSSg=
MIDlet-Jar-RSA-SHA1: CCpnX6v2MvZw...WxnemHUsNg==
MIDlet-Jar-Size: 3213
MIDlet-Jar-URL: MidletPersonalData.jar
MIDlet-Name: MidletPersonalData
MIDlet-Vendor: Vendor
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.1
MicroEdition-Profile: MIDP-2.0
```

Nous allons à présent pouvoir passer sur notre mobile ...

20.3 L'installation du certificat sur le mobile

Avant d'installer notre MIDlet signée sur l'appareil, il nous faut lui donner les moyens de l'authentifier, dont installer le certificat nécessaire dans son "magasin à certificats". Pour cela, 3 étapes :

- Nous exportons le certificat de HEPL-root dans un fichier HEPL-Info-Root.cer; ceci peut se faire facilement avec Keytool IUI (outil présenté en annexe) :



Une fois ce premier écran-formulaire rempli, on peut choisir l'entrée à exporter :



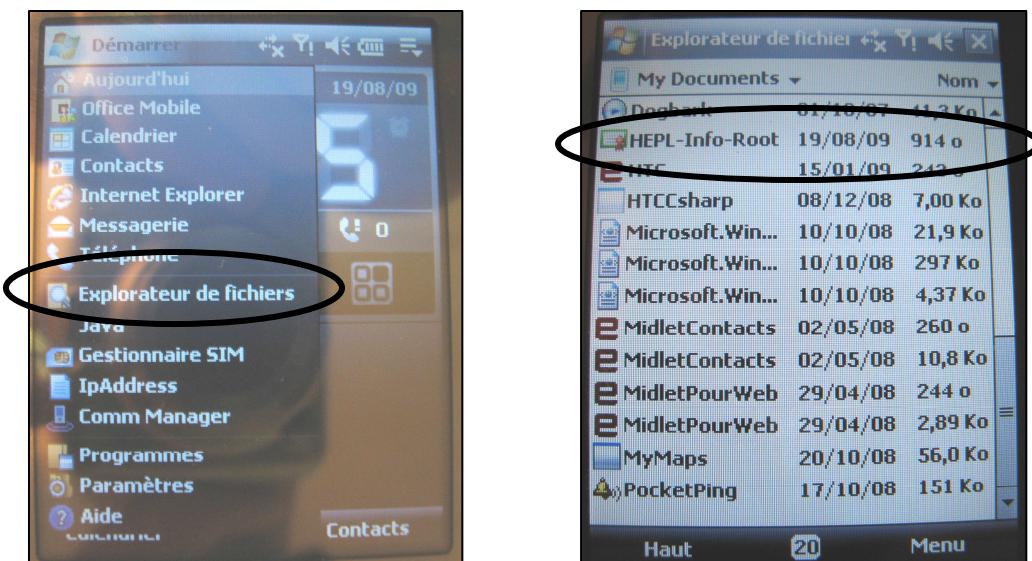
et on peut commander l'exportation :



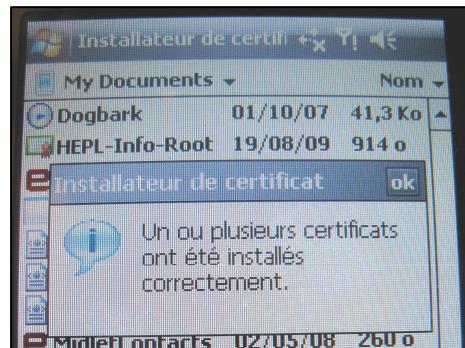
ainsi que la visualisation du résultat :



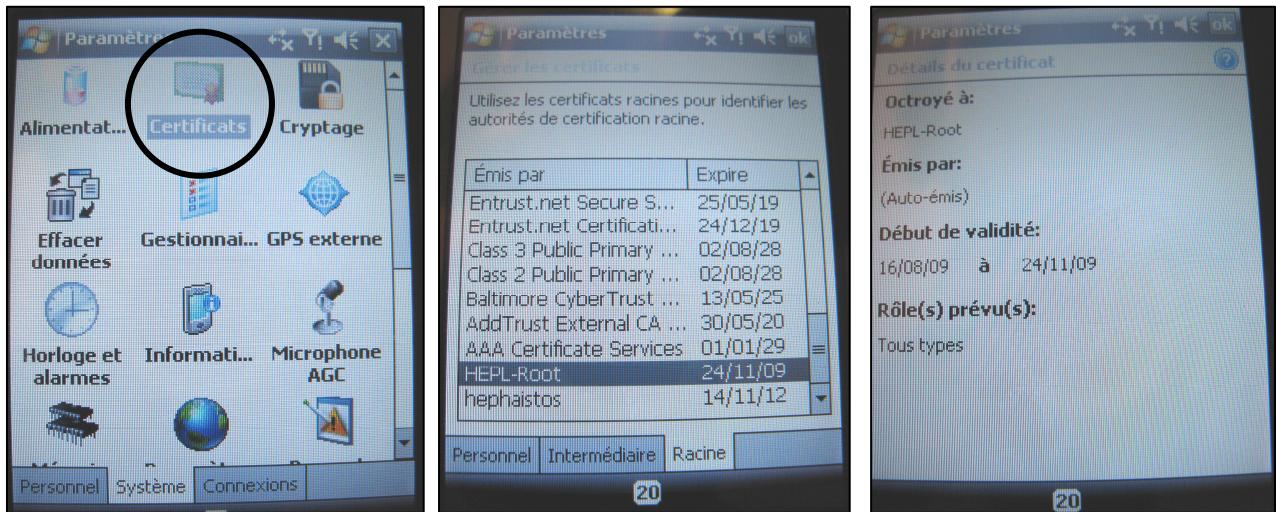
b) Nous installons le fichier résultat sur le mobile : après l'avoir copié dans le répertoire de synchronisation, nous synchronisons si bien que le fichier certificat est à présent physiquement sur le mobile. On peut le vérifier :



Mais il faut encore l'installer, ce qui se fait par un simple "double-clic" avec le stylet, pour obtenir :

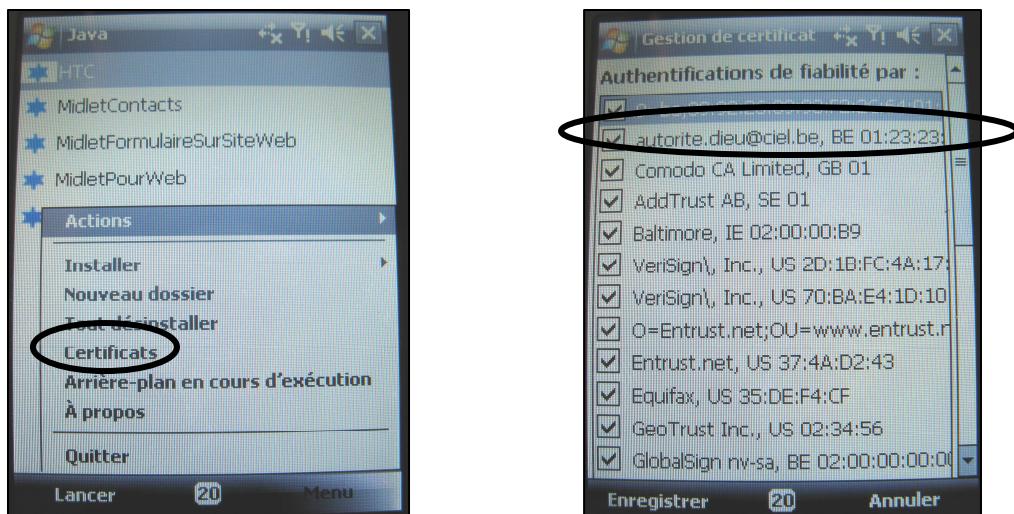


On peut consulter la liste des certificats installés en suivant depuis le menu principal de Windows Mobile : "Paramètres" → onglet "Système" → Certificats → onglet "Racine" :



- la sélection du certificat permettant de visualiser les informations intelligibles utiles.

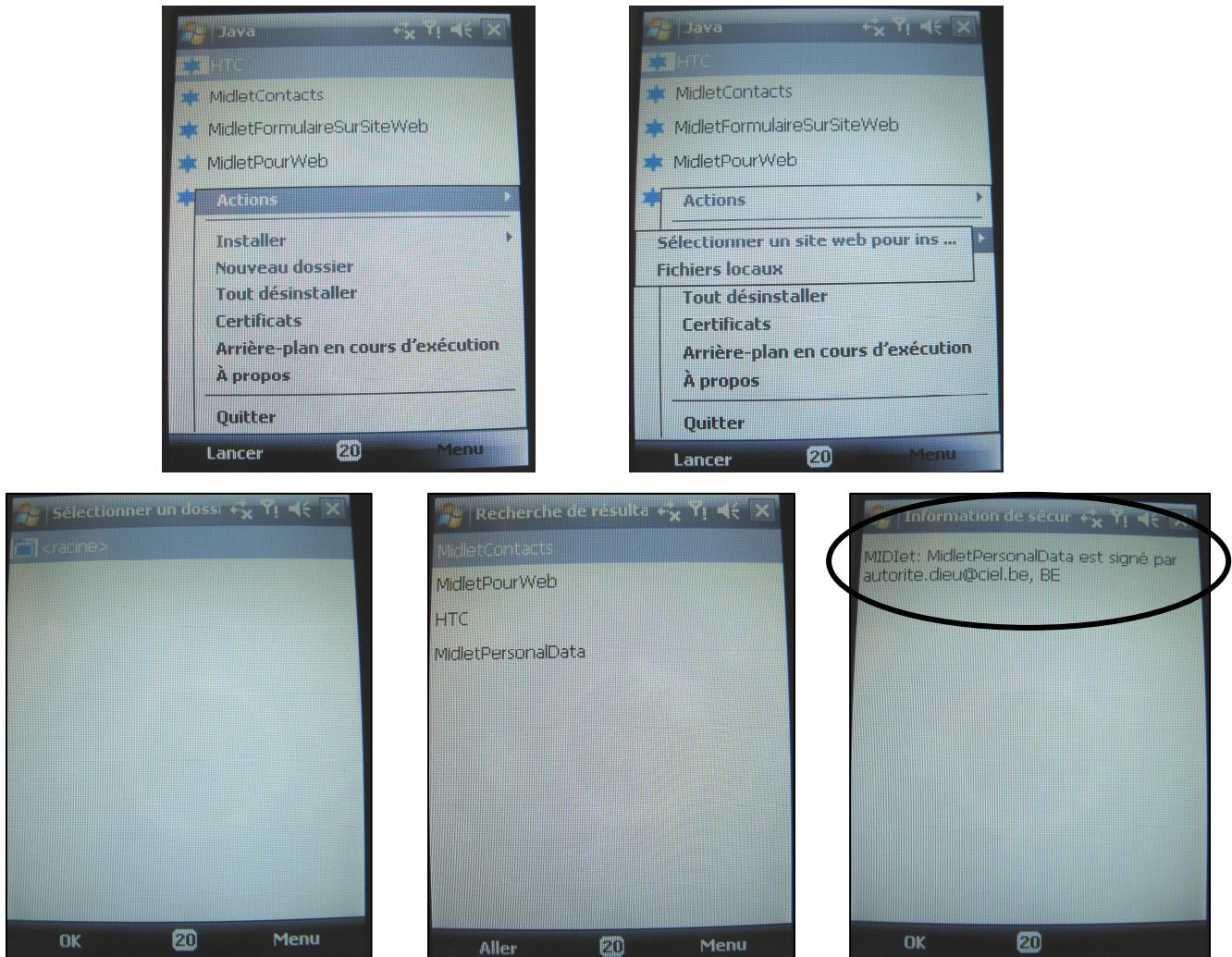
Il reste encore une vérification à réaliser : s'assurer que la machine virtuelle Java se sert de notre certificat. Pour le savoir, lançons Java depuis le menu principal de Windows Mobile : l'interface graphique obtenu permet de voir la liste des MIDlets installées mais aussi de réaliser divers paramétrages et installations par l'intermédiaire du menu :



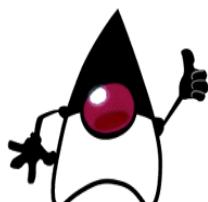
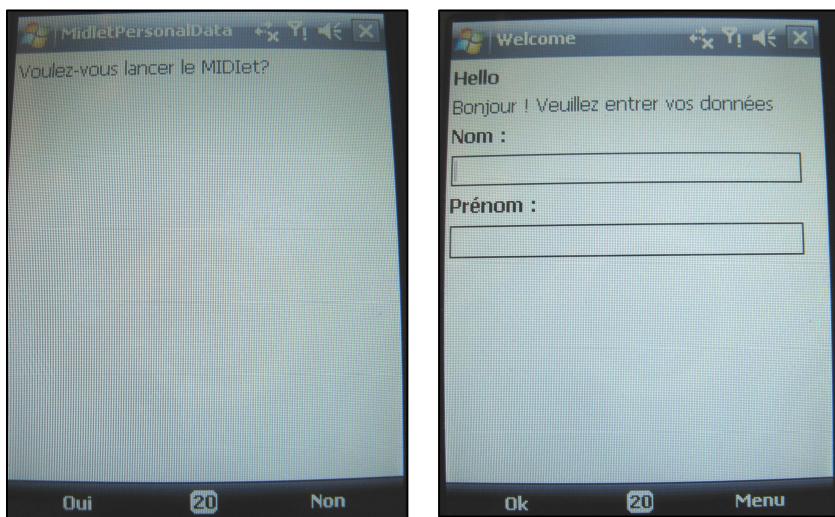
En particulier, le choix de "Certificats" permet de cocher les certificats qui seront utilisés pour les authentifications de code.

20.4 L'installation de la MIDlet sur le mobile

Nous pouvons enfin installer notre MIDlet. Après avoir transféré nos fichiers .jar et .jad avec l'outil de synchronisation, nous lançons l'interface Java et nous choisissons dans son menu l'item "Installer" :



Victoire ! Notre MIDlet a bien montré patte blanche et elle est exécutable :



Bien sûr, il y a encore à découvrir dans le monde des mobiles ! Ainsi, on pourrait par exemple s'intéresser aux mondes parallèles à celui des MIDlets : par exemple, découvrir le monde Google Android ...

XXVIII. Un autre Java embarqué : Google Android



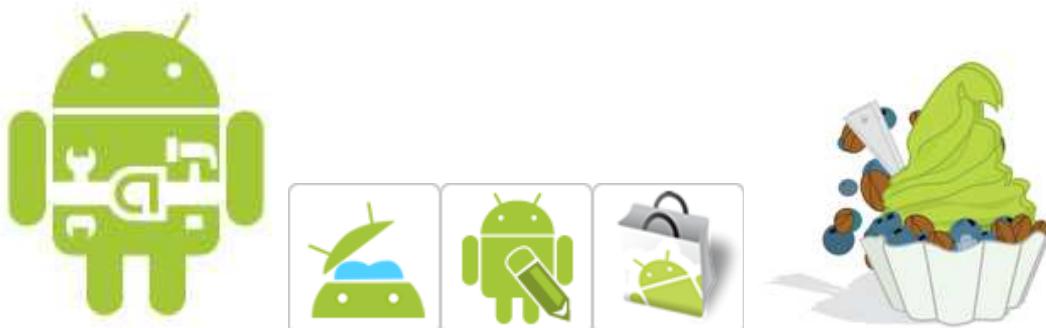
La pensée ne commence qu'avec le doute.

(R. Martin du Gard, Correspondance avec A.Gide)

Après J2ME qui vise à fournir un framework configurable et adaptable à des mobiles extrêmement divers, voici le point de vue diamétralement opposé : une technologie propriétaire, dédiée à une architecture bien précise avec comme langage de développement un Java non standard ... Une petite introduction à Android ?

1. Un framework pseudo-Java pour mobiles

Android est un framework combinant un système d'exploitation pour mobile et un SDK permettant le développement d'application pour tout mobile géré par cet OS.



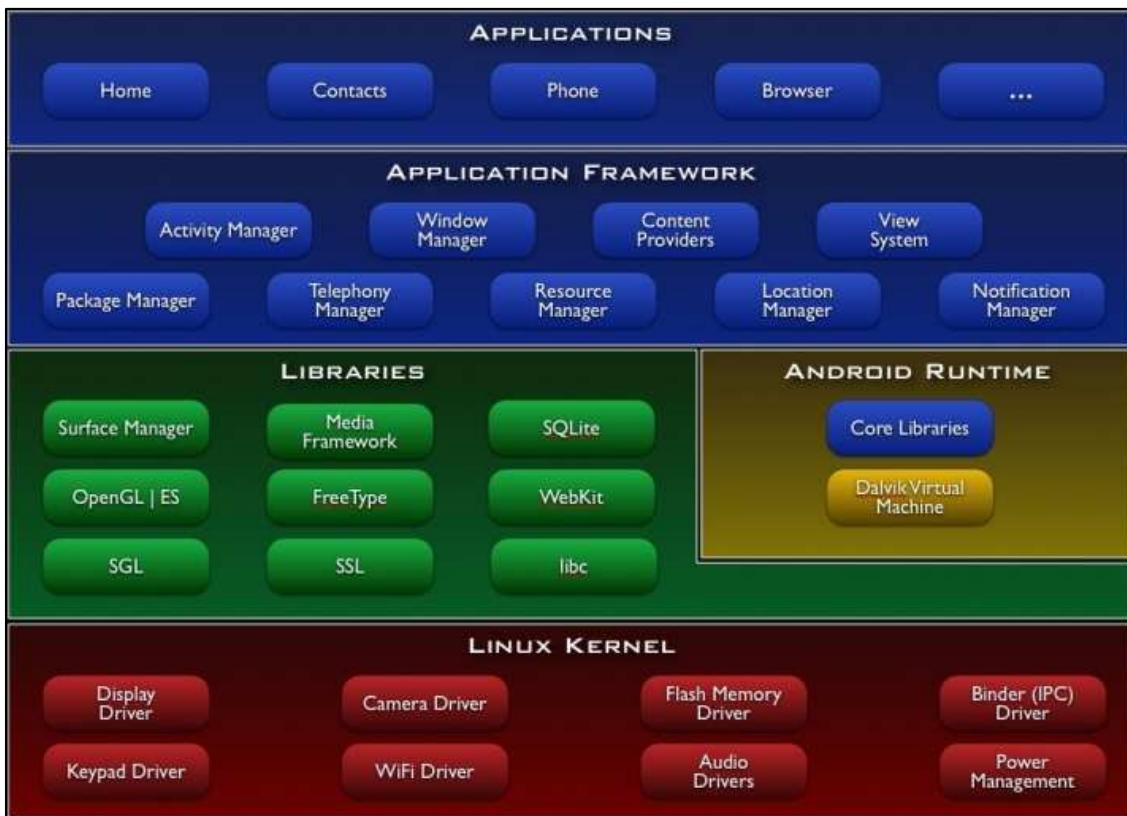
En fait, il utilise une version modifiée du kernel Linux - mais il ne s'agit pas ici d'une distribution de Linux puisqu'il manque, par exemple, le système X Window et le jeu complet des librairies standard GNU. Android a été développé dès 2007 par Android Inc. (bien logiquement) puis racheté par Google pour finalement passer sous le contrôle de la **OHA** (**Open Handset Alliance**), un consortium d'entreprises diverses intéressées par le marché des mobiles comme Bouygues, Alcatel, Dell, Acer, Samsung, HTC, Sony Ericsson, Toshiba, Intel, Vodafone, etc (voir http://www.openhandsetalliance.com/oha_members.html pour une liste complète).

La configuration matérielle prévue pour Android correspond à ce que l'on trouve sur un HTC Dream : essentiellement un écran tactile, Wifi et Bluetooth, un trackball, un clavier coulissant Qwerty et une carte SD de 1Gb. Du point de vue logiciel, **Android utilise un Java propriétaire**, non compatible avec J2ME et J2SE, et des librairies natives, construites à nouveau avec des **librairies C non standards** (la librairie Bionic).

2. Le modèle en couches d'Android

2.1 Le schéma bloc

La structure du framework est traditionnellement représentée sur les sites Web (<http://developer.android.com>) de la manière suivante :



Donnons immédiatement quelques précisions sur les différentes couches :

2.2 La couche Linux Kernel

On l'a dit, Android utilise *une version modifiée du kernel Linux* (un 2.6) qui prend en charge les drivers, les accès aux ressources, l'alimentation électrique. Cette couche représente donc l'interface d'accès au hardware pour les couches supérieures. Cependant, si certains éléments du kernel Linux traditionnel ont été enlevés, d'autres ont été ajoutés pour permettre son utilisation dans le monde particulier des mobiles : on parle de "*patches*". Citons :

- ◆ Alarm : pour le réveil de l'appareil quand il est en veille;
- ◆ Ashmem : gestion du partage mémoire, nécessaire sur un mobile où l'espace est réduit;
- ◆ Binder : il s'agit d'un module prenant en charge de manière sécurisée les IPC permettant une communication entre les différentes applications qui tournent dans des processus différents; l'idée est en fait de ne laisser les développeurs d'applications créer leurs propres IPC, ce qui ressentirait comme potentiellement dangereux.

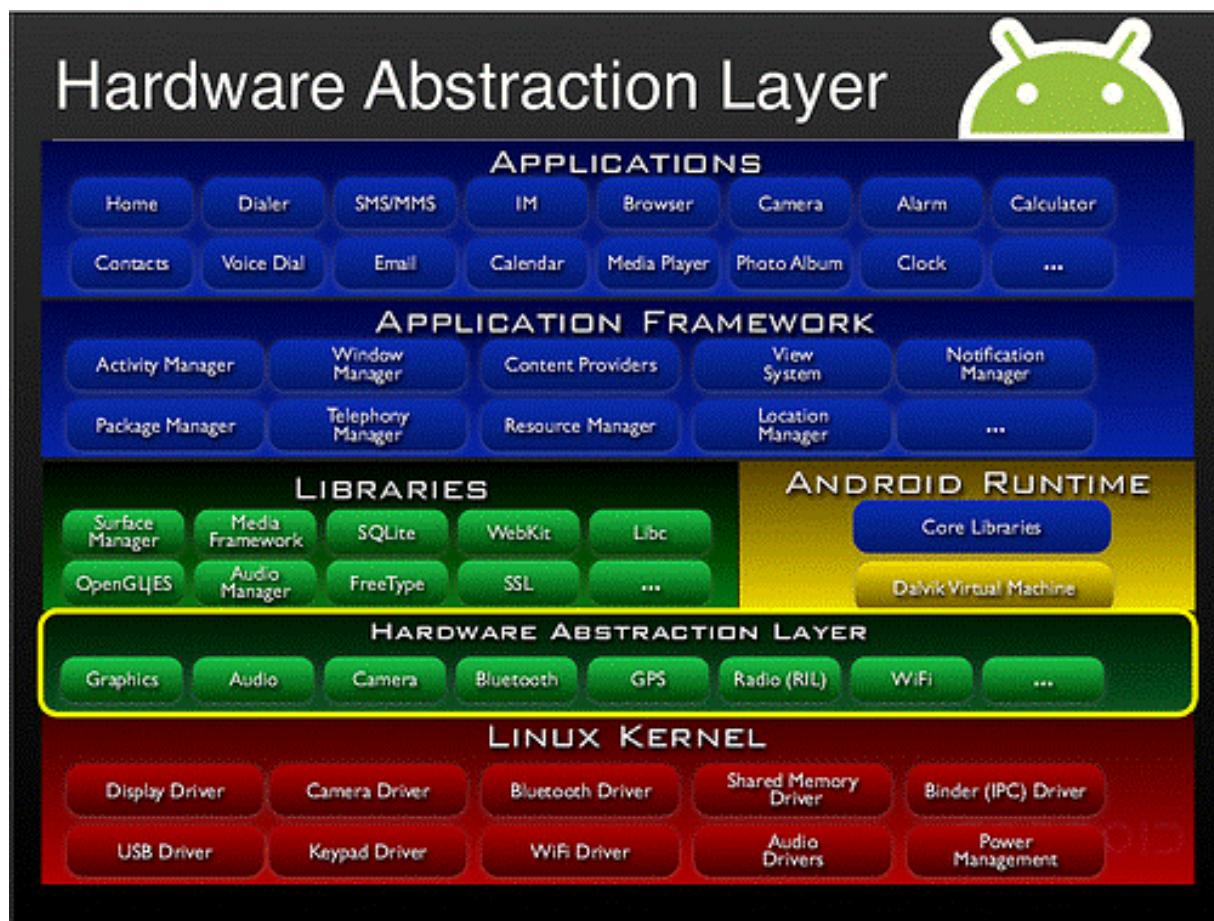
2.3 La couche librairies

Il s'agit d'un ensemble de librairies C/C++ qui seront utilisables depuis la couche supérieure (l'application framework). Citons :

- ◆ **Bionic LibC** : une espèce de *glibc revue à la baisse*, à nouveau pour tenir compte de l'environnement réduit d'un mobile;

- ◆ **Webkit** : une librairie de fonctions permettant aux logiciels d'afficher les éléments d'une page web - on parle encore de "*moteur de rendu*"; c'est un dérivé du moteur KHTML du projet KDE (ensemble de technologies dédiées à un environnement de bureau et de développement d'application dans un contexte multi-plateformes);
- ◆ **SQLLite** : un SGBD "léger" particulièrement adapté au monde des mobiles;
- ◆ Surface Flinger : pour les rendus graphiques en 2d et 3D.

Un cas particulier est celui des "**Hardware Abstraction Libraries**" puisqu'elle fournit simplement les interfaces que doivent implémenter les différents kernels : les librairies utiliseront cet interface pour accéder aux ressources matérielles. Le bloc diagramme d'Android sera donc rectifié de la manière suivante :



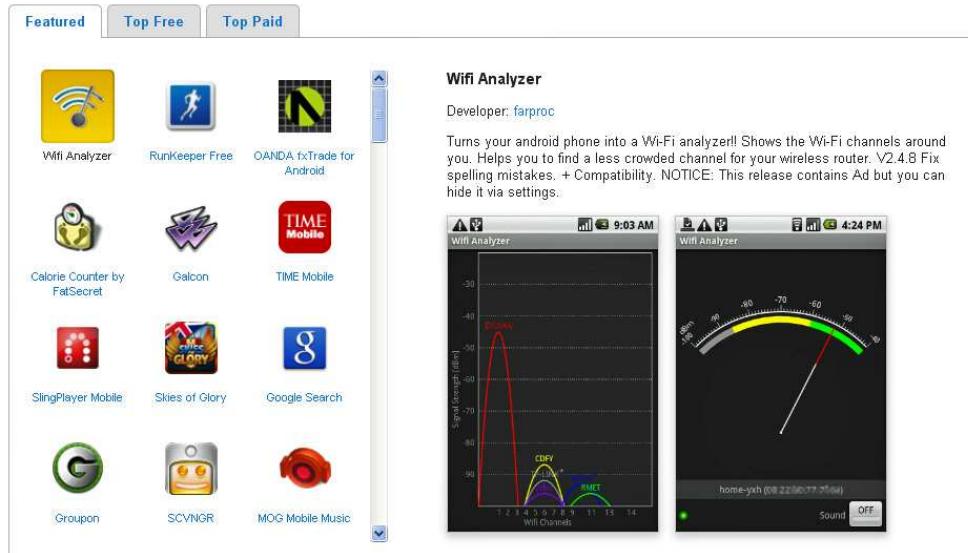
2.4 La couche Java

Du point de vue développement rapide d'applications, il s'agit bien sûr de la couche la plus intéressante. On peut remarquer :

- ◆ les **Cores Librairies** : ce sont les librairies Java de base, de type J2SE 1.5 mais propriétaires et différentes (par exemple, pas de javax.swing);
- ◆ la **machine virtuelle Dalvik** : nous la présenterons plus en détail ci-dessous.

A remarquer encore que

- ◆ si une application Java nécessite du code en C/C++, elle pourra le faire par JNI sur du code développé avec le Native Development Kit (**NDK**);
- ◆ il existe un **Android Market** (<http://www.android.com/market>) sur lequel on peut se fournir des applications gratuites ou payantes (avec une contribution de 30% laissée à Google) proposées par les développeurs Android :



2.5 La couche Application Framework

Il s'agit ici du *toolkit* que toutes les applications utiliseront, quelles qu'elles soient. En fait, ce sont des services qui tournent en arrière-plan et qui sont essentiels au fonctionnement du mobile. Citons :

1) les Core Platform Services

Il s'agit évidemment des services fondamentaux pour le fonctionnement même des applications sur le mobile :

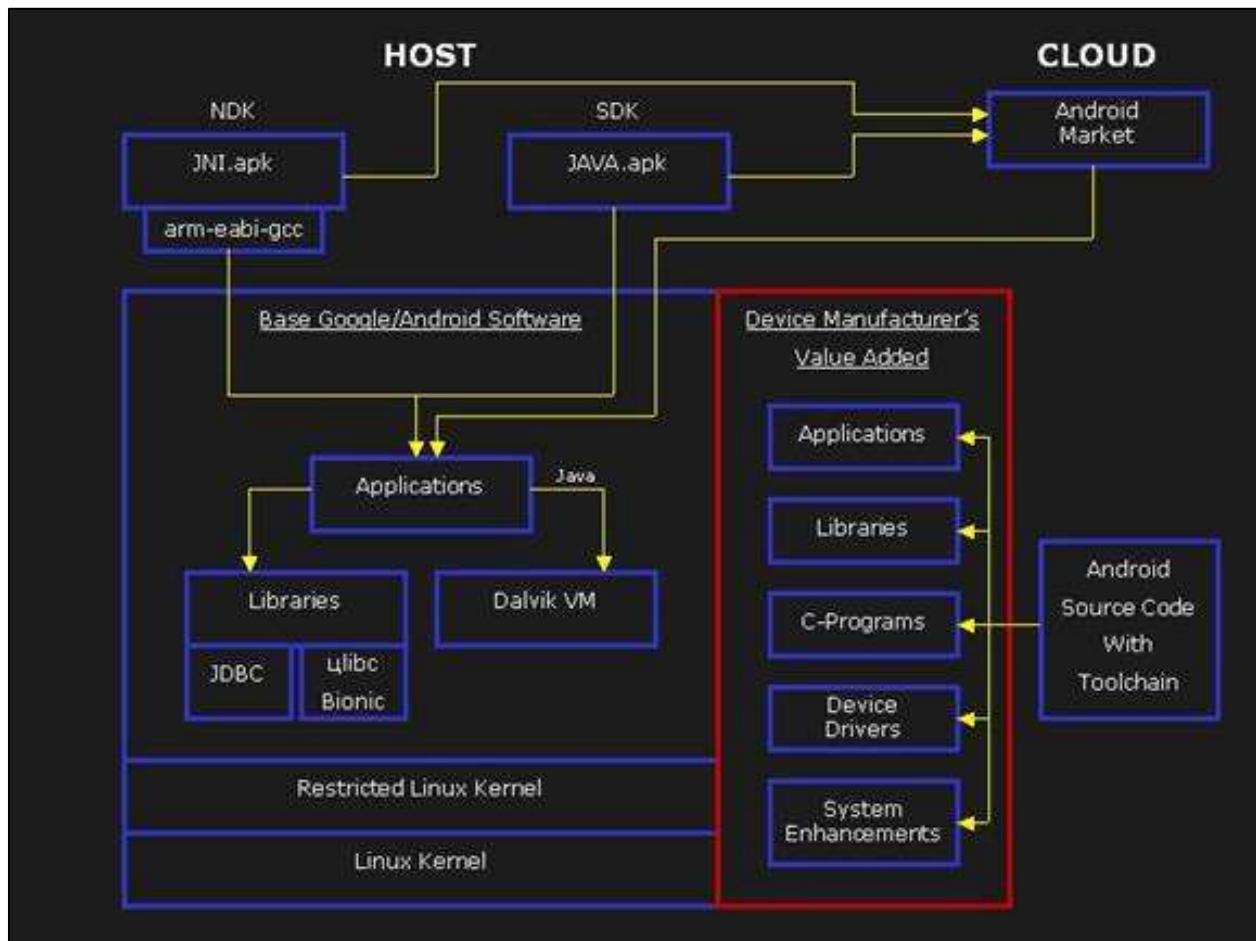
- ◆ Activity Manager : gestion du cycle de vie des applications et de la pile de navigation (pour revenir à l'application précédente quand une application se termine);
- ◆ Package Manager : chargement des fichiers **.apk** (voir ci-dessous);
- ◆ Window Manager : au-dessus du Surface Flinger, il gère les recouvrements des fenêtres;
- ◆ Resource Manager : bon, c'est clair;
- ◆ Content Provider : gestion de partage de données entre applications (provenant par exemple d'une base de données);
- ◆ View System : tous les composants graphiques classiques, comme les listes, grilles, zones d'édition, etc

2) les Hardware Services

Il s'agit des APIs vers le matériel. Les citer suffit à les identifier :

- ◆ Telephony Service
- ◆ Location Service (pour le GPS)
- ◆ Bluetooth Service
- ◆ Wifi Service
- ◆ USB Service

En résumé, on peut donc relire l'architecture d'Android selon le point de vue du développeur de la manière suivante (<http://www2.empress.com/whatsnew/techNews/May2010EmpressOnAndroid.html>) :



3. La machine virtuelle Dalvik

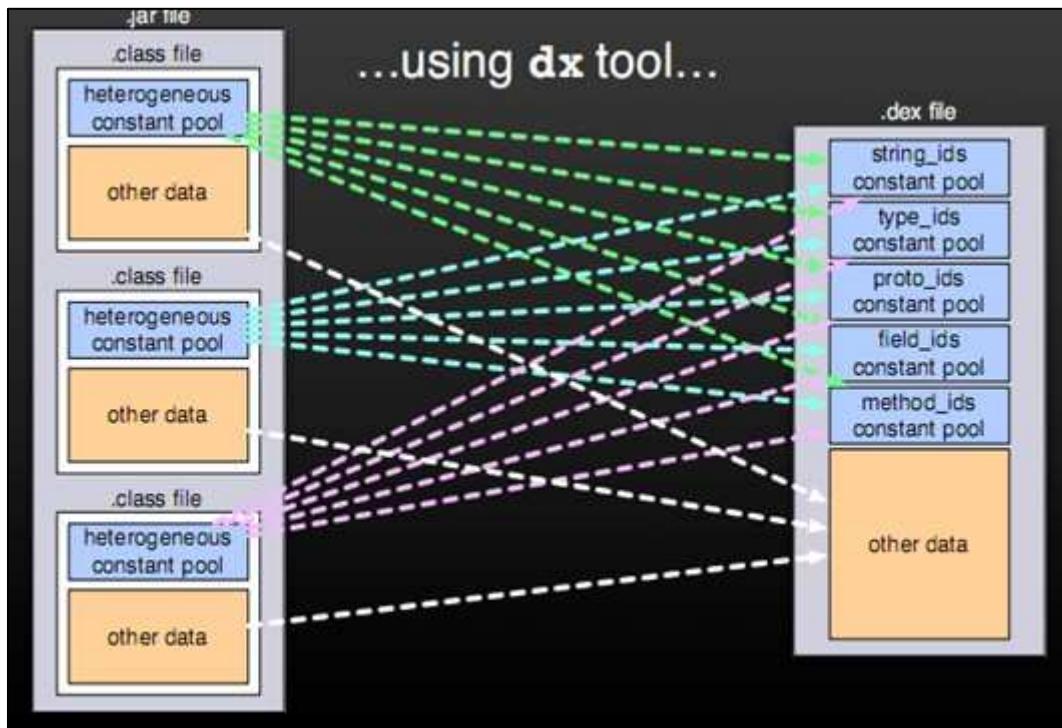
3.1 Une machine orientée registres

La machine virtuelle d'Android est tout à fait particulière. En effet, cette JVM nommée Dalvik (d'après le village des ancêtres de son inventeur, Dan Bornstein ...) possède une **architecture "orientée registres"** [*register-based architecture*], à l'opposé des JVM classiques à **architecture "orientée pile"** [*"stack machine"*]. La différence entre les deux se marque particulièrement pour les instructions de haut niveau qui manipulent des données et qu'il faut convertir en langage machine. Ainsi, sur une machine à pile, le mécanisme automatisé de la pile permet de faire l'économie des adresses dans les instructions machines mais réclame un nombre plus important d'instructions bas niveau pour manipuler les données. Par contre, sur une machine à registres, les instructions bas niveau contiennent directement les adresses des registres source et destination mais, par le fait même, sont plus longues. Les spécifications techniques des formats d'instructions se trouvent sur <http://www.dalvikvm.com>.

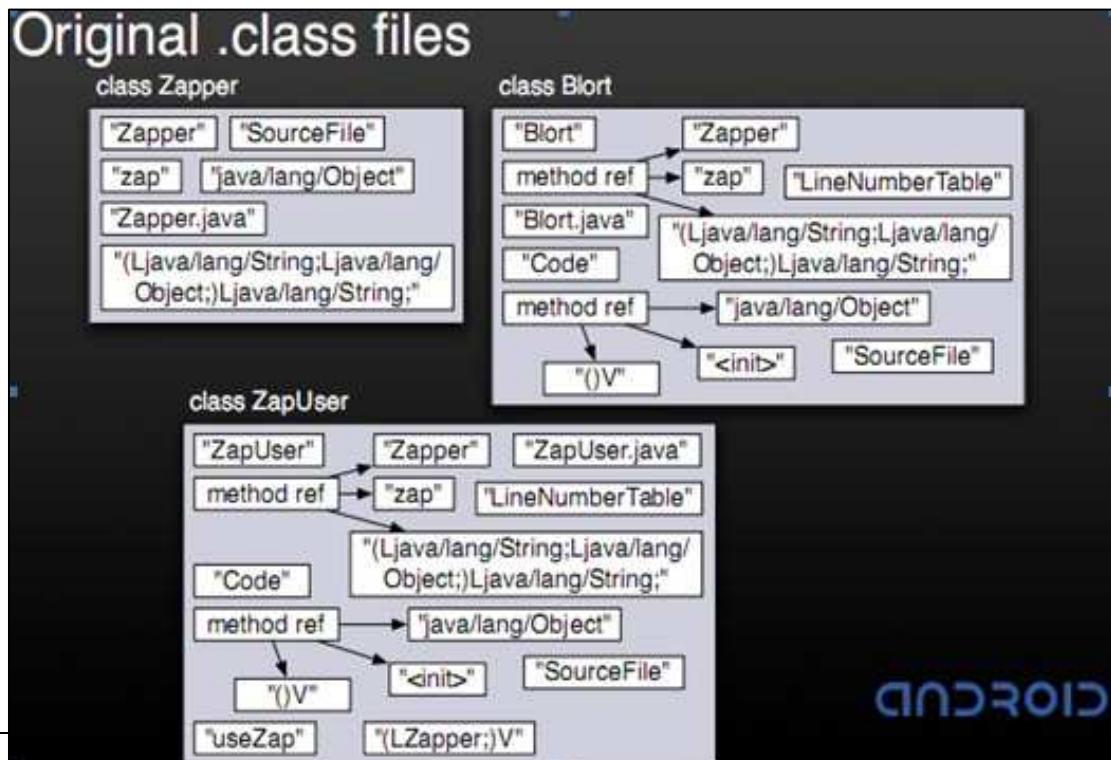
3.2 Un bytecode particulier

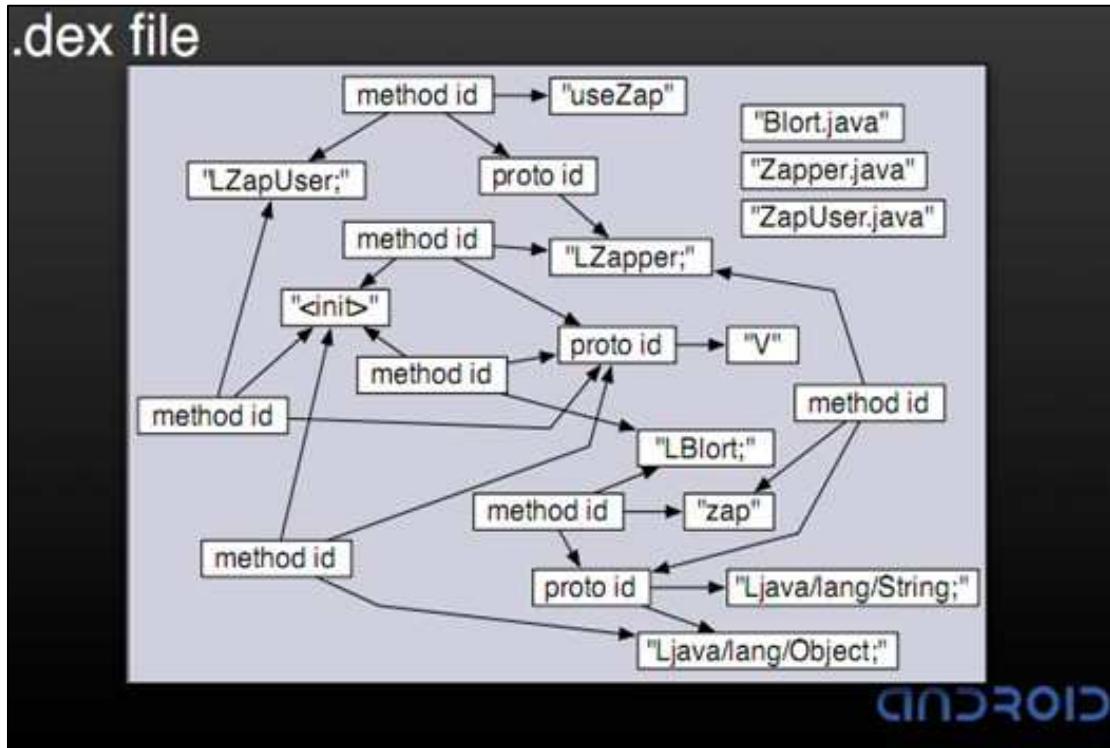
De plus, la JVM Dalvik ne consomme pas du bytecode classique, rassemblés en fichiers **.class**, mais exécute des applications converties en un fichier **.dex** (**Dalvik EXecutable**). Ce format dex est plus particulièrement adapté aux mobiles dotés d'un processeur de faible puissance et/ou d'une mémoire centrale limitée. Par exemple, les

références multiples sont restructurées en des références uniques à un pool, supprimant ainsi les redondances et allégeant donc l'ensemble produit. Le bytecode qui sera fabriqué par le compilateur est donc converti en un nouveau set d'instructions Dalvik (au moyen d'un outil appelé dx) : on parle encore d'"exécutable Dalvik". Le schéma suivant illustre le processus de transformation :



Un tel exécutable peut encore être modifié lors de son installation sur le mobile visé, ceci dans un but d'optimisation (modification du byte order, éviction de classes non utilisées, etc). Un exemple schématique permet de montrer la différence entre un fichier .jar contenant des fichiers .class et un fichier .dex (joyeusement pillé chez <http://www.ophonesdn.com/article/show/15>) :



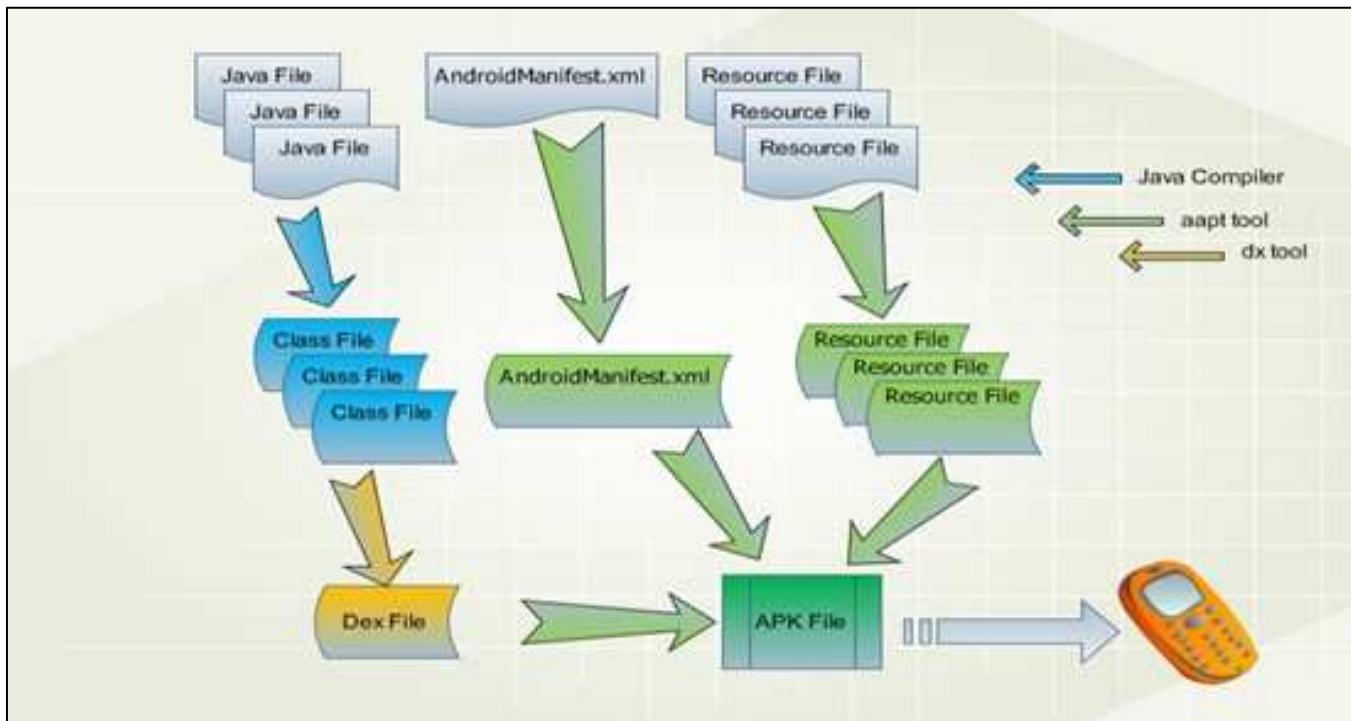


3.3 Le fichier de distribution apk

L'empaquetage final d'une application Android est réalisée dans un fichier **.apk** (Android PacKage), qui est une variante du format des fichiers **.jar** : on peut donc l'ouvrir avec les outils classiques (7-Zip, WinZip, WinRar, etc). Le type MIME est **application/vnd.android.package-archive**. Bien logiquement, ce fichier APK contient, outre l'habituel répertoire META-INF, les éléments suivants :

- ◆ le fichier **.dex**;
- ◆ un fichier xml décrivant l'application (**AndroidManifest.xml** - nous y reviendrons);
- ◆ les fichiers de ressources **.arsc** et **.res** : en fait, **.res** est le répertoire qui contient les ressources proprement dites tandis que le fichier **.arsc** est en fait la mise à plat de la table des ressources (que nous trouverons dans nos développements dans le fichier **R.java** – nous y reviendrons aussi).

Très schématiquement, les choses se passent donc ainsi :



4. Les composants applicatifs Android

On l'a dit, une application Android est écrite en Java, mais il s'agit donc d'un Java non standard. Ainsi, par exemple, une telle application n'a pas de fonction `main()` qui servirait d'unique point d'entrée. En fait, une application [task] peut utiliser ses propres composants ou se servir des composants d'autres applications (si celles-ci le permettent) : elle démarre simplement la partie de code dont elle a besoin.



Mais qu'entend-on alors par "application" ? Le fichier apk dont il a déjà été question sera placé sur le mobile, constitue de facto ce qu'Android appelle une application. Celle-ci fonctionne au sein d'un processus Linux, avec son ID unique et sa propre machine virtuelle, ce qui assure l'isolation du code. Elle possède seule (du moins à priori – on peut modifier cela) les permissions pour en utiliser les composantes et ressources. **Une application se présente donc comme un groupe d'un certain nombre de composants susceptibles d'être instanciés et exécutés** : en fait, ils sont "empilés" selon les appels, le composant du sommet de la pile étant celui qui est actif. Par défaut, tout se passe dans un seul thread, ce qui implique que toute activité doit s'abstenir d'effectuer des tâches bloquantes. Si cela doit néanmoins être le cas, à sa charge de lancer un thread secondaire (instance de la classe `Thread`) pour gérer cette tâche; Android fournit des classes utilitaires pour permettre le dialogue entre ces threads au moyen de messages (classes `Handler`, `Looper`, `HandlerThread`).

Concernant les composants proprement dits, il peut s'agir

- ◆ d'une activité [activity] : objet instance de la classe **Activity** dont on peut lancer l'exécution pour réaliser une tâche quelconque utilisant un GUI (il possède une fenêtre par défaut); c'est l'association de ces activités qui permet de définir un interface graphique complet pour l'utilisateur de l'application;

- ◆ d'un service : objet instance de la classe **Service** dont on peut lancer l'exécution en tâche de fond et qui ne possède donc pas d'interface graphique (exemple typique : une musique de fond); il est évidemment possible de démarrer un service mais aussi de se connecter à lui si il est déjà en cours d'exécution;
- ◆ d'un récepteur des annonces broadcast [*broadcast receiver*] : les annonces peuvent provenir du système (batterie faible, fuseau horaire modifié, etc) ou d'une autre application (exemple typique : fin d'un téléchargement);
- ◆ d'un fournisseur de contenu [*content provider*] : un tel objet rend des ressources de l'application disponibles aux autres applications (celles-ci utiliseront un "content resolver", instance de ContentResolver) pour viser le "content provider" en question et dialoguer avec lui).

Les trois premiers types de composants sont activés au moyen d'un "**intent**" : il s'agit d'un message asynchrone spécifiant une action et contenant l'URI du composant à manipuler ou de l'action à initier. Ceci permet aux composants de communiquer car, de base, chaque application fonctionne dans une sandbox distincte. C'est le rôle d'un objet implementant la classe abstraite **Context** (objet fourni par le framework Android au démarrage) de provoquer les démaragements des activités et services, avec par exemple :

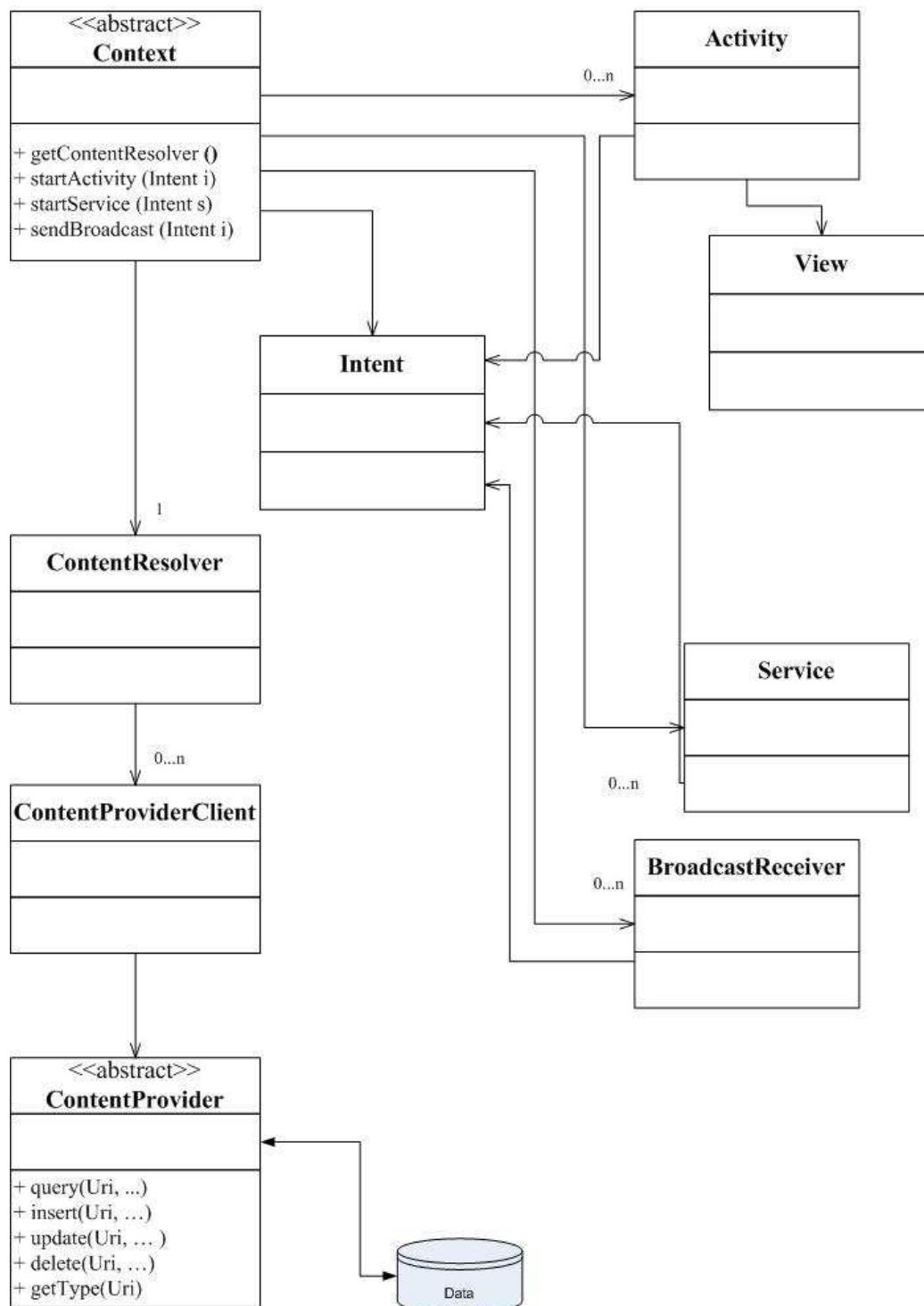
```
public abstract void startActivity (Intent intent)
public abstract ComponentName startService (Intent service)
public abstract boolean stopService (Intent service)
public abstract boolean bindService (Intent service, ServiceConnection conn, int flags)
public abstract void sendBroadcast (Intent intent)
```

et de permettre l'accès aux ressources de l'application, avec par exemple :

```
public abstract ContentResolver getContentResolver () (qui lui-même dispose de la méthode
public final ContentProviderClient acquireContentProviderClient (Uri uri) )

public abstract Resources getResources ()
```

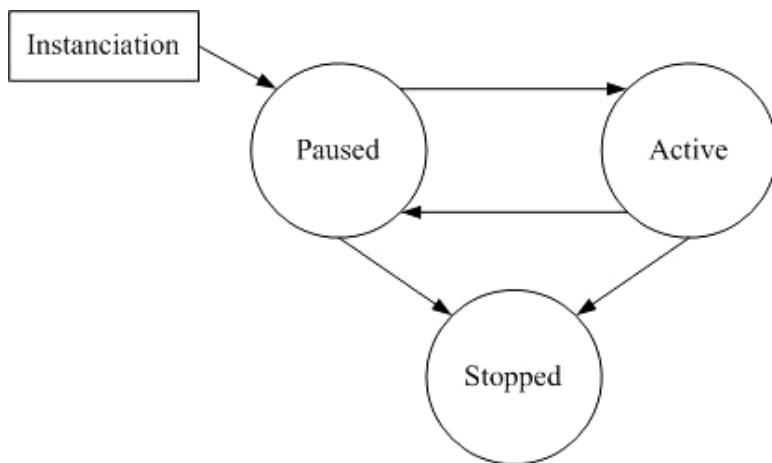
Schématiquement :



5. Le cycle de vie d'une activité Android

Un objet "Activity", instance d'une classe dérivée de la classe android.app.Activity, est donc un objet dont on peut lancer l'exécution pour réaliser une tâche quelconque utilisant un GUI, en fait une instance d'une classe dérivée de android.view.View. Une application peut consister en une seule activité ou en un groupe d'activités, l'une d'entre elles étant désignée comme celle de départ et chacune pouvant passer la main à une autre.

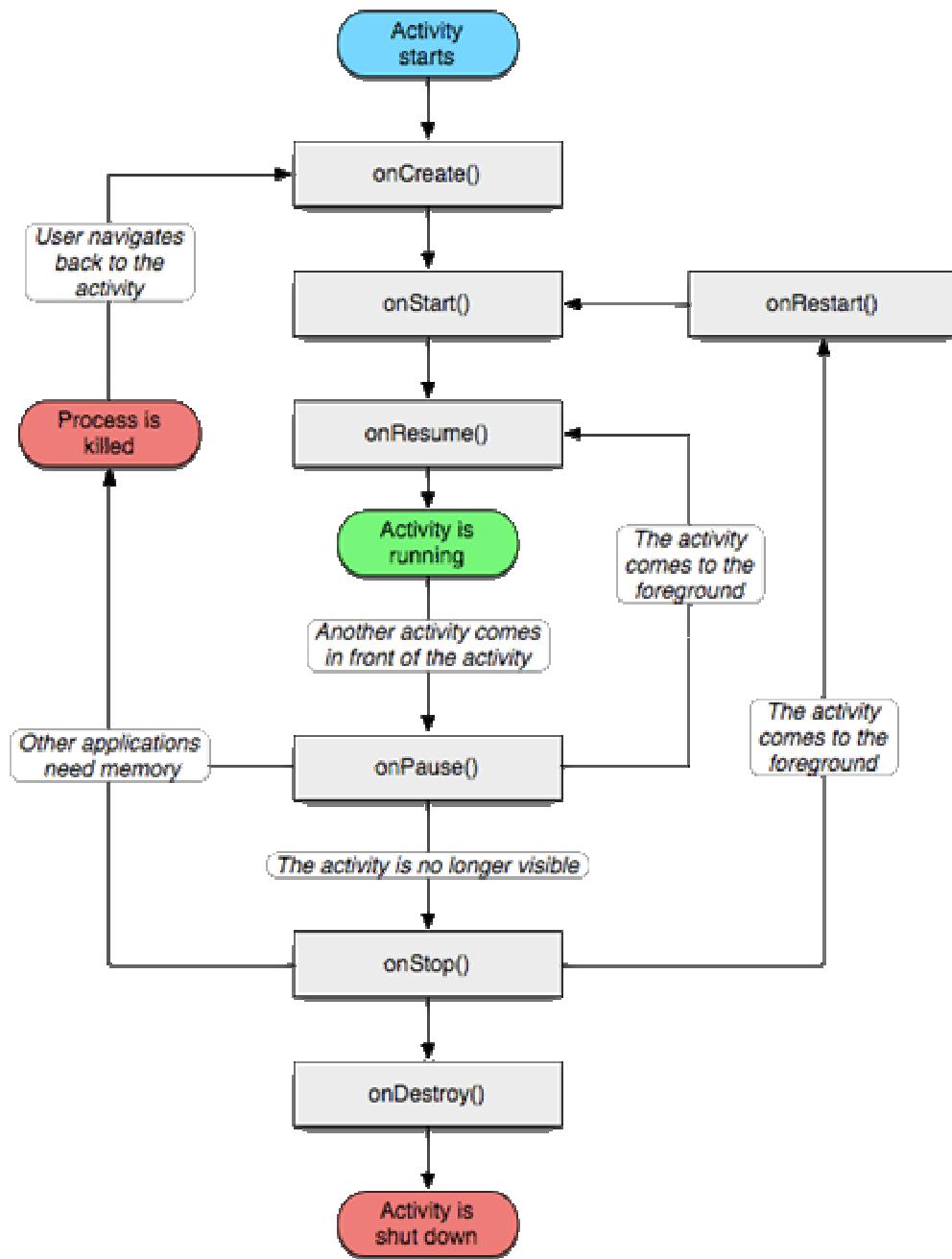
Le cycle de vie d'une activité rappelle classiquement celui d'une MIDlet de J2ME :



Le passage d'un état à l'autre est notifié à l'application par l'appel aux méthodes (protected) :

cycle de vie de base	
void onCreate(Bundle savedInstanceState);	void onDestroy();
cycle de vie visible	
void onStart();	void onStop();
cycle de vie à l'arrière-plan	
void onPause();	void onRestart();
	void onResume();

Les transitions sont schématisées dans la documentation Android de la manière suivante :



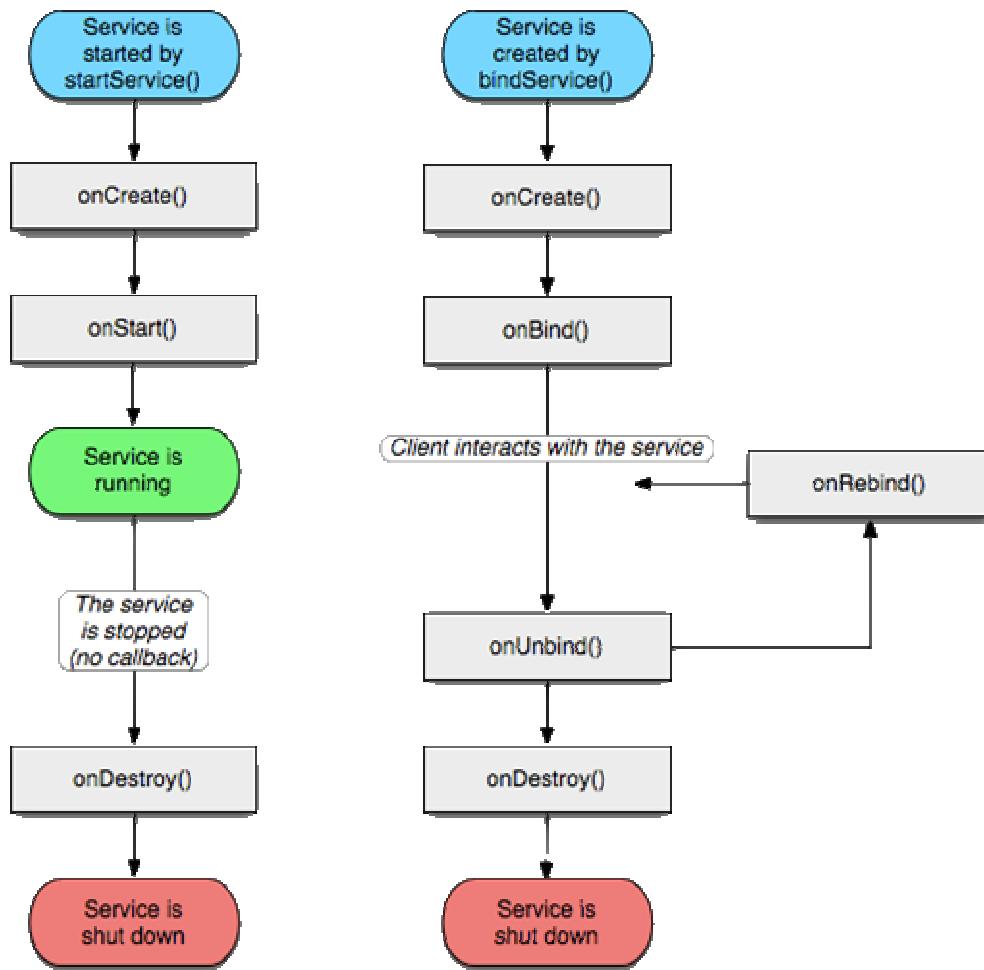
On remarque donc que si une activité est à l'arrêt (temporaire ou définitif), le processus qui lui donne son contexte peut être détruit en cas de besoin de mémoire pour d'autres applications.

6. Le cycle de vie d'un service Android

Un service diffère d'une activité par le fait qu'il attend que l'on fasse appel à lui :

- ◆ il peut être démarré en cas de besoin, puis arrêté (par lui-même ou par un autre composant);
- ◆ il peut se mettre en attente d'une connexion établie par un autre composant qui l'utilisera en utilisant les méthodes d'un interface qu'il a défini et rendu public.

Schématiquement :



7. Une application Android de base

Une application peut être constituée de plusieurs activités, mais l'utilisateur n'interagit qu'avec une seule d'entre elles à un instant donné. A priori (même si ce n'est pas indispensable), une application possède au moins une activité. Une classe basique dérivée d'**Activity** va *typiquement redéfinir deux méthodes* :

- 1) protected void **onCreate** (Bundle savedInstanceState)
Cette méthode est appelée quand l'activité est démarrée.

Le paramètre instance de android.os.**Bundle** est une implémentation de l'interface android.os.**Parcelable** caractérisant les classes qui peuvent être sauveées ou restaurées à partir d'un objet android.os.**Parcel**, celui-ci constituant ce qui est transmis par IPC dans le contexte de la gestion des processus sur le mobile (il ne s'agit donc en aucun cas d'une sérialisation au sens habituel du terme).

Classiquement, la méthode onCreate() appellera notamment :

```
public void setContentView (View view)
ou
public void setContentView (int layoutResID)
```

la première forme utilise une classe android.view.View qui désigne simplement une zone rectangulaire susceptible de contenir des graphiques et d'interagir avec l'utilisateur; la deuxième faisant référence à une vue considérée comme une ressource définie dans R.java (déjà évoqué - nous allons y revenir).

2) protected void onPause ()

appelée quand l'activité passe à l'arrière-plan sans être détruite pour la cause.

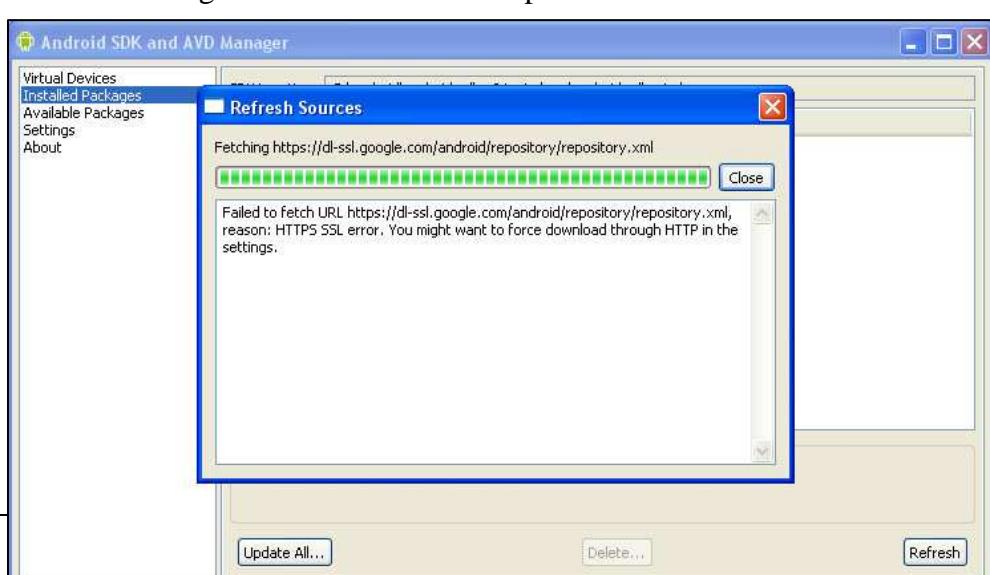
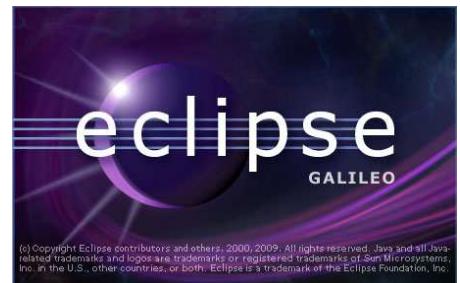
En pratique, on peut développer en ligne de commande ☺ mais il nous faudrait évidemment une EDI pour nous simplifier la vie ☺ ...

8. Le plugin Android pour Eclipse

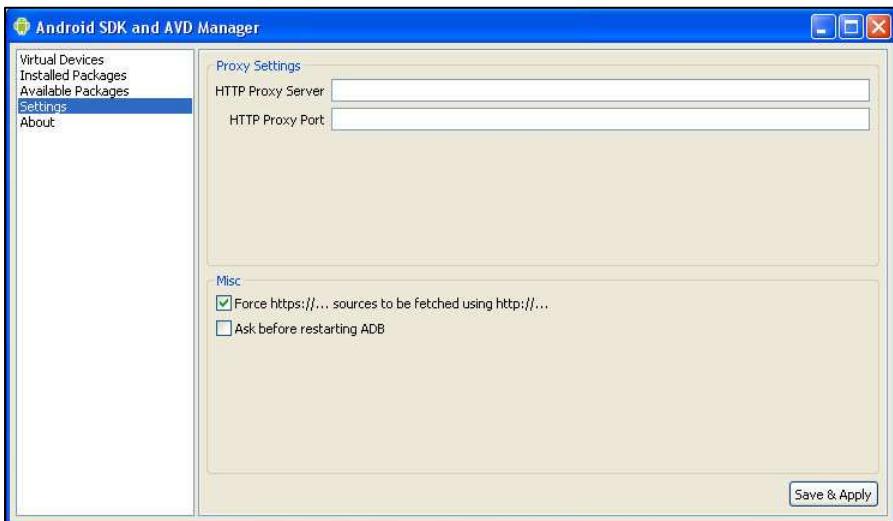
L'EDI le plus simple à utiliser pour développer des applications Andoid n'est pas NetBeans IDE mais bien Eclipse. On s'y serait presque attendu, puisqu'on se trouve dans le domaine des développeurs "ni dieu, ni maître", celui de l'open source. Et c'est donc le concurrent de Netbeans qui se place dans l'obéissance open source qui convient le mieux pour le développement Android (même si il existe un plugin pour Netbeans : voir http://wiki.netbeans.org/IntroAndroidDevNetBeans#Introducing_Android_Development_with_NetBeans).

Une fois Eclipse installé (dans notre cas, il s'agit de Eclipse SDK 3.5.1 - Galileo), il nous faut donc un **SDK Android** sous forme de plugin à placer dans Eclipse.

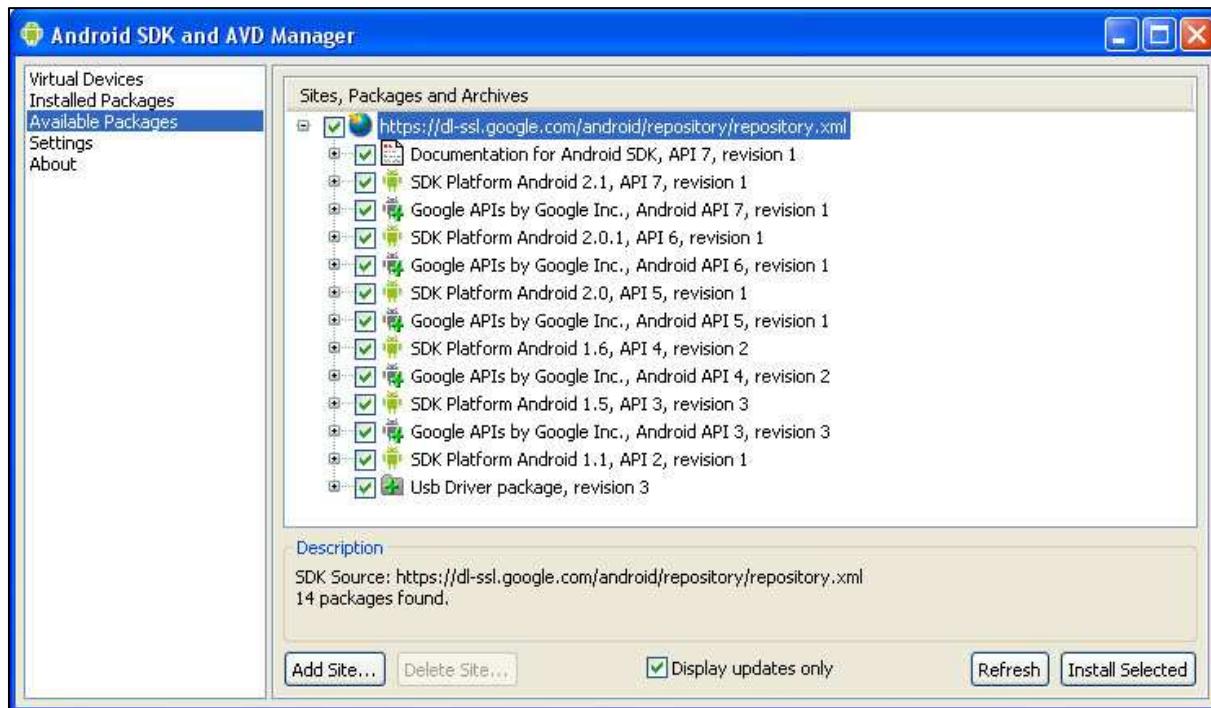
a) Commençons par nous procurer ce SDK et à l'installer : il peut être obtenu sur <http://developer.android.com/sdk/index.html>. Une fois le fichier android-sdk_rOX-windows.zip décompressé (si on travaille sous Windows, avec X=4 ou X=6), on peut exécuter le programme d'installation. En fait, pour X=6, les choses passent simplement mais pour X=4, le résultat est prévisible puisque l'on essaie d'établir un dialogue SSL sans la moindre précaution :



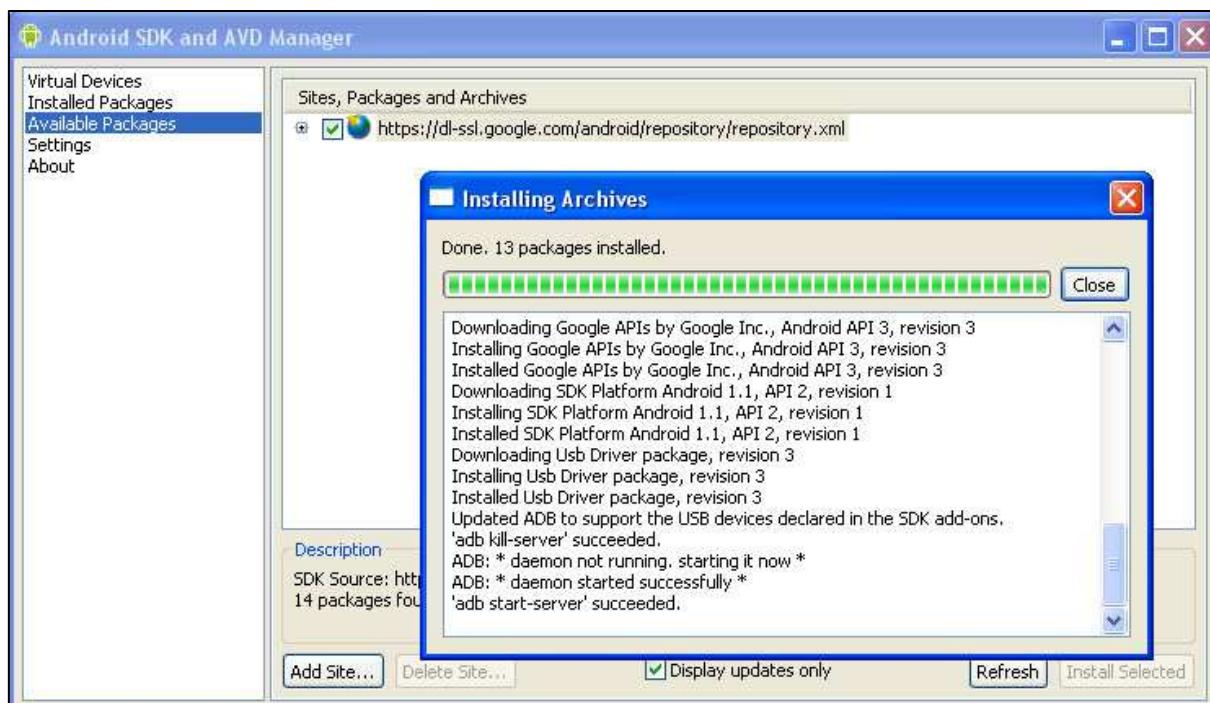
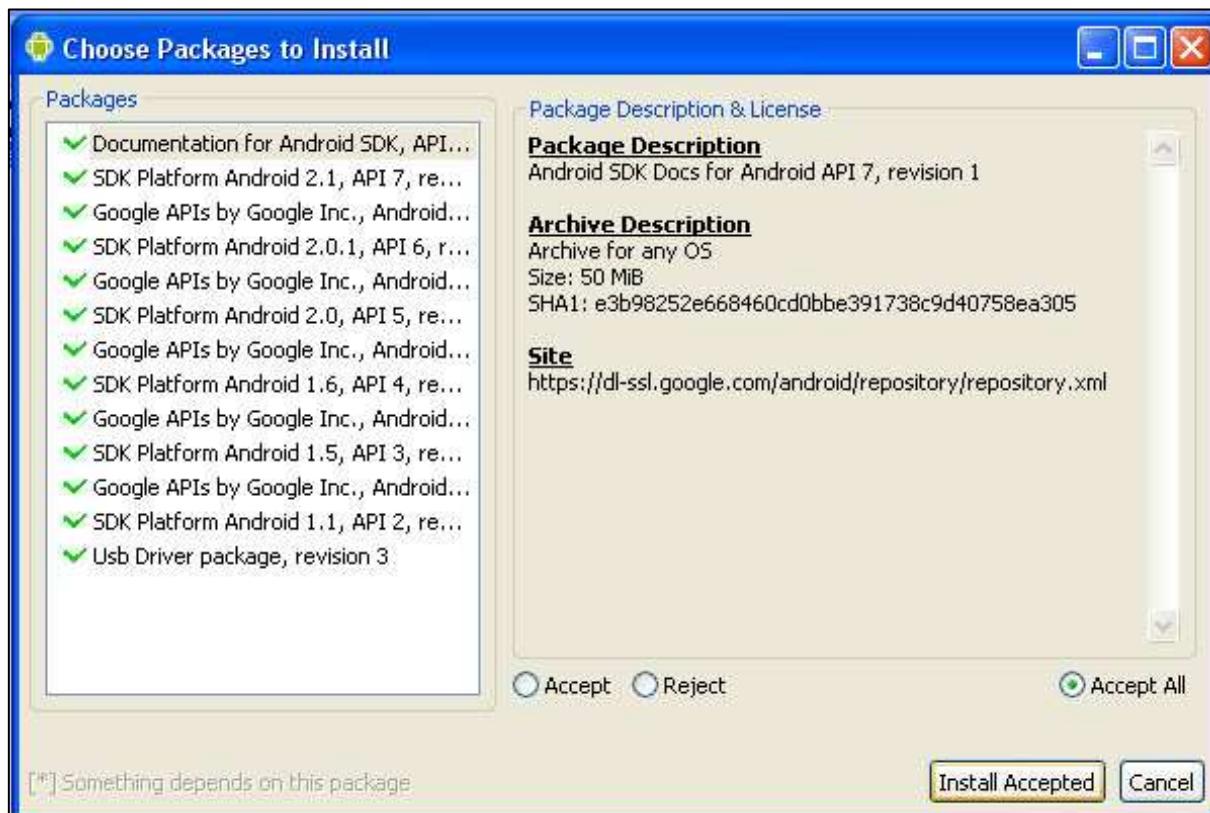
Forçons donc, puisque 'on nous le suggère :



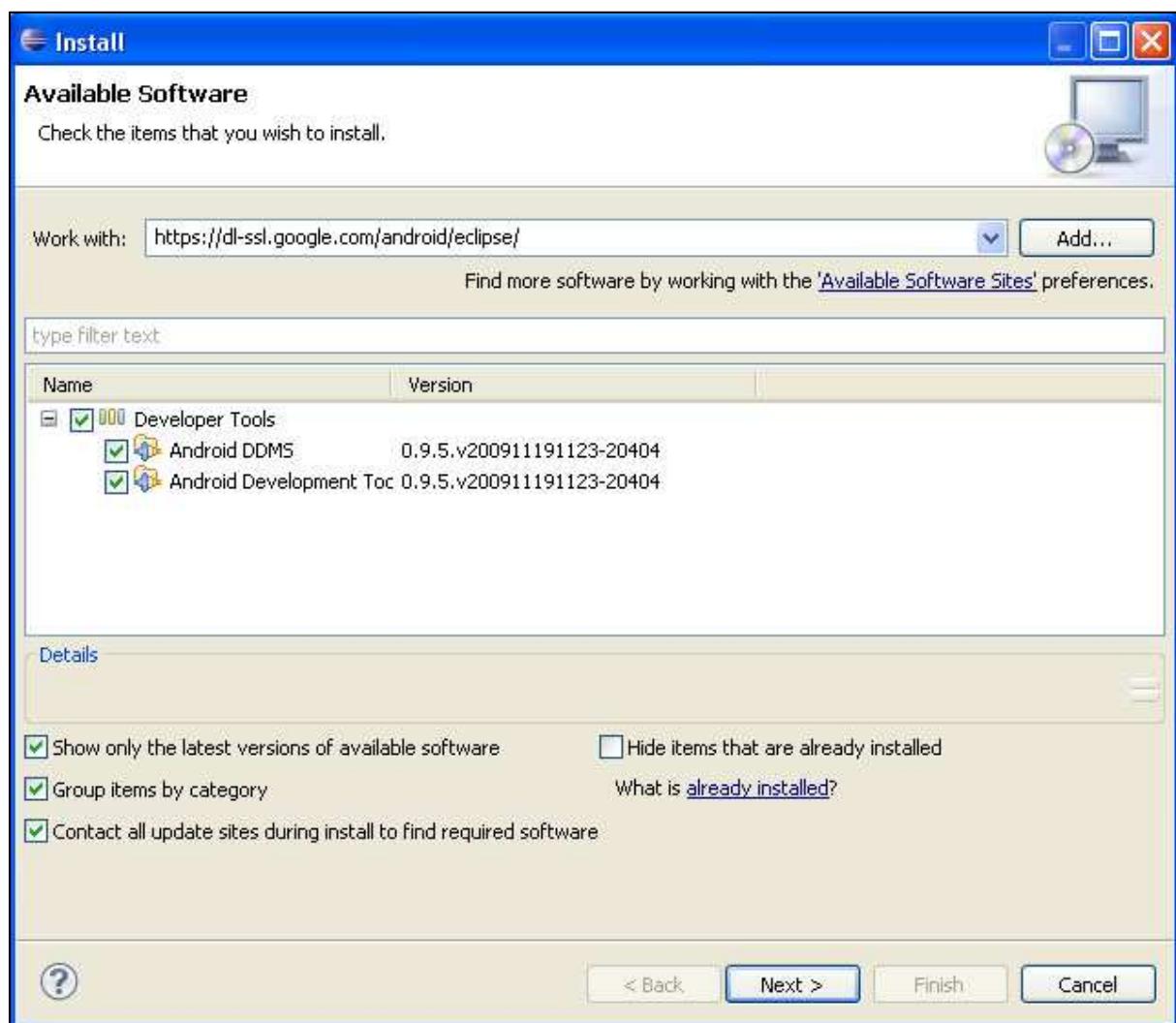
Dans les deux cas, les **packages Android** finissent par être disponibles :



et après avoir sélectionné les packages nécessaires, on peut lancer l'installation.

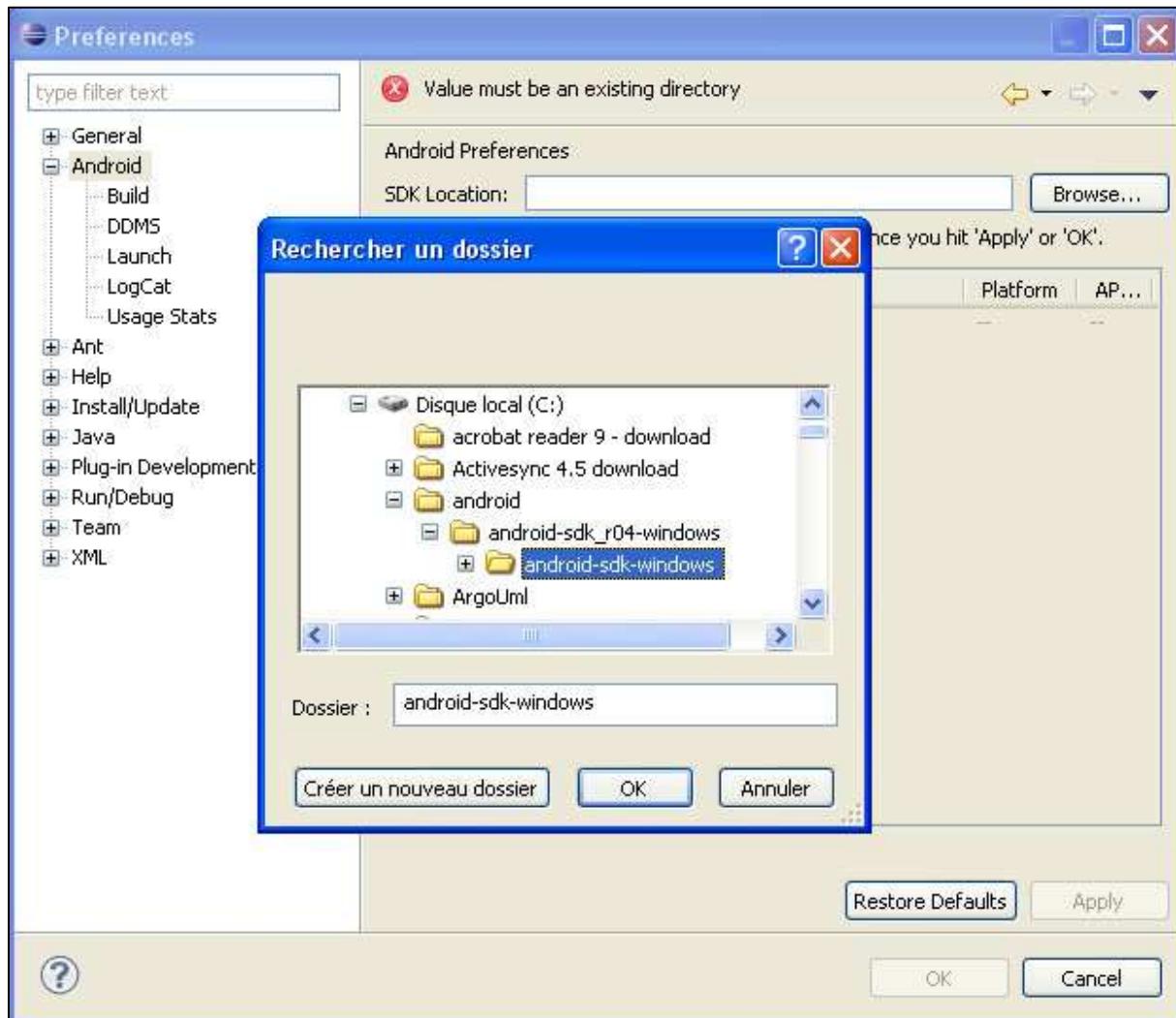


b) Passons à l'installation du **plugin** qui va utiliser ce SDK, installation analogue, par exemple, à celle du plugin que l'on ajoute pour disposer d'un Visual Editor pour gérer des interfaces GUI en Swing (*pour le lecteur peu habitué à Eclipse, voie l'introduction à Eclipse en annexe*) : dans le menu Help → Install New Software, nous le téléchargeons depuis <https://dl-ssl.google.com/android/eclipse/> puis cochons les éléments à installer :

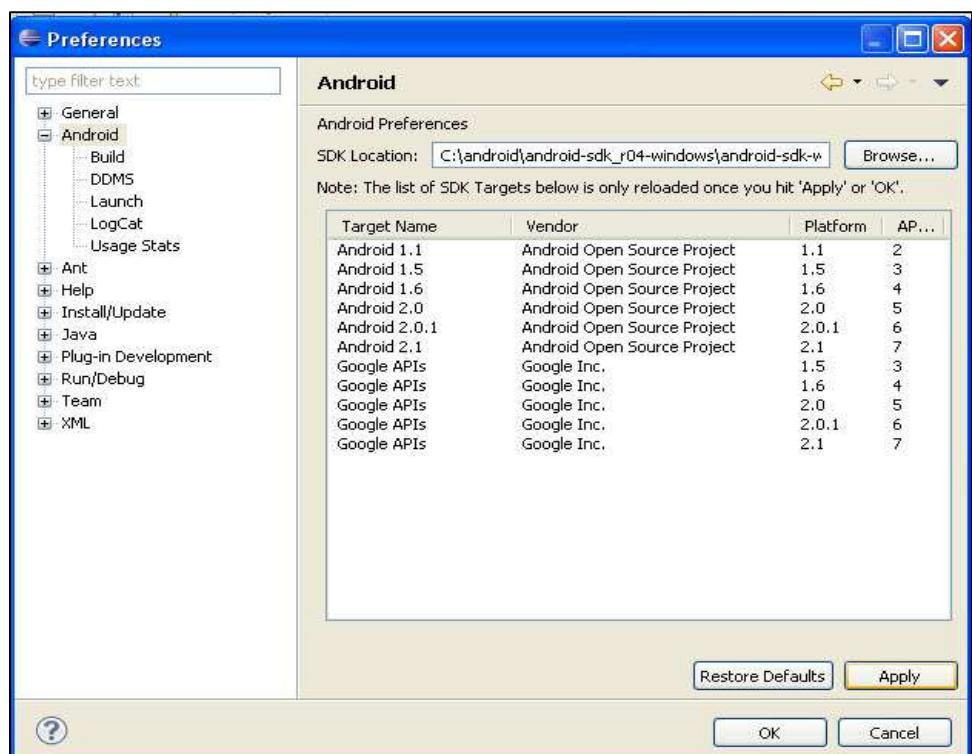


La suite de la procédure (avec acceptation de la licence) est sans surprise. Une fois l'installation terminée, on peut redémarrer Eclipse.

c) Il faut encore définir le **répertoire du SDK d'Android** dans les paramètres d'Eclipse : le menu Window → Preferences :



avec comme résultat :

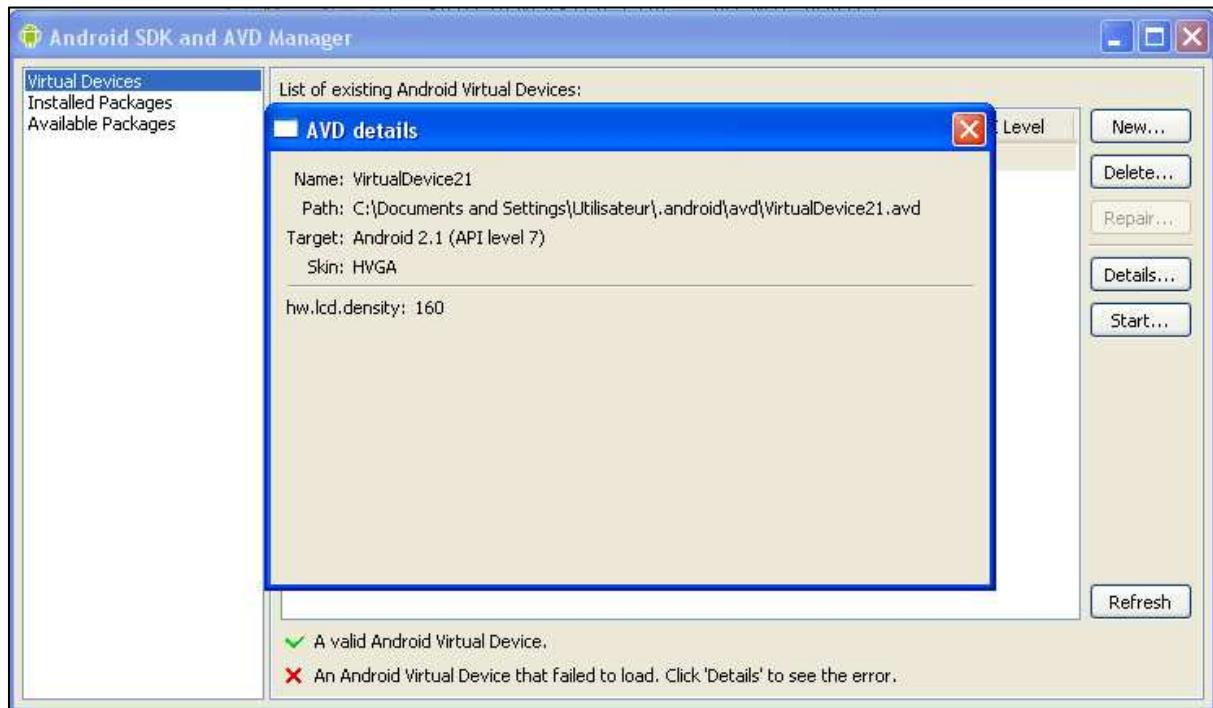


La présence d'une icône supplémentaire (un mobile noir (ver.4) ou une flèche vers le bas (ver.6)) indique que le plugin est en place :



- mais on peut aussi en voir la présence par le menu selon Window → Android SDK and AVD Manager.

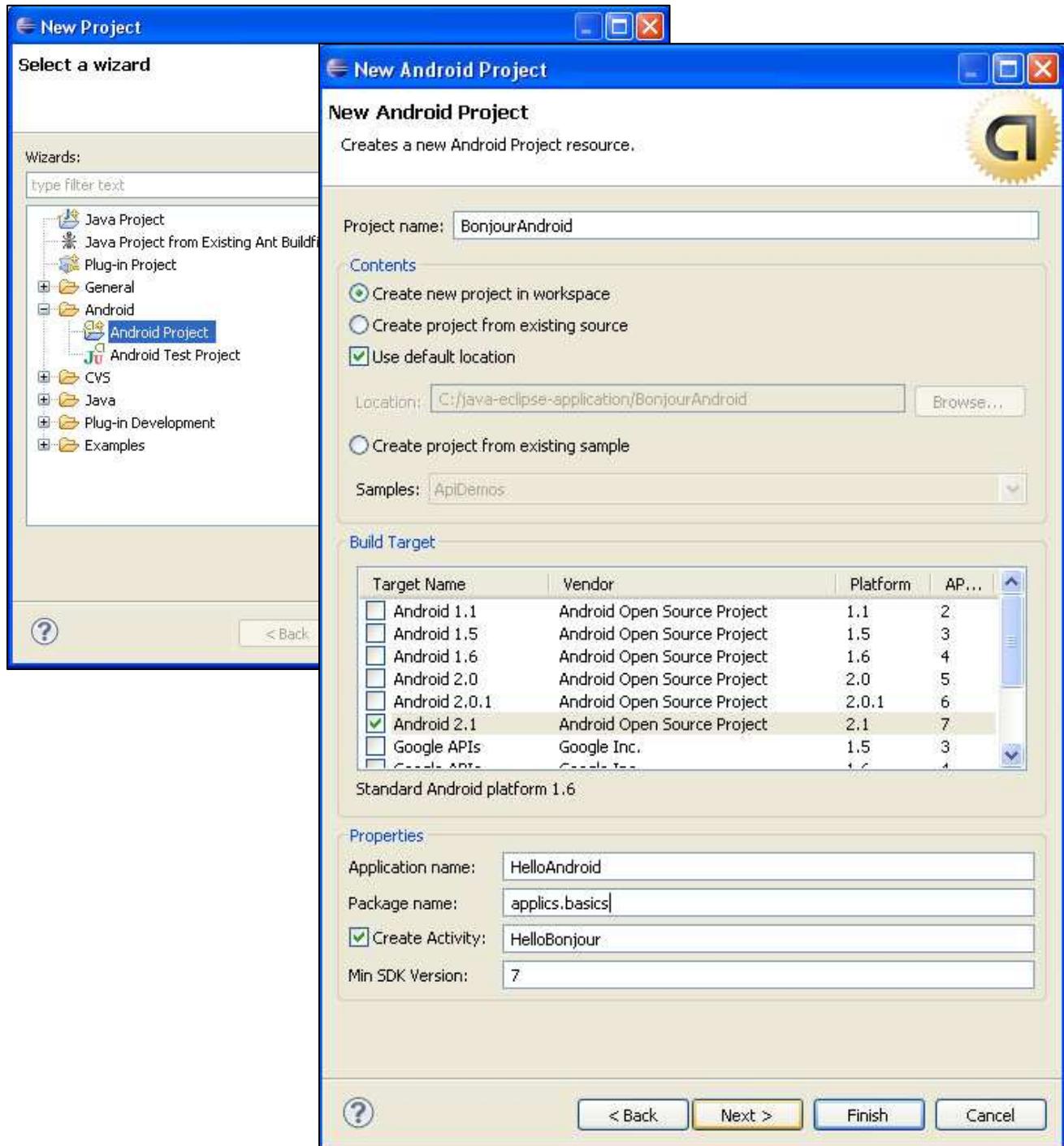
d) Mais il faut encore installer un **AVD** (Android Virtual Device), c'est-à-dire **une version de l'émulateur de mobile** configuré selon certaines options pour représenter au mieux un mobile qui existe dans la réalité. L'étape est obligatoire, car il n'y a pas d'AVD par défaut. Cela peut se faire par un clic sur l'icône des mobiles : on obtient la boîte de dialogue "Android SDK and AVD Manager" dans laquelle on peut définir un appareil virtuel. On en arrive, par exemple, à ceci :



9. Développement d'une application Android basique avec Eclipse

9.1 Génération d'une application basique

Avec Eclipse, on peut générer très simplement une application de base :



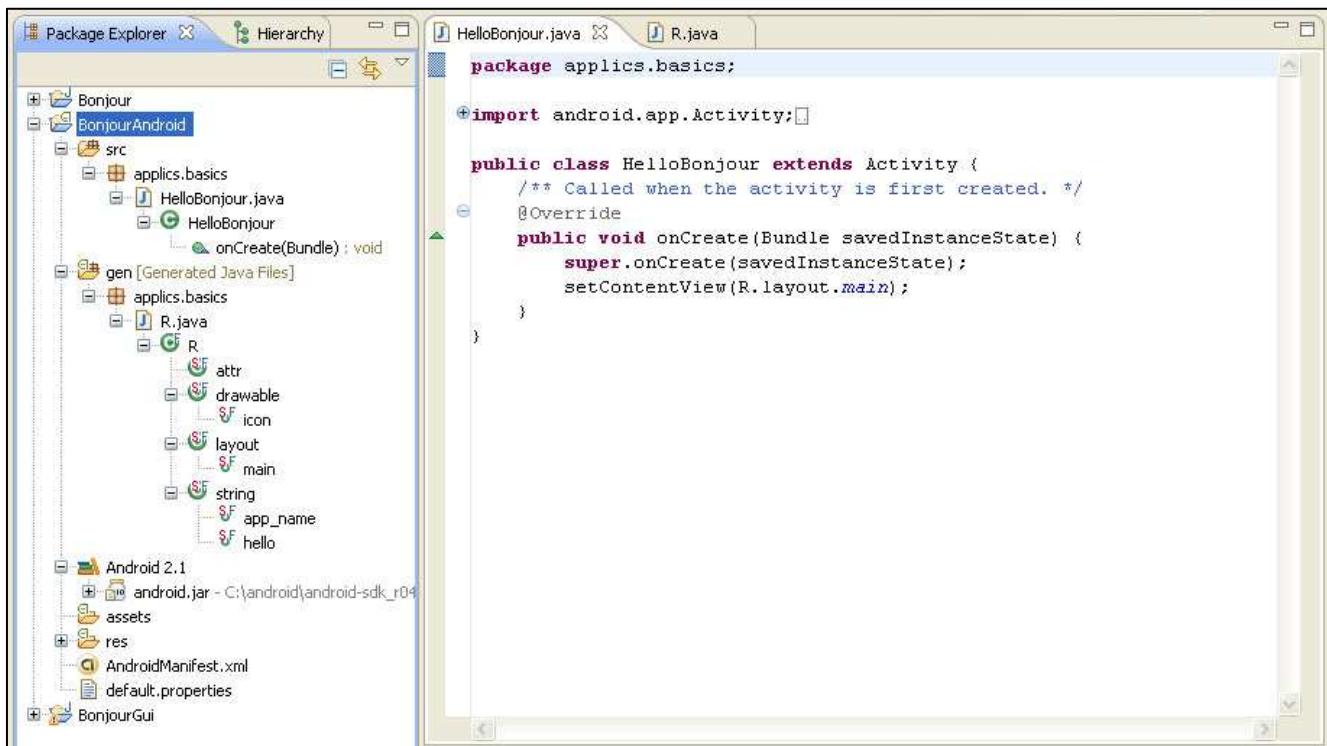
Dans la foulée, on peut aussi faire créer un projet de test. Il faut surtout savoir que

- ◆ le nom du package doit être unique par rapport à l'ensemble de tous les packages installés sur le mobile qui utilisera l'application;
- ◆ la "create activity" est l'activité principale;
- ◆ le niveau de l'API choisi doit être compatible vers le haut avec l'AVD cible.

Après un moment de réflexion :



on obtient :



soit plus clairement :

HelloBonjour.java

```
package applics.basics;  
  
import android.app.Activity;  
import android.os.Bundle;  
  
public class HelloBonjour extends Activity {  
    /** Called when the activity is first created. */  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.main);  
    }  
}
```

On peut donc constater qu'au lancement de notre application, c'est l'activité définie (instance de la classe Activity du package android.app) qui est lancée et c'est la méthode `setContentView()` qui est appelée : elle détermine bien sûr le composant graphique qui sera affiché. En fait, elle peut désigner ce composant de deux manières :

- ◆ soit à partir d'une élément défini dans la classe R générée par le SDK (voir ci-dessous) :

```
public void setContentView (int layoutResID)
```

- ◆ soit en utilisant un objet View explicite (c'est ce que nous ferons sous peu) :

```
public void setContentView (View view)
```

```
public void setContentView (View view, ViewGroup.LayoutParams params)
```

9.2 Les ressources générées

La classe R est donc une classe qui a été générée dans le même package que la classe activité que nous avons créée. Son code est relativement cryptique :

R.java

```
/* AUTO-GENERATED FILE. DO NOT MODIFY.  
*  
* This class was automatically generated by the  
* aapt tool from the resource data it found. It  
* should not be modified by hand.  
*/  
  
package applics.basics;  
  
public final class R {  
    public static final class attr {  
    }  
    public static final class drawable {  
        public static final int icon=0x7f020000;  
    }  
    public static final class layout {  
        public static final int main=0x7f030000;  
    }  
    public static final class string {  
        public static final int app_name=0x7f040001;  
        public static final int hello=0x7f040000;  
    }  
}
```

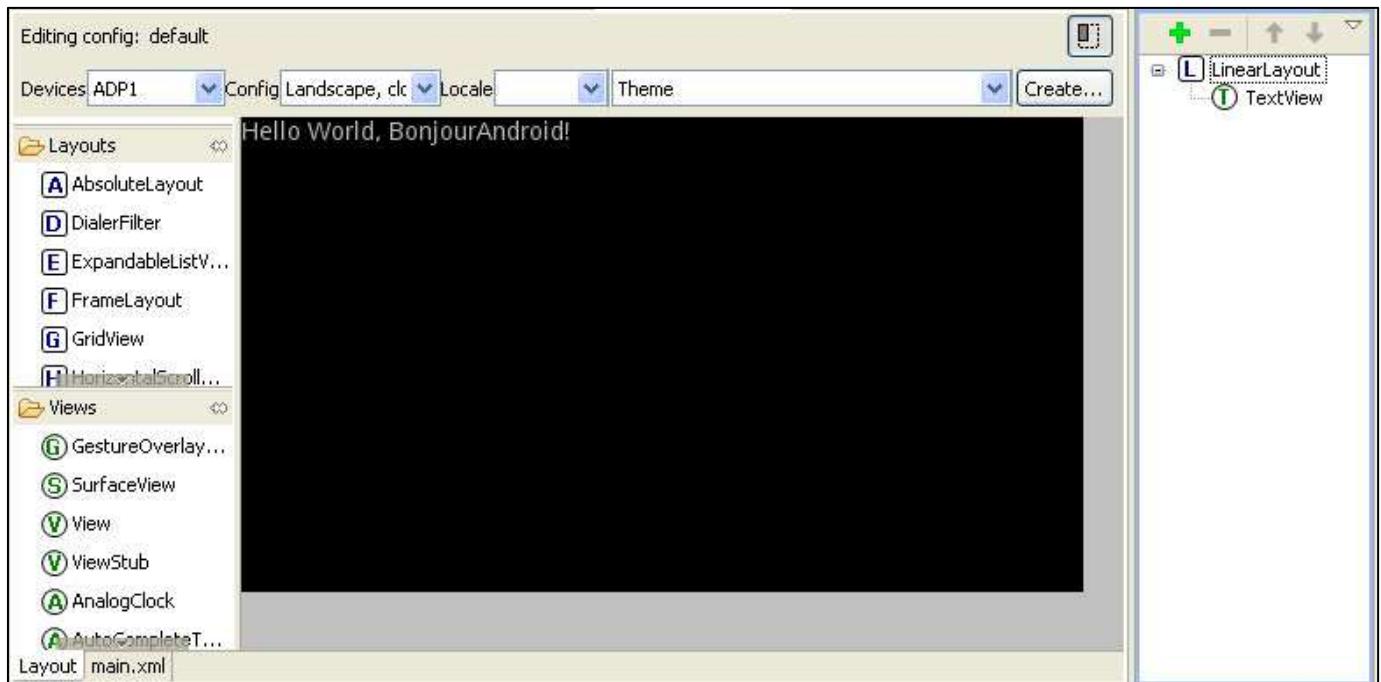
On aura compris qu'elle sert essentiellement à encapsuler les références internes d'objets Java comme le layout actif ou la chaîne de caractères qui sera affichée (et même l'icône qui sera utilisée dans la liste des applications installées). Les composants visuels et les ressources proprement dits sont décrits dans le nœud "res" du projet, sous forme de fichier XML :

main.xml (onglet layout)

```
<?xml version="1.0" encoding="utf-8"?>  
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:orientation="vertical"  
    android:layout_width="fill_parent"
```

```
    android:layout_height="fill_parent"
    >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
/>
</LinearLayout>
```

qui peut aussi se voir (en choisissant "Open with" → "Android Layout Editor" dans le menu contextuel obtenu par clic droit, puis l'onglet Layout) comme ceci (c'est plus joli ;-)) :

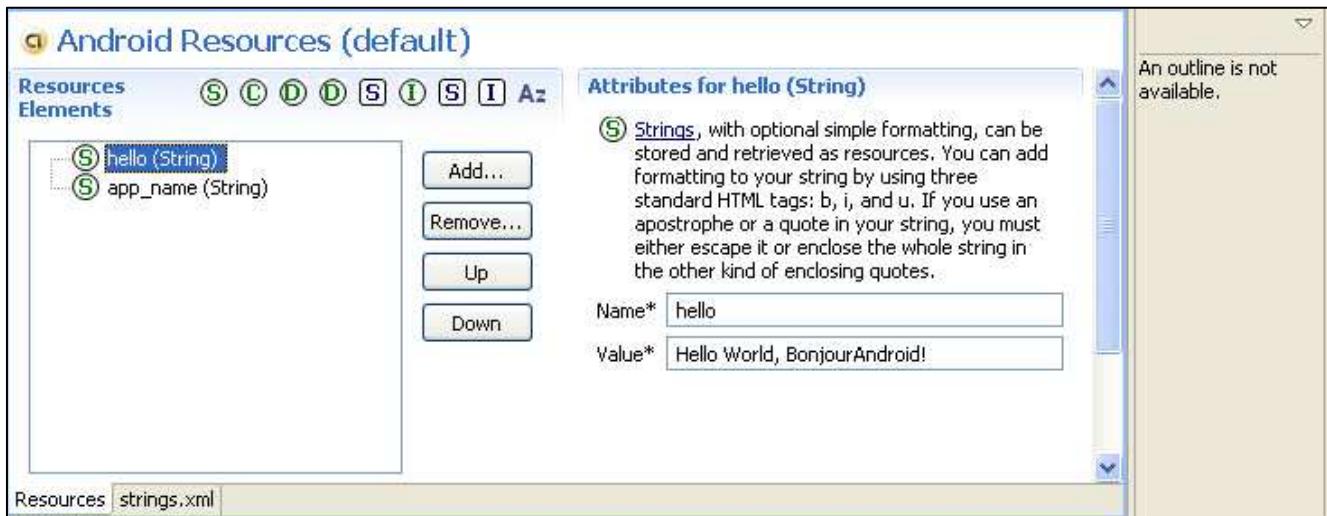


et

strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello World, HelloBonjour!</string>
    <string name="app_name">HelloAndroid</string>
</resources>
```

qui peut également (en choisissant "Open with" → "Android Resource Editor" dans le menu contextuel obtenu par clic droit) se voir de manière plus friendly :

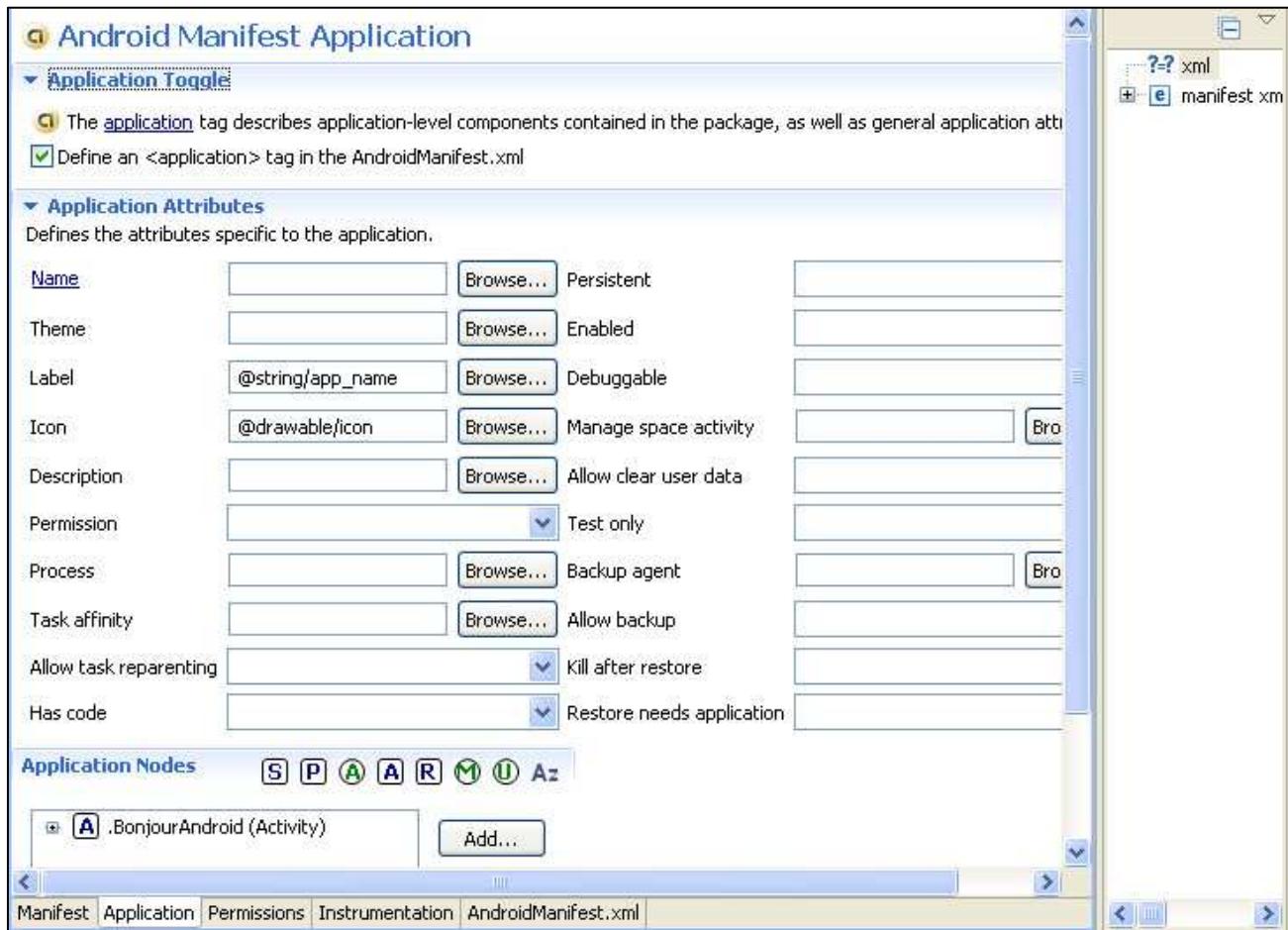


9.3 Le manifeste de l'application

Toute application Android doit posséder un fichier **AndroidManifest.xml** dans son répertoire racine. Son rôle est bien sûr de décrire l'application en termes de packages, classes, versions, permissions, etc. Mais sa première raison d'être est de déclarer les composants utilisables par le système : les composants qui n'y sont pas déclarés sont invisibles au système et ne pourront donc pas être exécutés. Dans notre cas, le fichier manifeste est simplement :

```
AndroidManifest.xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="applics.basics"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".HelloBonjour"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-sdk android:minSdkVersion="7" />
</manifest>
```

ce qui peut, encore une fois, se visualiser plus friendly avec une boîte à onglets (Open with Manifest Editor) :



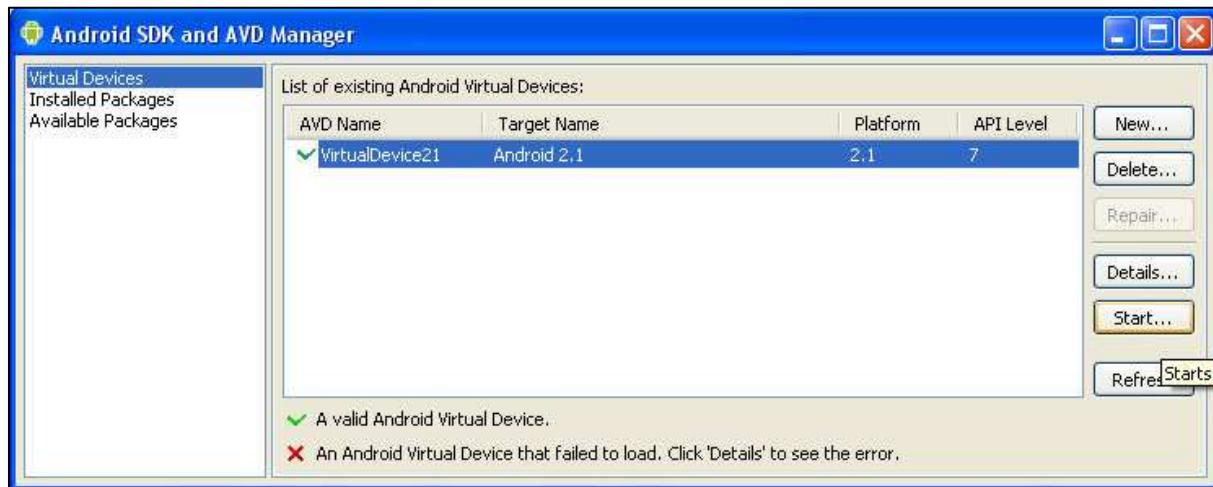
Seules les balises <manifest> et <application> sont requises. A remarquer que le nom de classe précédé d'un point ("HelloBonjour") est un raccourci pour signifier que ce nom doit être préfixé du nom du package (ici, "applics.basics"). On remarquera aussi la définition de l'**intent** (pour rappel, un intent est un message asynchrone spécifiant une action et contenant l'URI de la donnée à manipuler) : il spécifie qu'il faut lancer (LAUNCHER) l'activité "HelloBonjour" en tant que composant initial (MAIN).

10. Exécuter une application sur un mobile virtuel

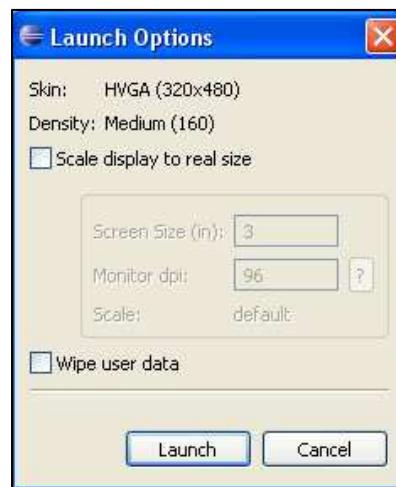
Quand on sollicite l'exécution au moyen de l'item Run du menu (en choisissant de l'exécuter comme une application Android), on obtient :

- ◆ soit le lancement de l'exécution de l'application;
- ◆ soit d'abord le manager d'AVD si on n'a jamais défini un AVD - il nous est donc alors demandé de le faire comme décrit ci-dessus :

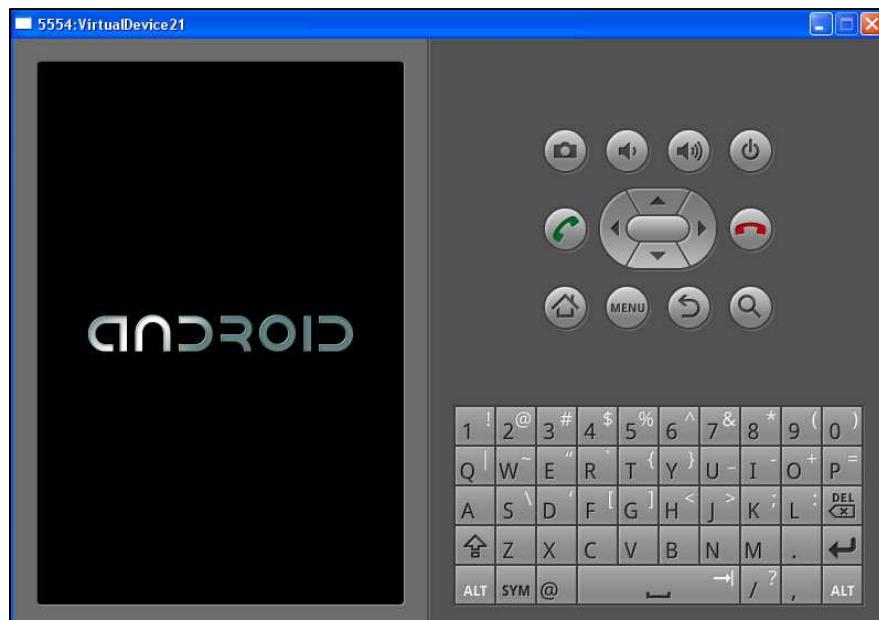




L'appui sur le bouton Start donnera :



et enfin le même résultat que si on avait défini l'AVD au préalable, c'est-à-dire d'abord :



puis, après un certain temps :



- pour lancer l'application, il peut être nécessaire d'appuyer sur le bouton Menu de l'interface



pour obtenir finalement :



A remarquer que l'appui sur le bouton Home-Maison rend l'interface obtenu au démarrage d'Android. A remarquer aussi le verbiage dans la console :

```
[2010-04-28 17:57:04 - BonjourAndroid]-----
[2010-04-28 17:57:04 - BonjourAndroid]Android Launch!
[2010-04-28 17:57:04 - BonjourAndroid]adb is running normally.
[2010-04-28 17:57:04 - BonjourAndroid]Performing aplics.basics.HelloBonjour activity
launch
[2010-04-28 17:57:04 - BonjourAndroid]Automatic Target Mode: launching new emulator
with compatible AVD 'VirtualDevice21'
[2010-04-28 17:57:04 - BonjourAndroid]Launching a new emulator with Virtual Device
'VirtualDevice21'
[2010-04-28 17:57:06 - BonjourAndroid]New emulator found: emulator-5554
[2010-04-28 17:57:06 - BonjourAndroid]Waiting for HOME ('android.process.acore') to be
launched...
[2010-04-28 17:57:35 - BonjourAndroid]HOME is up on device 'emulator-5554'
[2010-04-28 17:57:35 - BonjourAndroid]Uploading BonjourAndroid.apk onto device
'emulator-5554'
[2010-04-28 17:57:35 - BonjourAndroid]Installing BonjourAndroid.apk...
[2010-04-28 17:57:49 - BonjourAndroid]Success!
[2010-04-28 17:57:49 - BonjourAndroid]Starting activity aplics.basics.HelloBonjour on
device
[2010-04-28 17:57:56 - BonjourAndroid]ActivityManager: Starting: Intent {
    cmp=aplics.basics/.HelloBonjour }
```

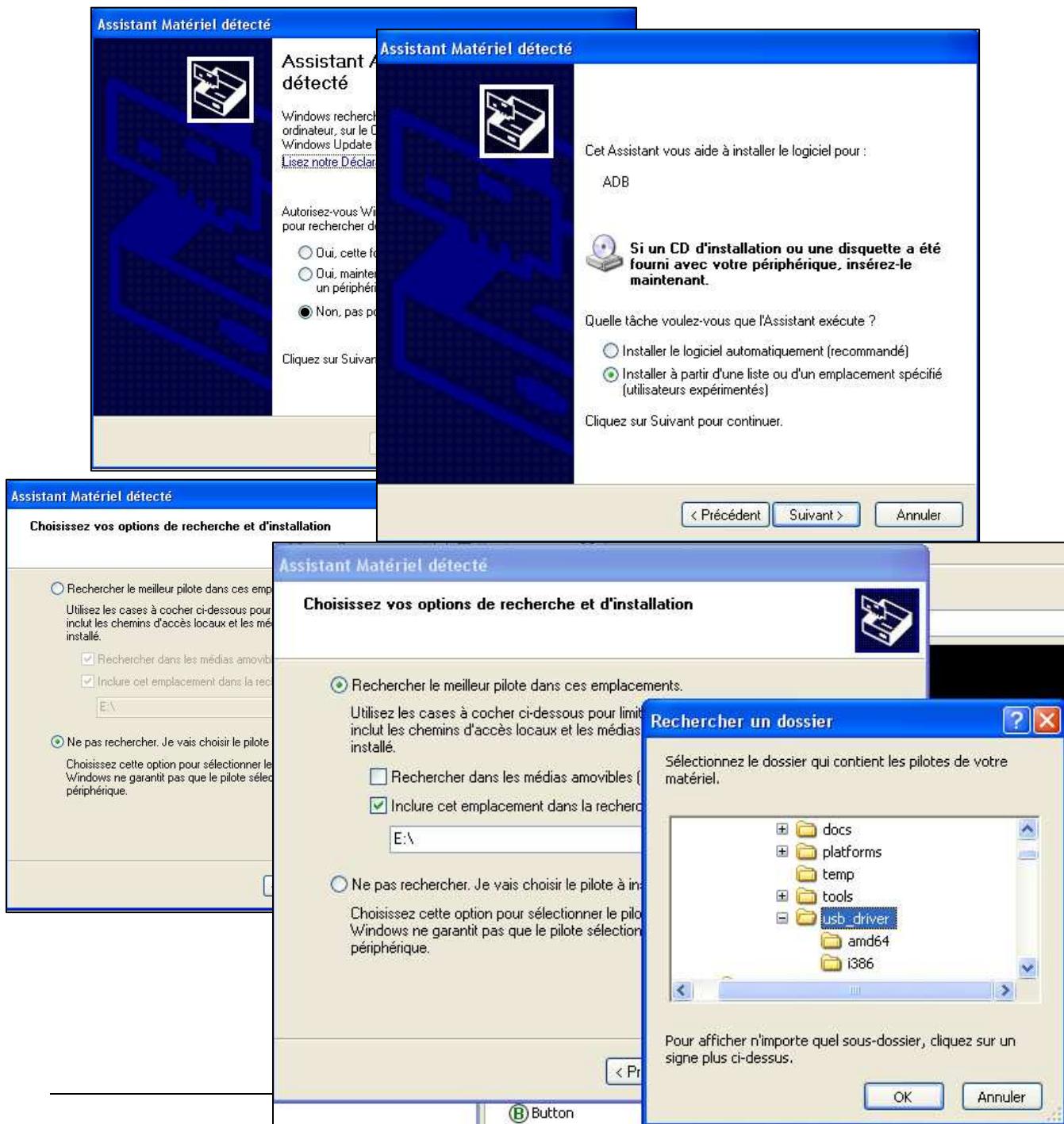
On remarquera les opérations de "téléchargement" et d'"installation" sur le mobile virtuel – mais précisément, comment faire pour "vrai" mobile ?

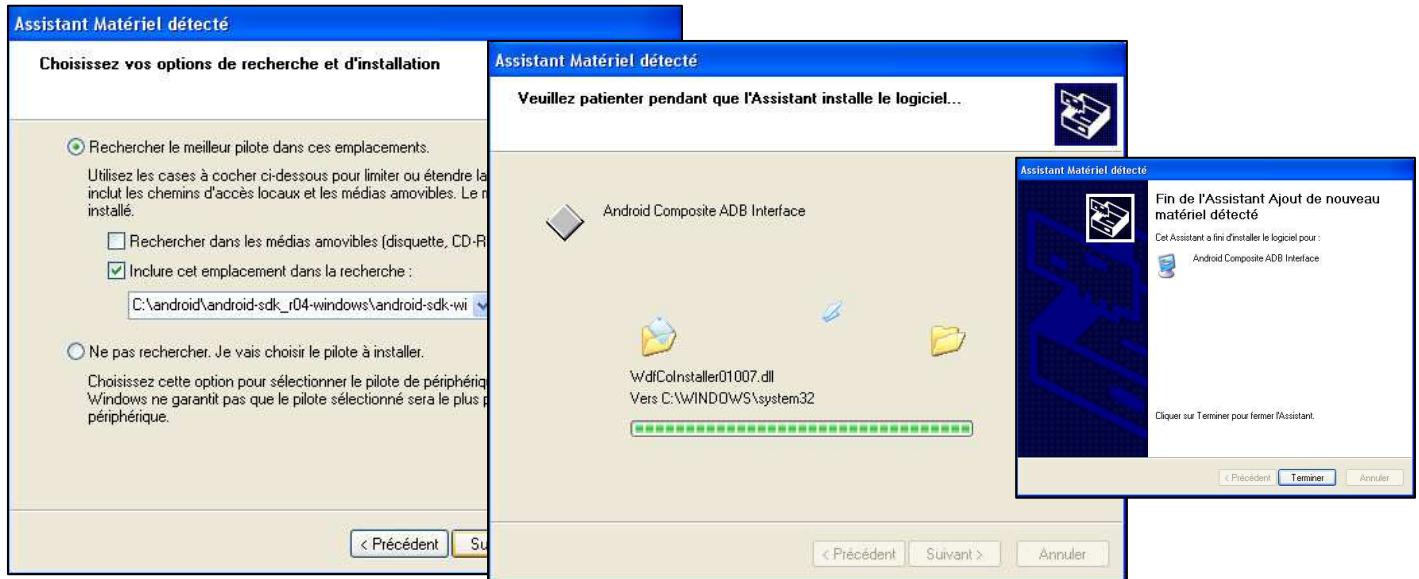
11. Installation d'une application sur un mobile

Nous allons à présent utiliser un véritable mobile qui possède le framework Android, en l'occurrence un mobile de marque HTC Hero utilisant un Google Android 1.5. En développant avec Eclipse, deux possibilités de portage s'offrent à nous : le déploiement automatique et le déploiement manuel. Mais auparavant, il faut d'abord que notre mobile soit connu de notre ordinateur de développement.

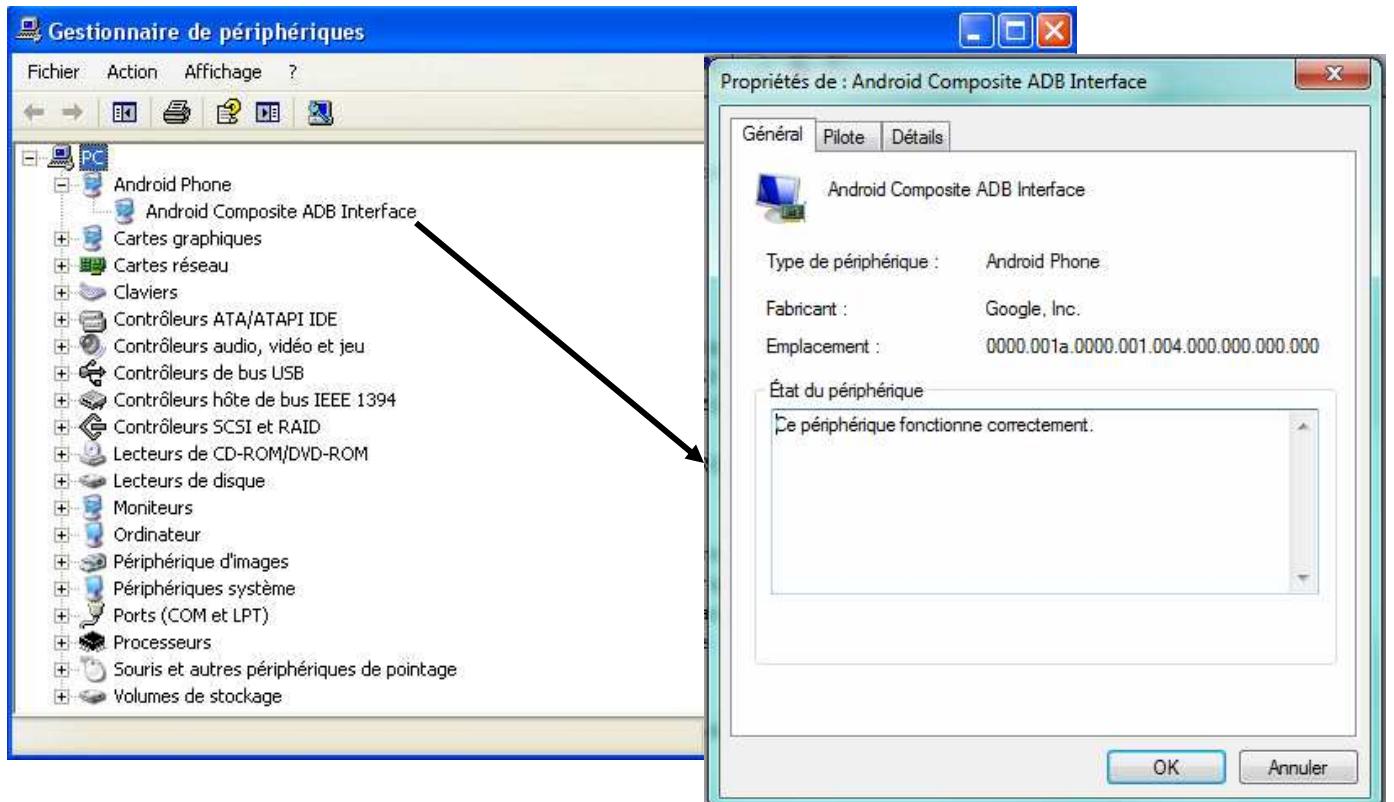
11.1 Installation des drivers du mobile Android

Après avoir connecté le mobile sur un port USB, il est reconnu comme un périphérique dont il faut installer les drivers : ceux-ci sont disponibles dans le répertoire du SDK nommé "usb_driver", donc pour nous C:\android\android-sdk_r06-windows\android-sdk-windows\usb_driver :



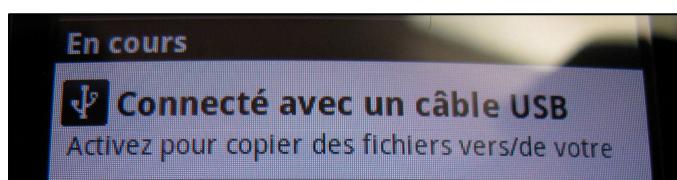


On peut vérifier par le panneau de configuration (noeud Système) que tout est en ordre :



11.2 Déploiement automatique de l'application depuis Eclipse

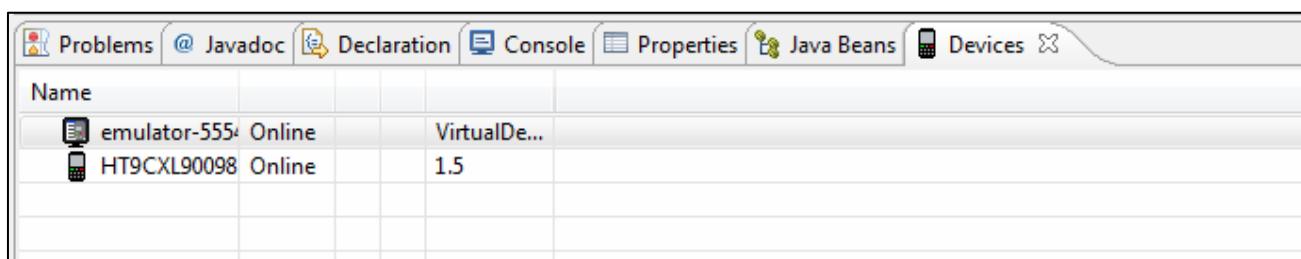
Une fois notre mobile connecté - il doit afficher "Connecté avec un câble USB" :



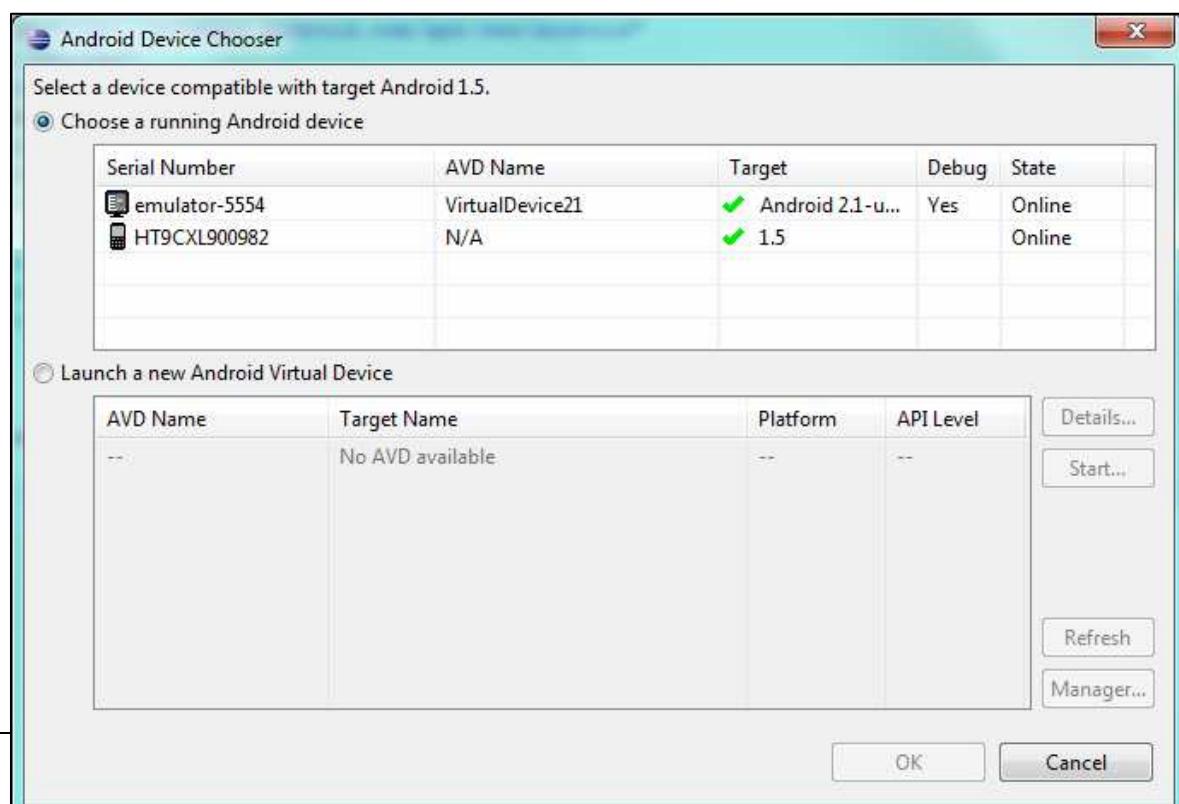
Eclipse va le reconnaître comme un "device Android". On peut le vérifier en faisant apparaître depuis le menu d'Eclipse Window → Show view → Other ... :



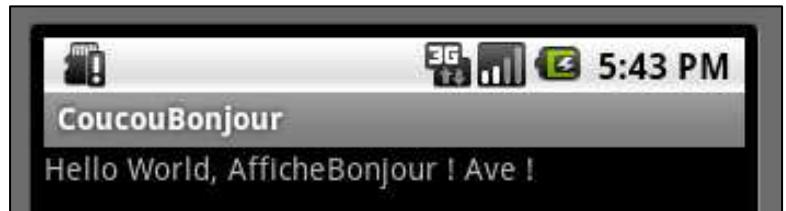
On obtiendra ainsi une fenêtre additionnelle dans laquelle le mobile virtuel et le mobile physique apparaissent tous deux :



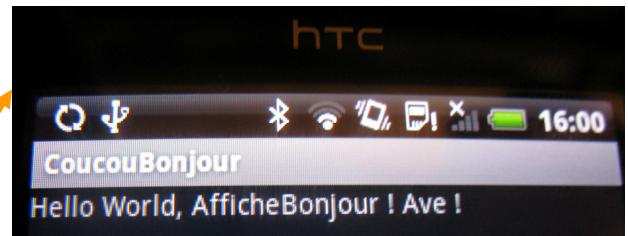
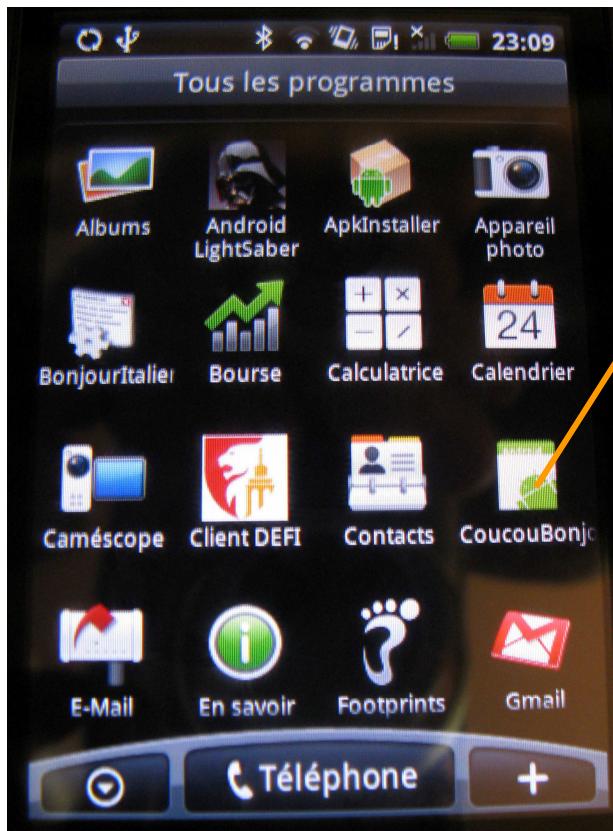
En exécutant le projet, on peut choisir le mobile hôte :



Le choix de l'émulateur donne :



tandis que celui du mobile HTC donne sur celui-ci :



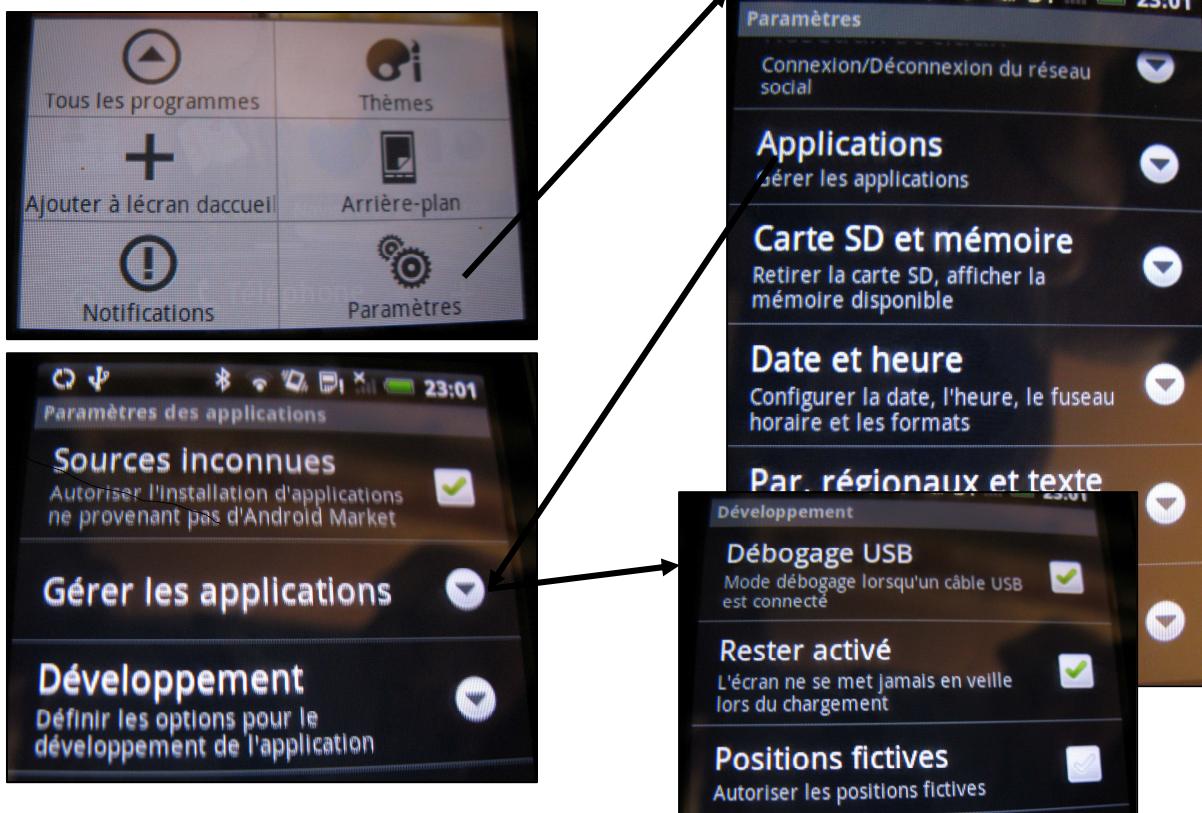
Eh oui, l'application a été déployée et exécutée automatiquement sur le mobile ☺ (mais pas installée) !

11.3 L'Android Debug Bridge (ADB)

Au cours des multiples essais du développeur qui explore le monde Android, une situation classique de blocage est celle de multiples déploiements de la même application avec des signatures (automatiques ou non) différentes. Le syndrome se manifeste avec des messages Android du type suivant :

```
[2010-08-26 22:37:38 - Coucou]Android Launch!
[2010-08-26 22:37:38 - Coucou]adb is running normally.
[2010-08-26 22:37:38 - Coucou]Performing apps.essais.AfficheBonjour activity launch
[2010-08-26 22:37:38 - Coucou]Automatic Target Mode: Several compatible targets. Please
select a target device.
[2010-08-26 22:37:42 - Coucou]Uploading Coucou.apk onto device 'HT9CXL900982'
[2010-08-26 22:37:42 - Coucou]Installing Coucou.apk...
[2010-08-26 22:37:43 - Coucou]Re-installation failed due to different application
signatures.
[2010-08-26 22:37:43 - Coucou]You must perform a full uninstall of the application.
WARNING: This will remove the application data!
[2010-08-26 22:37:43 - Coucou]Please execute 'adb uninstall apps.essais' in a shell.
[2010-08-26 22:37:43 - Coucou]Launch canceled!
```

Il faut donc interagir sur le mobile depuis l'ordinateur de développement. Ceci peut se faire avec un outil du SDK d'Android, l'**ADB** (Android Debug Bridge). Il faut cependant que le mobile soit en état de recevoir les commandes de cet outil, ce dont on peut s'assurer en vérifiant que le mode "USB debugging est activé" :



Dans ce cas, on peut alors utiliser cet outil adb depuis une fenêtre de commande DOS de l'ordinateur de développement :

```
C:\android>cd android-sdk_r06-windows  
C:\android\android-sdk_r06-windows>cd android-sdk-windows  
C:\android\android-sdk_r06-windows\android-sdk-windows>cd tools  
C:\android\android-sdk_r06-windows\android-sdk-windows\tools>adb shell  
error: more than one device and emulator
```

Le problème vient ici du fait que, outre le mobile HTC, l'émulateur est également actif :

```
C:\android\android-sdk_r06-windows\android-sdk-windows\tools>adb devices  
List of devices attached  
emulator-5554 device  
HT9CXL900982 device
```

Nous pouvons donc envisager de 'nettoyer' le répertoire des packages du package qui pose des problèmes de signature :

```
C:\android\android-sdk_r06-windows\android-sdk-windows\tools>adb uninstall  
applics.essais  
adb server is out of date. killing...  
* daemon started successfully *  
- waiting for device -  
error: more than one device and emulator  
...  
- waiting for device -  
Success
```

- l'aboutissement de nos commandes étant du au fait que l'émulateur a été arrêté.

L'outil adb permet en fait de dialoguer avec le noyau linux au moyen d'un shell :

```
C:\android\android-sdk_r06-windows\android-sdk-windows\tools>adb shell  
$ ls  
ls  
sqlite_stmt_journals  
cache  
sdcards  
etc  
system  
sys  
sbin  
proc  
logo.rle  
init.rc  
init.hero.rc  
init.goldfish.rc  
init  
default.prop
```

```
data
root
dev

$ cd app
cd app

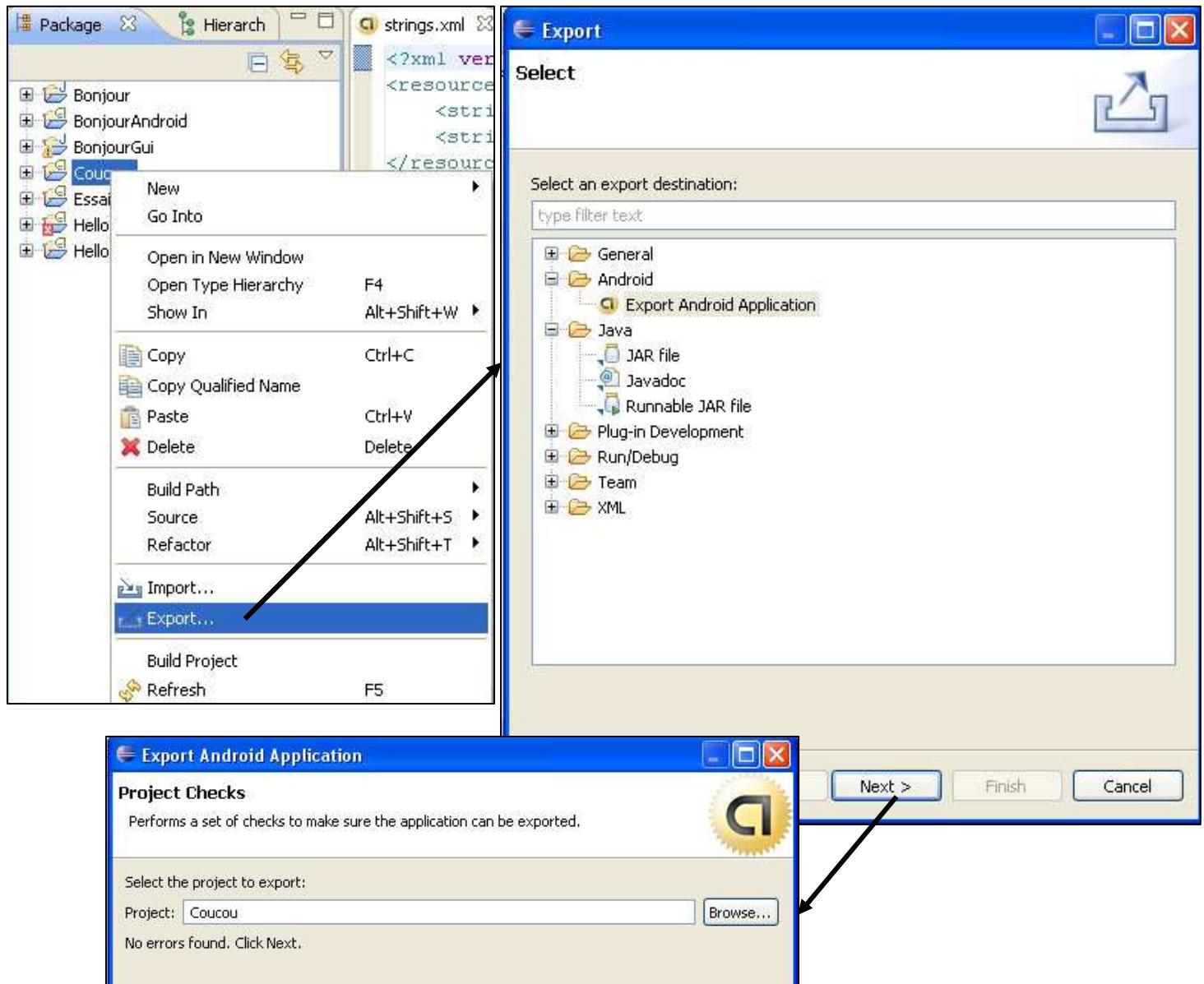
$ ls
ls
PDFViewer.apk
DownloadProvider.odex
DCSUtility.apk
HTCSetupWizard.apk
FilePicker.odex
CertificateService.odex
...
UploadProvider.apk
GoogleSearch.odex
HtcSettingsProvider.apk
CustomizationSettingsProvider.odex
...
WorldClock.apk
WeatherProvider.apk
...
PackageInstaller.odex
Maps.apk
...
WeatherProvider.odex
...
PDFViewer.odex
...
Weather.odex
Launcher.apk
...
HTMLViewer.apk
Browser.odex
...
$ exit
C:\android\android-sdk_r06-windows\android-sdk-windows\tools>
```

Les fichiers 'odex' sont en fait des fichiers 'dex' optimisés (**Optimized DEX files**). L'objectif est évidemment que les fichiers apk qui les utilisent sont plus rapides – mais ils dépendent alors de la présence de ces fichiers optimisés.

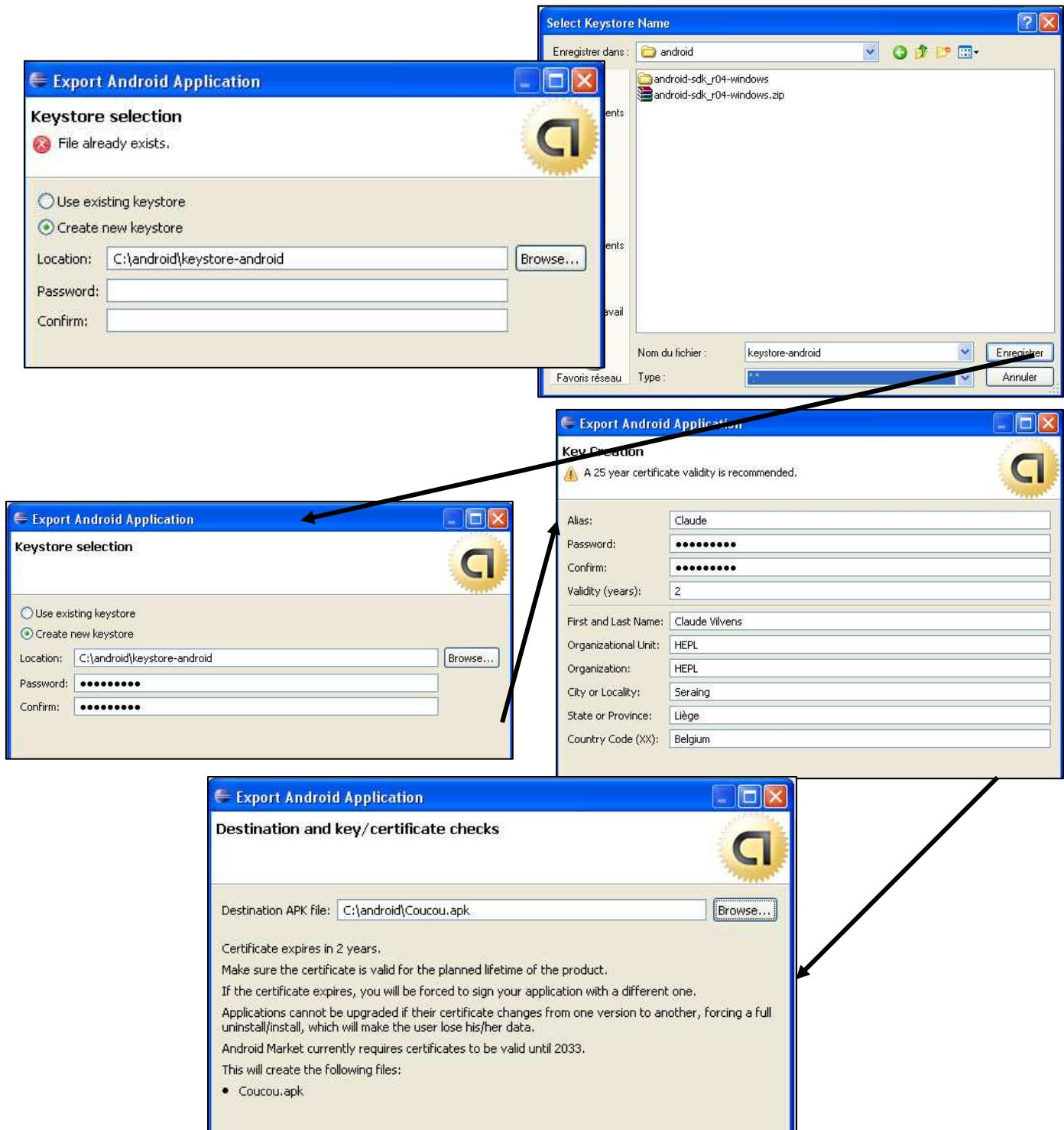
11.4 Déploiement manuel de l'application

On se souviendra que le framework Android ne fait fonctionner que des applications apk. Nous allons donc tout d'abord, depuis Eclipse, générer de fichier apk sur l'ordinateur de développement pour ensuite, bien logiquement, le copier et l'installer sur le mobile.

La création de l'apk se réalise par le choix Export dans le menu contextuel sur le projet :



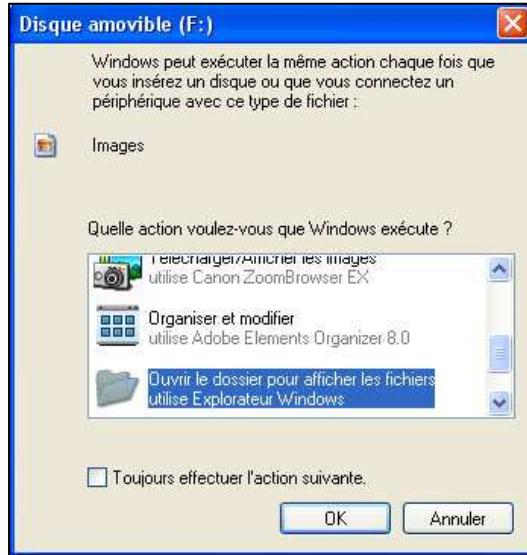
Mais voici une petite surprise : *Android nous demande de signer l'apk* que nous voulons construire (encore qu'un mobile Android puisse faire fonctionner une application d'origine inconnue, pour peu que l'utilisateur le permette) en utilisant une clé se trouvent dans un keystore, existant ou que nous pouvons créer pour l'occasion, option que nous choisissons ici :



Nous nous retrouvons donc avec :

Nom	Taille	Type	Date de modification
android-sdk_r04-windows		Dossier de fichiers	28/02/2010 18:09
android-sdk_r04-windows.zip	22.529 Ko	WinRAR ZIP archive	25/02/2010 13:03
Coucou.apk	8 Ko	Fichier APK	26/08/2010 15:47
keystore-android	2 Ko	Fichier	26/08/2010 15:47

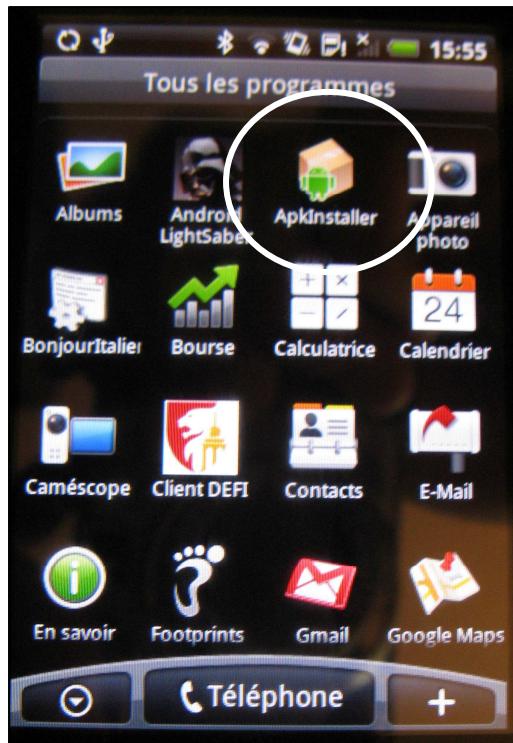
Nous pouvons à présent basculer notre mobile en mode "**Périphérique de stockage**", ce qui aura un effet immédiat :



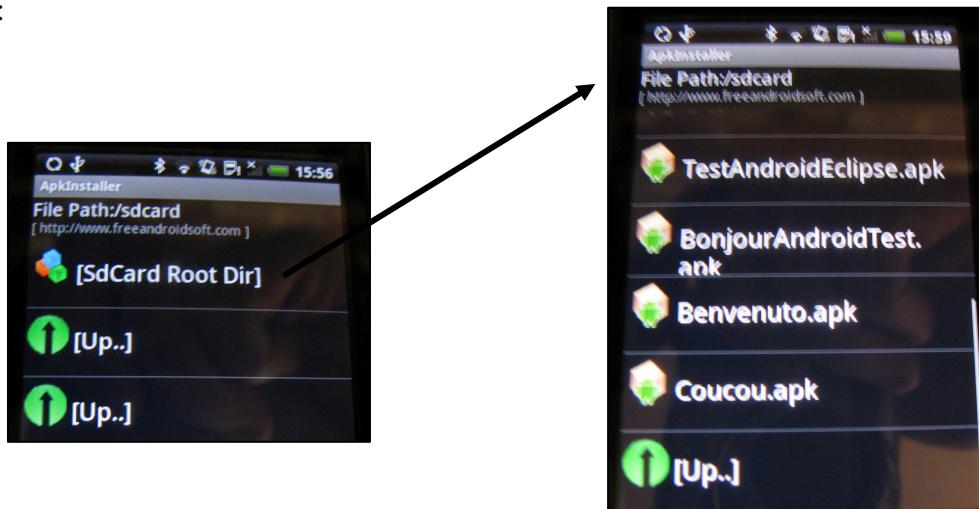
Il suffit dès lors de *copier notre fichier apk dans un répertoire de la carte SD* de notre mobile :

.footprints	Dossier de fich...	29/04/2010 9:23
DCIM	Dossier de fich...	21/05/2010 10:14
download	Dossier de fich...	11/05/2010 15:10
rosie_scroll	Dossier de fich...	29/04/2010 9:44
Benvenuto.apk	14 Ko Fichier APK	25/08/2010 11:12
BonjourAndroidTest.apk	13 Ko Fichier APK	22/08/2010 14:13
boot.img	1 750 Ko Fichier IMG	15/08/2009 23:34
cm-hero-recovery.img	Type : Fichier APK	IMG 29/04/2010 11:16

Mais ce n'est pas fini : il faut encore installer l'application Coucou.apk. Pour cela, nous faisons appel à l'application très logiquement appelée "ApkInstaller" :



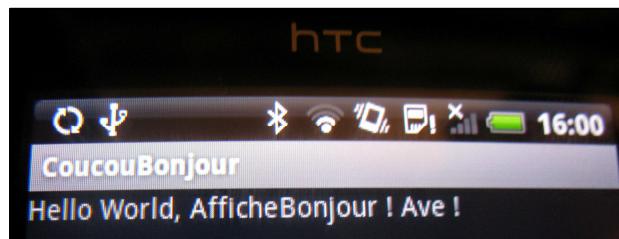
On se retrouve ainsi à choisir l'application à installer parmi celles qui se trouvent sur la carte SD :



Nous choisissons donc notre application "Coucou.apk" :



Et donc finalement ☺ :

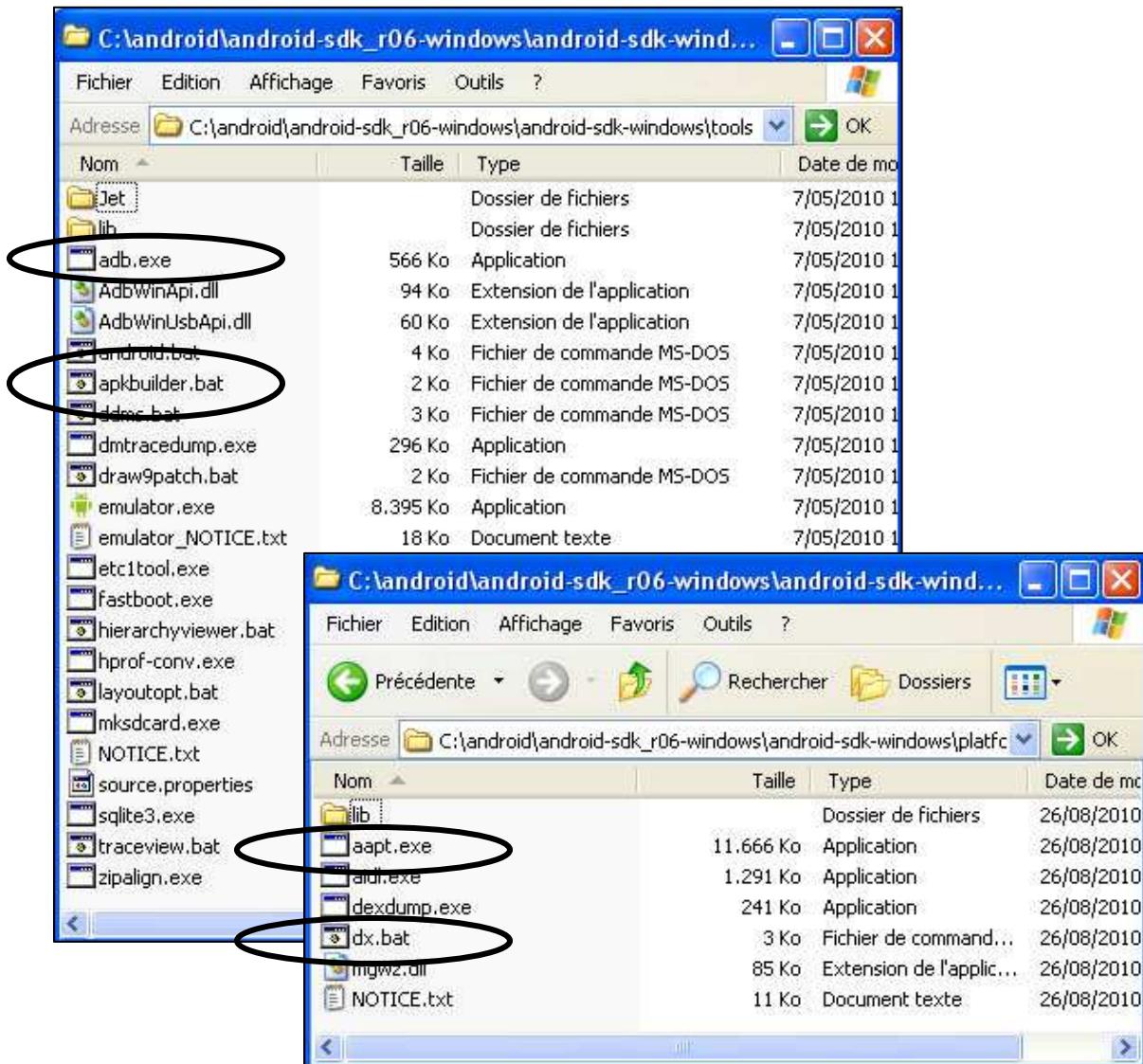


12. Le développement manuel d'une application Android

Bien qu'il soit peu probable que l'on soit amené à développer une application Android entièrement manuellement depuis la ligne de commande (encore qu'il semblerait que ce soit parfois nécessaire pour des applications manipulant de grosses ressources qui posent problème à Eclipse), il n'est pas intéressant de résumer les étapes successives d'un tel développement – ne serait-ce que pour voir à quoi on échappe ;-) mais aussi surtout pour bien appréhender le processus de développement qu'Eclipse occulte en partie.

On utilise pour ce travail de titan des outils qui se trouvent dans deux répertoires du SDK :

```
C:\java-eclipse-application\Coucou\src\aplics\essais>set PATH=\\android\\android-sdk_r06-windows\\android-sdk-windows\\platforms\\android-5\\tools;\\android\\android-sdk_r06-windows\\android-sdk-windows\\tools;%PATH%
```



Le mode opératoire type est le suivant.

1) Le premier outil à invoquer est **aapt** : son rôle est de *construire le fichier R* qui décrit les ressources. Ceci sous-entend que

- ◆ on a rédigé un fichier **AndroidManifest.xml**;
- ◆ on a créé un répertoire de ressources (typiquement nommé "res") structuré en les sous-répertoires qu'Eclipse nous montre dans les projets Android : par exemple **layout** et **values** pour contenir les fichiers descriptifs xml (que l'on aura généré en utilisant les templates qui se trouvent dans android-sdk-windows\platforms\android-X\templates) et **drawable** pour une icône png, le tout empaqueté dans un fichier au format zip.

La syntaxe de la ligne de commande est alors du type :

```
aapt package -f -M <fichier manifeste> -F <fichier empaquetant les ressources> -I <path pour les jar d'Android> -S <répertoire des ressources d'Android> [-m -J <répertoire où placer le R.java produit>]
```

2) On peut ensuite compiler le source .java et le fichier R.java obtenu à l'étape précédente au moyen du compilateur javac du JDK :

javac *.java

3) On génère ensuite le bytecode modifié avec l'utilitaire **dx** :

dx --dex --output=<nom du fichier dex produit> <répertoire des fichiers .class à traiter> <fichiers jar à utiliser>

4) On peut enfin créer le fichier apk avec l'utilitaire shell **apkbuilder** :

apkbuilder <nom du fichier apk produit> -u -z <fichier des ressources> -f <fichier dex>

On produit ainsi un fichier apk non signé (c'est la signification du commutateur –u).

5) A priori, on ne distribue que des applications signées – on usera donc du keytool pour créer ou gérer un keystore :

keytool -list <keystore utilisé>

6) Il reste à signer l'apk (qui, rappelons-le, a un format zip/jar) au moyen de l'utilitaire jarsigner :

jarsigner -keystore <nom du keystore> -storepass <mot de passe du keystore> -keypass <mot de passe de l'alias utilisé> -signedjar <nom du fichier signé à produire> <nom du fichier à signer> <alias dans le keystore>

7) Il reste à installer l'application au moyen de l'outil adb (Android Debug Bridge) :

adb -d install -r <fichier apk>

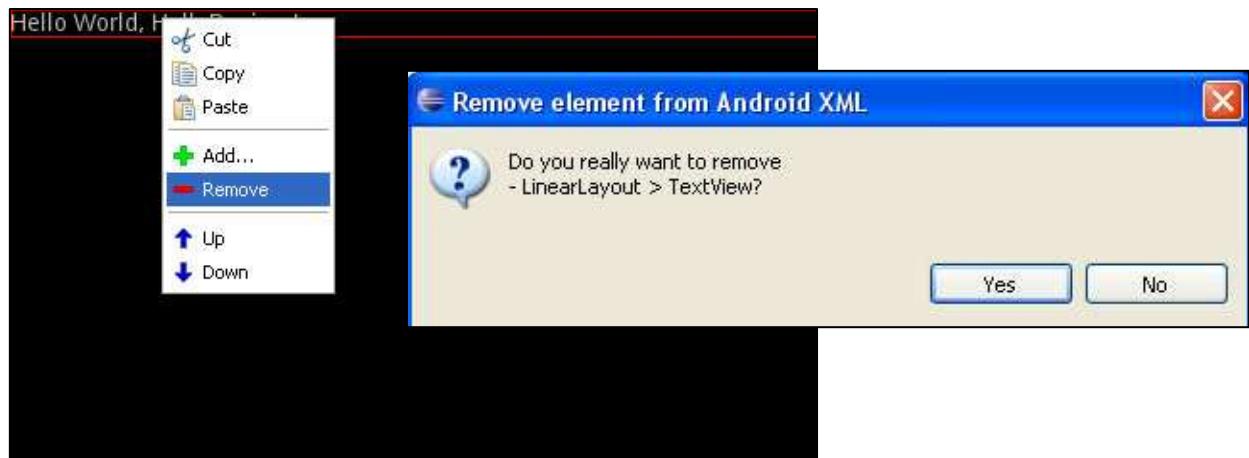
8) Accessoirement, on peut visualiser le contenu du fichier apk avec l'outil aapt :

```
C:\java-android-application\Coucou\bin>aapt list -v Coucou.apk
Archive: Coucou.apk
Length Method  Size Ratio Date Time      CRC-32 Name
----- -----  -----  ---  ---  -----
 2200 Stored    2200  0% 08-26-10 20:34 99a4f90b res/drawable/icon.png
   640 Deflate    277  57% 08-26-10 20:34 8ae7db9b res/layout/main.xml
 1424 Deflate    555  61% 08-26-10 20:34 17fb42dc AndroidManifest.xml
 1060 Stored    1060  0% 08-26-10 20:34 4e2f08e7 resources.arsc
 1900 Deflate    932  51% 08-26-10 20:34 ad5379e5 classes.dex
  401 Deflate    267  33% 08-26-10 20:34 e7e2db67 META-INF/MANIFEST.MF
  454 Deflate    299  34% 08-26-10 20:34 bf3ab414 META-INF/CERT.SF
  776 Deflate    603  22% 08-26-10 20:34 8d19d2aa META-INF/CERT.RSA
-----  -----  ---  -----
 8855          6193 30%                               8 files
```

Ben dis donc :-o ... Revenons à notre développement avec Eclipse ...

13. Une application avec vue explicite

Nous allons à présent définir la view de notre application plus explicitement que par un identifiant venu de nulle part. Tout d'abord, enlevons le composant "Hello world" :



Effectivement :

main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
</LinearLayout>
```

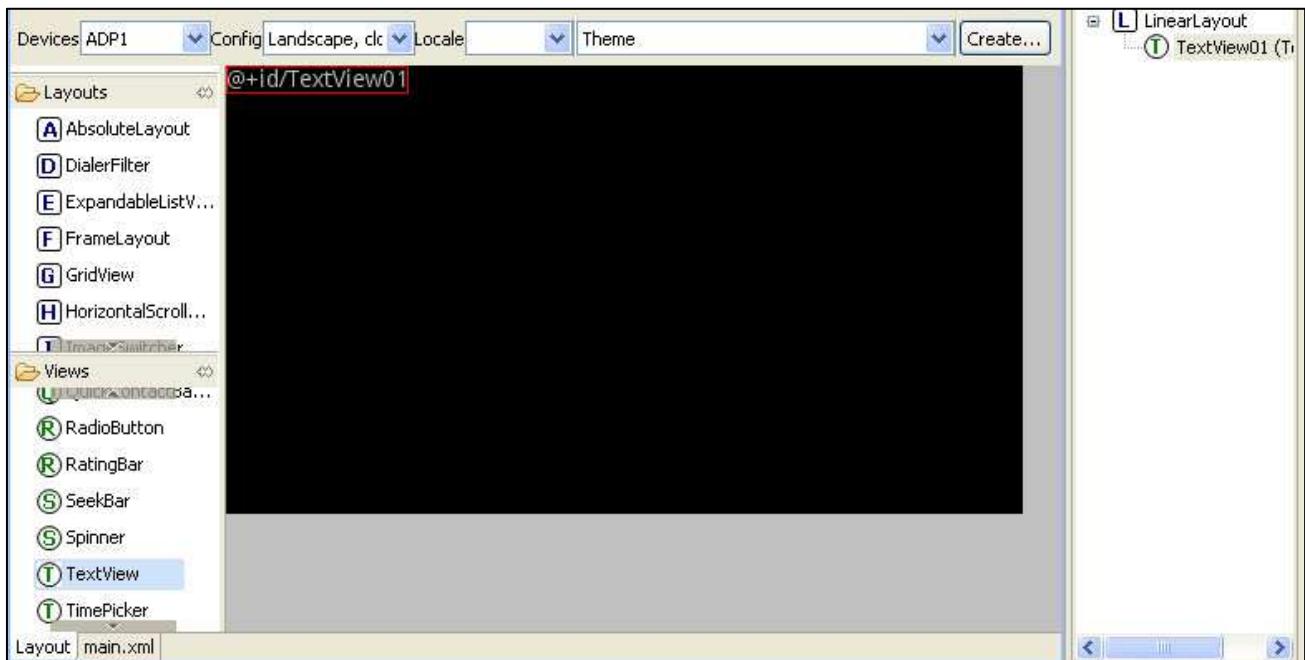
mais

R.java

```
package applics.basics;

public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
    }
}
```

Nous allons à présent ajouter un TextView : il suffit de sélectionner cette classe dans la liste des Views et de la faire glisser dans la fenêtre :



La liste des propriétés située en dessous nous permet, par exemple, de

- ◆ visualiser l'identifiant généré :

Property	Value
Height	
Hint	
Id	@+id/TextView01
Ime action id	
Ime action label	
Ime options	
Include font padding	
Input type	

- ◆ fixer le texte :

Property	Value
Style	
Tag	
Text: Text to display. [String] Text color	@+id/TextView01
Text color highlight	
Text color hint	
Text color link	

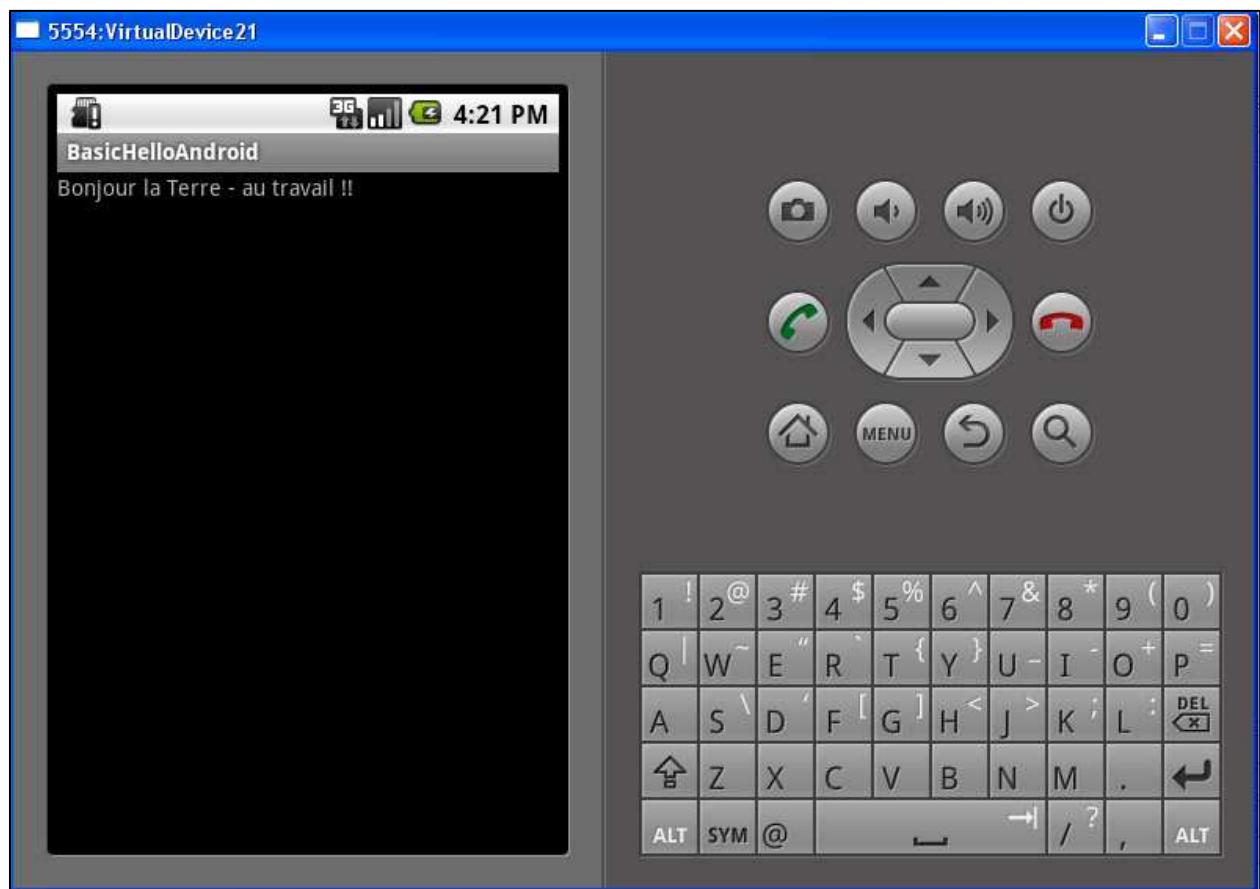
Ce qui nous donne :

main.xml (2)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
```

```
    android:layout_height="fill_parent"
    >
<TextView android:id="@+id/TextView01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Bonjour la Terre - au travail !!">
</TextView>
</LinearLayout>
```

Ce qui donne, après rafraîchissement (F5) et ré-exécution :



Après avoir encore ajouté un EditText et un Button (il s'agit de deux classes dérivées de TextView au sein du package android.widget), on obtient à l'exécution :



Notre classe R se présente à présent comme ceci :

```
R.java (2)
package applics.mobiles;

public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class id {
        public static final int Button01=0x7f050002;
        public static final int EditText01=0x7f050001;
        public static final int TextView01=0x7f050000;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
    }
}
```

Reste encore à traiter l'appui sur le bouton "Se connecter". Le plus simple est d'implémenter une logique événementielle très similaire à celle que l'on connaît dans les GUIs classiques AWT et Swing : en fait, tous les descendants de la classe View possèdent la méthode

```
public void setOnClickListener (View.OnClickListener l)
```

l'interface (imbriqué à View) **OnClickListener** ne déclarant que la seule méthode :

```
public abstract void onClick (View v)
```

Mais, pour s'enregistrer comme listener du bouton, il faut d'abord que notre activité retrouve la référence de ce contrôle, ce qu'elle va faire par la méthode :

```
public View findViewById (int id)
```

Donc, finalement :

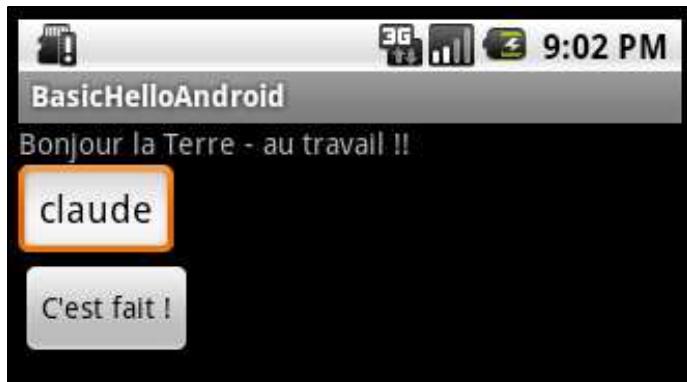
```
BonjourAndroid.java (2)
package applics.mobiles;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
```

```
import android.widget.Button;

public class BonjourAndroid extends Activity implements OnClickListener
{
    Button BLogin;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        BLogin = (Button)this.findViewById(R.id.Button01);
        BLogin.setOnClickListener(this);
    }
    @Override
    public void onClick(View arg0)
    {
        // TODO Auto-generated method stub
        BLogin.setText("C'est fait !");
    }
}
```

Résultat après l'appui sur le bouton :



Mouais ... Et si on se connectait pour de vrai ?

14. Les communications réseaux

Les communications réseaux sous Android nous sont bien connues ! En effet, elles utilisent les classes sockets de la programmation réseau habituelle sous Java, comme la classe Socket par exemple. Une petite condition pour que notre application puisque accéder au réseau : il faut ajouter dans le fichier manifeste une clause autorisant cet accès, ce qui se fait au moyen de

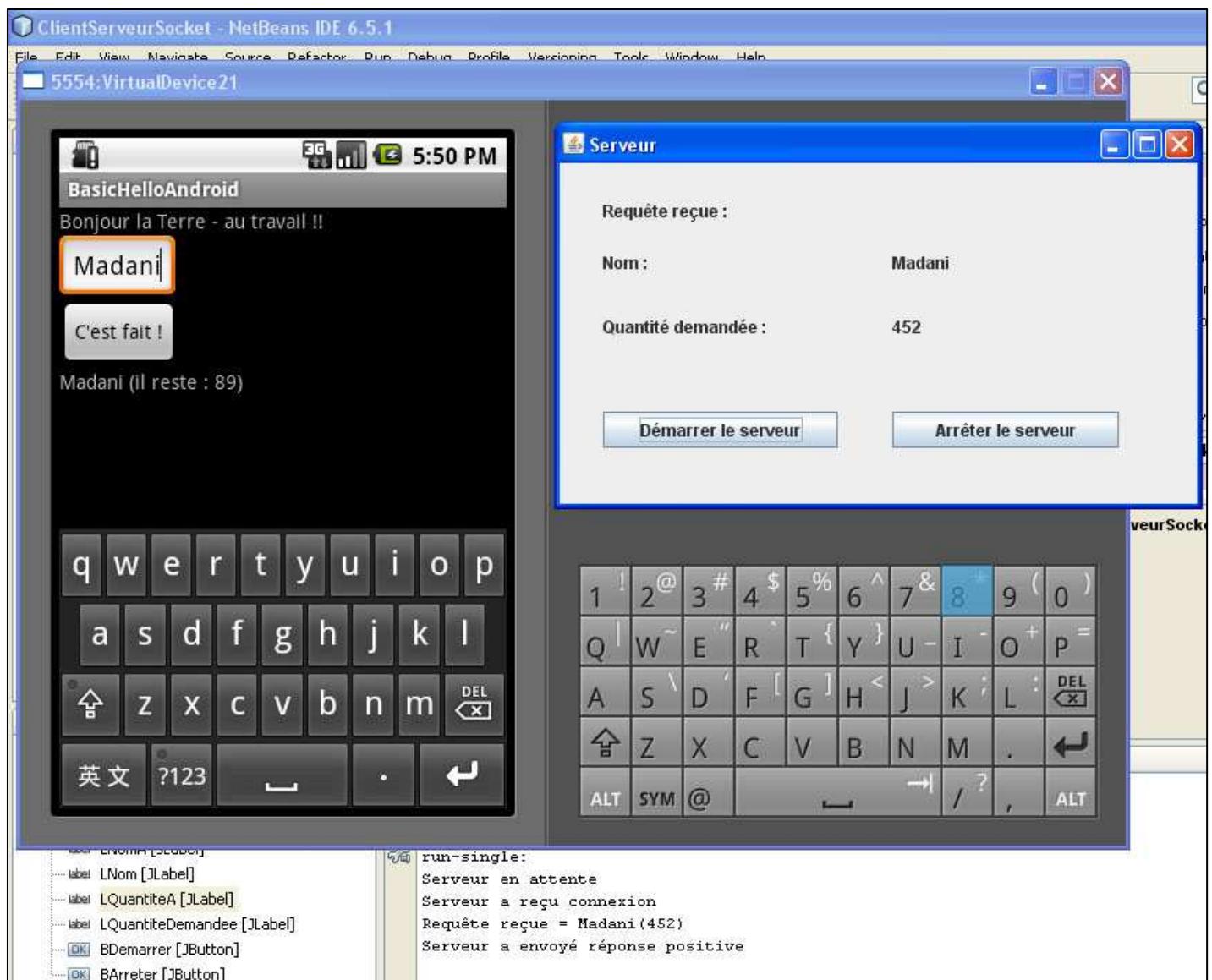
```
<uses-permission android:name="android.permission.INTERNET"></uses-permission>
```

Donc :

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="applies.basics"
    ...
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".HelloBonjour"
            android:label="@string/app_name">
            ...
        </activity>
    </application>
    ...
    <uses-permission android:name="android.permission.INTERNET"></uses-
permission>
</manifest>
```

On peut donc imaginer modifier notre application pour qu'elle puisse atteindre un serveur réseau tout à fait conventionnel (il s'agit d'un serveur FenServeurSocket dont l'air devrait dire quelque chose au lecteur qui a lu Java II ;-)) :



BonjourAndroid.java (3)

```
package applics.mobiles;

import java.io.*;
import java.net.*;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class BonjourAndroid extends Activity implements OnClickListener
{
    Button BLogin;
    EditText ETLogin;
    TextView TVReponse;

    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        BLogin = (Button)this.findViewById(R.id.Button01);
        BLogin.setOnClickListener(this);
        ETLogin = (EditText)this.findViewById(R.id.EditText01);
        TVReponse = (TextView)this.findViewById(R.id.TextView02);
    }
    @Override
    public void onClick(View arg0)
    {
        String nom = (String) ETLogin.getText().toString();
        contacteServeur(nom);
        BLogin.setText("C'est fait !");
    }

    private void contacteServeur(String n)
    {
        Socket cliSock=null;
        DataInputStream dis = null; DataOutputStream dos = null;
        try
        {
            cliSock = new Socket(InetAddress.getByName("192.168.1.5"),
                50000);
        }
        catch (UnknownHostException e)
        { System.err.println("Erreur ! Host non trouvé [" + e + "]"); }
        catch (IOException e)
        { System.err.println("Erreur ! Pas de connexion ? [" + e + "]"); }
    }
}
```

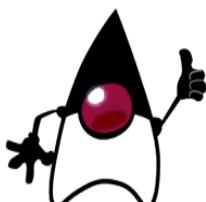
```
try
{
    dis = new DataInputStream(cliSock.getInputStream());
    dos = new DataOutputStream(cliSock.getOutputStream());
}
catch (IOException e){ }

String outNom = n;
int outQuantite = Integer.parseInt("452");
String reponse = null;
int inQuantiteRestante = 0;
if (!outNom.equals("") && outQuantite>0 && dos!=null && dis!=null)
{
    try
    {
        dos.writeUTF(outNom);dos.writeInt(outQuantite);
        reponse = dis.readUTF();
        inQuantiteRestante = dis.readInt();
        TVReponse.setText(reponse + " (il reste : " +
                           inQuantiteRestante+ ")");
    }
    catch (IOException e) { }
    finally
    {
        try { dis.close();dos.close(); } 
        catch (IOException e) { e.printStackTrace(); }
    }
}
}
```

Le serveur J2SE n'a donc absolument pas été modifié ☺ !

Bien sûr, il reste à entrer plus en profondeur dans tous les packages disponibles pour les interfaces graphiques. Mais cela dépasse une simple introduction – on utilisera avec bonheur la référence en ligne disponible sur

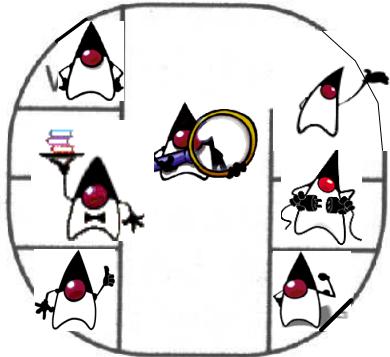
<http://developer.android.com/reference/packages.html>



Bien sûr, il y a encore à étudier dans le monde Android (l'utilisation de SQLite par exemple) et à découvrir dans le monde des mobiles (ainsi, nous pourrions aller visiter les Blackberrys ou les IPads).

Mais passons à présent aux petites sœurs des mobiles ...

XXIX. Les cartes à puce sous Java



Les bourgeois honnêtes ne comprennent pas que l'on puisse être honnête autrement qu'eux.

(A. Gide, Les Faux-Monnayeurs)

Qui ne connaît les cartes magnétiques, à moins de vivre dans une île déserte du Pacifique ou sur la banquise Antarctique ? Ces cartes font partie intégrante de la vie moderne, en tant que moyen d'authentification (badges d'entrée, carte de sécurité sociale, carte de session sur un ordinateur, carte SIM pour GSM) ou de paiement (Mr Cash, cartes de crédit de type Visa ou Mastercard). Les dernières années ont vu ces cartes se munir d'une puce électronique complémentaire, devant ainsi les "cartes à puce". Nous allons découvrir ici comment les gérer au moyen de Java :-o ...

1. Cartes magnétiques et cartes à puces

Les cartes les plus simples sont les **cartes magnétiques** : elles ne servent qu'à mémoriser des informations sur un circuit mémoire accédé par une piste magnétique, d'où leur nom anglo-saxon de "*magnetic stripe cards*".

Par opposition, les **cartes à puce** comportent un véritable processeur, dont le rôle est notamment de permettre ou pas l'accès aux informations mémorisées en effectuant des opérations relevant de la cryptographie. Ce sont plus particulièrement ces cartes qui sont désignées sous le vocable de "*smart cards*" (les termes de "*chips cards*" et d'"*integrated circuit cards*" [ICC] se rencontrent également). Leur succès a été rapide parce qu'elles apportent précisément une sécurité bien plus assurée que celle fournie par les cartes magnétiques sans processeur. En particulier, le secteur du *paiement électronique* s'est emparé de ces cartes pour deux raisons :

- ◆ une smart card est capable de réaliser des contrôles internes en utilisant les outils de la cryptographie : une telle carte est en fait très difficile à copier;
- ◆ une smart card permet de réaliser un **porte-monnaie électronique** (*e-purse* ou *e-wallet*) : la carte elle-même contient un crédit (avec une limite maximale assez peu élevée) qui peut être décrémenté ou incrémenté – le système Proton a popularisé cet usage.

Notons encore que, lorsque la carte ne comporte pas de microprocesseur mais seulement un dispositif d'accès sécurisé élémentaire (du type mot de passe), on parle encore plutôt de **carte mémoire** ou "*memory cards*".

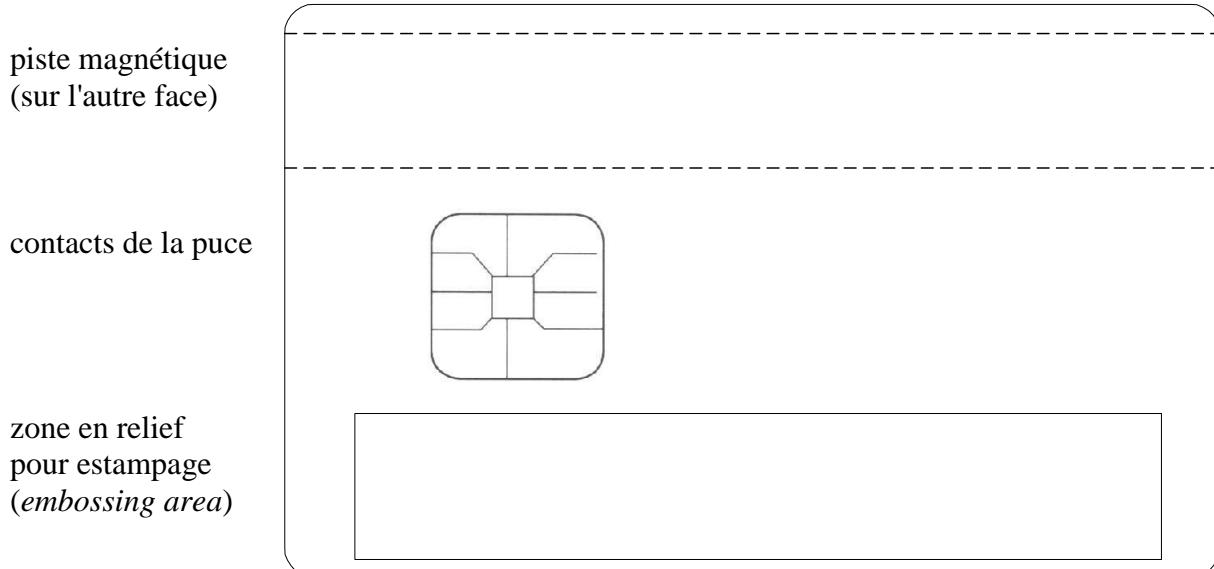
Enfin, les cartes **RFID** (Radio Frequency IDentification) ne possèdent aucun contact : une telle carte et son lecteur possèdent chacun une antenne et communiquent par ondes radio – on les appelle encore les "puces radio". Leur but est de stocker et de gérer des données à distance comme par exemple un stock dans un entrepôt.

2. L'anatomie matérielle et logicielle d'une smart card

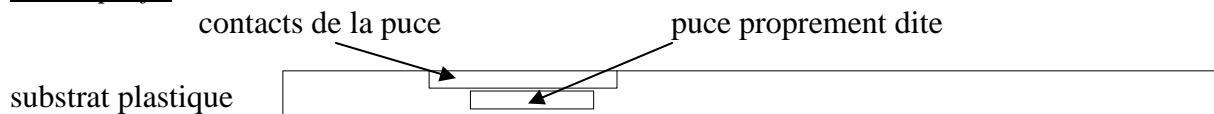
2.1 L'aspect général

Les cartes à puces existent en deux formats : 85.6x53.98x0.8 mm pour les plus courantes, 25x15 mm pour les cartes **SIM** (Subscriber Identity Modules) utilisées à des fins d'authentifications, notamment dans les GSM (Global System for Mobile communication). Schématiquement, les cartes à puce ont l'aspect suivant :

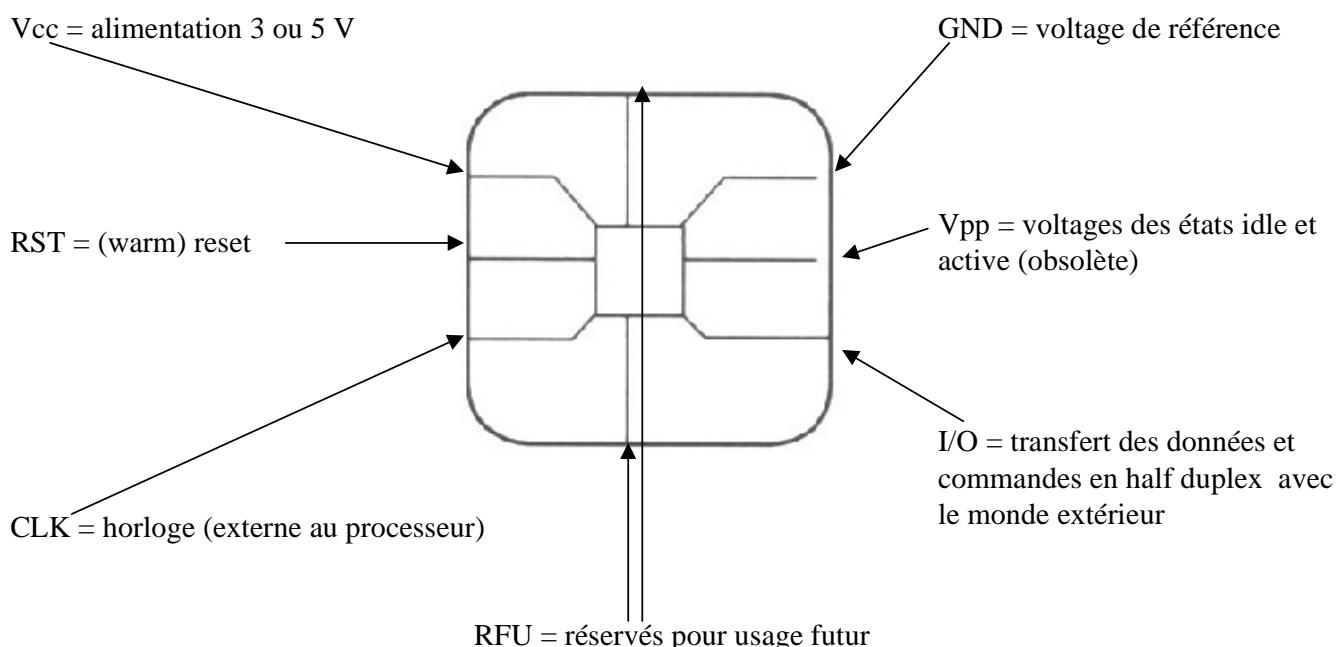
vue de face



vue de profil



Les contacts, au nombre de 8, correspondent aux fonctionnalités suivantes :



A remarquer qu'il existe aussi des cartes sans contacts (l'antenne y est incorporée), mais leur usage est moins répandu à l'heure actuelle.

2.2 Les composants du circuit intégré

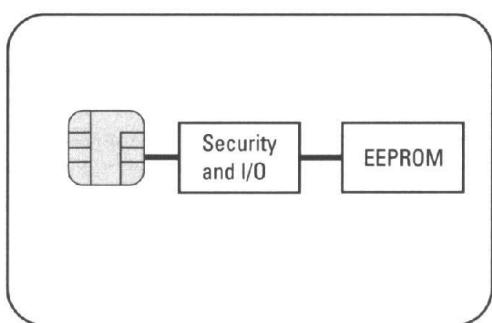
Une carte magnétique de type carte mémoire contient un circuit intégré fort simple, puisqu'il comporte simplement

- ◆ une mémoire : de type **EEPROM** (Electrical Erasable and Programmable Read-Only Memory), elle contient les données à caractère permanent associées à la carte;
- ◆ un canal d'entrée/sortie (**I/O**) qui permet la communication vers l'extérieur via la puce.

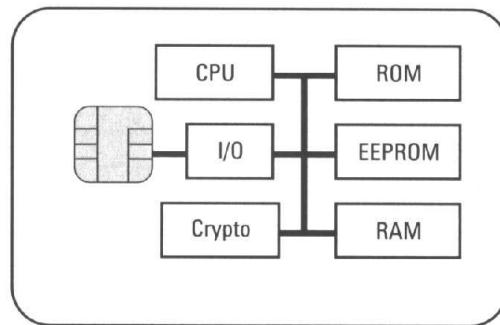
Par contre, à l'image d'un ordinateur, le circuit intégré se trouvant sur une smart card comporte :

- ◆ un microprocesseur (**CPU**), à priori 8 bits et 5 MHz (type Motorola 6805 ou Intel 8051) – mais on peut trouver beaucoup mieux;
- ◆ un **coprocesseur** destiné aux applications cryptographiques – celle-ci sont en effet souvent amenées à utiliser de grands nombres ou à réaliser des opérations longues (du type algorithme DES, génération de nombres aléatoire, arithmétique modulaire et RSA);
- ◆ trois mémoires : une **ROM**, une **RAM** et une **EEPROM** qui contient des données à caractère permanent mais modifiable durant l'usage de la carte : on compare souvent cette dernière au "disque dur de la carte";
- ◆ un canal d'entrée/sortie (**I/O**) qui permet la communication vers l'extérieur via la puce;
- ◆ les contacts, partie visible de la puce, qui est le point de passage obligé des informations.

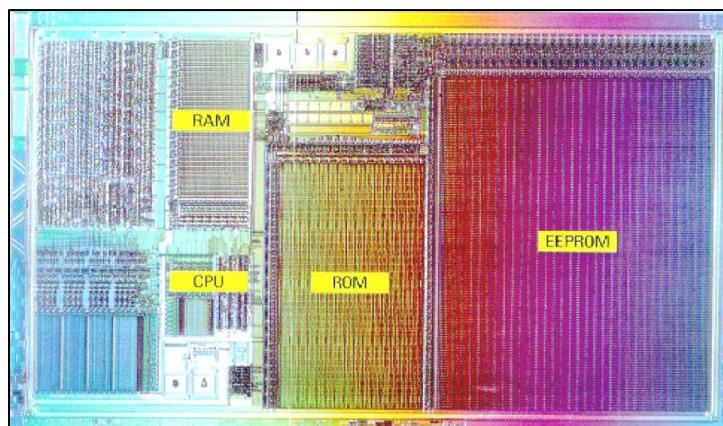
Donc, schématiquement :



memory card



smart card



2.3 Le système d'exploitation des smart cards

Le hardware qui vient d'être décrit est géré par un système d'exploitation assez primitif, dont les commandes sont exprimées dans le format du protocole de communication entre la carte et l'ordinateur qui l'utilise (ce protocole s'appelle APDU - nous allons y revenir). Ces commandes sont en fait largement orientées vers le système de fichiers; ceci se justifie par le fait qu'un application est souvent un fichier contenant des informations spécifiques à l'application.

Le système de fichiers est une structure arborescente sommes toutes assez classique :

- ◆ la racine est appelée le **Master File** (MF); il contient des éléments des deux autres types;
- ◆ un répertoire est appelé un **Dedicated File** (DF) – le MF est donc un DF particulier;
- ◆ un fichier de données est appelé un **Elementary File** (EF); il est identifié par un FID (File Identifier) de 2 bytes et peut être structuré en flot de bytes, en enregistrements (de longueur fixe ou variable) ou encore en pile.

Bien sûr, un DF peut contenir des EF et d'autres DF. Tout aussi normalement, un fichier ne peut être manipulé avant d'avoir été ouvert, ce qui se dit ici "**sélectionné**" [*selected*].

2.4 Les potentialités en matière de sécurité et d'authentification

Les cartes à puces sont capables de fournir, en plus des fonctionnalités classiques d'un OS embarqué, des services de stockage sécurisés de données pour des informations sensibles telles que :

- ◆ des clés privées;
- ◆ des numéros d'identification (comptes bancaires par exemple);
- ◆ des mots de passe;
- ◆ des informations privées personnelles, comme des informations médicales.

L'idée maîtresse est que ces informations ne sortent jamais de la carte : elles sont utilisables sur la carte même par son microprocesseur qui peut réaliser sur place les opérations d'identification et d'authentification,

Une illustration de ces possibilités est le paiement par cartes ...

2.5 La sécurité des paiements par cartes à puces

Lors de sa fabrication, une carte à puce se voit attribuer un nombre appelé la "**valeur de signature**", nombre qui est mémorisé dans la puce. En fait, il s'agit de la signature des informations publiques que comporte la carte (numéro, date de validité, éventuellement nom du propriétaire, ...). Cette signature est créée en utilisant une clé privée de type **RSA** (ou générée par l'algorithme multivarié SFLASH pour les cartes économiques), qui est stockée dans un endroit hautement sécurisé par l'émetteur de la carte. Initialement, la longueur des clés était de 320 bits, mais elle à présent de 768 bits.

Lorsqu'un client veut réaliser un paiement au moyen de sa carte à puce, la carte est introduite dans le terminal ce qui déclenche les vérifications suivantes.

1) La signature est vérifiée avec la clé publique associée à la carte. On peut ainsi vérifier l'authenticité de la carte.

En fait, cette protection rend quasiment impossible la fabrication d'une fausse carte à partir de rien, mais n'empêche pas vraiment le clonage (encore que ce soit loin d'être facile).

2) Le client tape son **code PIN** à 4 chiffres et celui-ci est vérifié à celui qui est mémorisé dans la puce.

On peut ainsi vérifier, en principe, que la carte n'a **pas été volée**. Cependant, certains hackers ont créé des "*Yes cards*" : ce sont de fausses cartes qui fournissent toujours une réponse positive à la vérification du code PIN.

3) Pour les montants d'une certaine importance, les données de la transaction (montant, numéro de carte, etc) sont **cryptées** au moyen d'un algorithme **Triple DES** utilisant une clé secrète stockée de manière sécurisée dans la puce. Le terminal transmet le résultat de ce cryptage au **centre de contrôle** qui décode avec la même clé et peut comparer qu'il retrouve les données attendues.

On peut améliorer le procédé en utilisant une valeur de signature dynamique, c'est-à-dire une signature différente à chaque transaction, calculée au moyen d'un crypto-processeur intégré à partir d'informations variables, comme par exemple l'heure de la transaction. Dans ce cas, il devient impossible de fabriquer des clones ou des Yes-cards.

Evidemment, si on voit les choses à cette échelle, il importe que les divers systèmes impliqués soient basés sur une structure commune ...

3. La communication entre une carte à puce et un ordinateur

3.1 Les dispositifs de lecture

Une carte à puce peut être accédée au moyen d'un **CAD** (Card Acceptance Device) qui, concrètement, peut être :

- ♦ un lecteur de carte connecté à un ordinateur par un câble (série, parallèle ou USB); ce lecteur fournit le courant et les signaux d'horloges, ce qui permet au CPU de la carte de se synchroniser;
- ♦ un terminal, au sens populaire du terme (c'est-à-dire comme dans une station d'essence ou un guichet automatique), soit un écran, un clavier, divers composants et un lecteur de cartes à puce.

3.2 La connexion et la déconnexion

Lorsque la carte est insérée dans la CAD, elle est à nouveau alimentée en énergie et reçoit une requête de réinitialisation de la part du lecteur. La communication ainsi établie est du type **série asynchrone**, avec par conséquent l'utilisation de bytes de contrôle (notamment début et arrêt). La réponse à cette requête (33 bytes au maximum), appelée **ATR** (Answer To Reset – réponse au démarrage), permet de négocier les caractéristiques de la liaison, comme le protocole T qui sera utilisé (voir plus loin), la vitesse de transmission, mais contient aussi des informations comme l'identification du produit, la version du système d'exploitation, le type et l'état de la carte. Un ATR typique peut ressembler à ceci :

TS	3Bh Caractère de synchronisation
T0	19h Paramètre TA1 présent - 9 octets d'information
TA1	95h Vitesse de protocole = 38400 bauds
T1	64h 5 octets d'information préémission
T2	40h Famille de produit
T3	42h Identification de l'O.S.

T4	11h Version de l'O.S. = 1.1
T5	CCh Identification de la puce
T6	83h 3 octets d'information d'état
T7	1Fh État de la carte
T8	90h SW1 au démarrage
T9	00h SW2 au démarrage

Toutes les communications entre la carte et le lecteur se terminent par l'envoi de deux mots, les Status Word (**SW**). Ceux-ci signalent si l'ordre a effectivement bien été exécuté (ordre entrant), ou si toutes les données ont été correctement transmises (ordre sortant). La demande provenant toujours du lecteur, la carte est donc considérée comme passive.

Le protocole T est le protocole de l'ISO7816 (spécification dont nous parlerons très bientôt). Ce protocole est défini selon le tableau suivant :

Protocole	Description
T = 0	asynchrone, semi-duplex, orienté octet
T = 1	asynchrone, semi-duplex, orienté bloc
T = 2	asynchrone, duplex intégral, orienté bloc
T = 3	duplex intégral, non utilisé pour le moment
T = 4	asynchrone, semi-duplex, orienté octet (extension de T=0)
T = 14	pour les fonctions nationales

Les protocoles T=0 et T=1 sont les protocoles les plus fréquemment utilisés. Les protocoles T=5 à T=13 et T=15 sont réservés à des applications et fonctions futures. T=14 est hors normes : cette valeur est utilisée actuellement dans les téléphones à cartes allemands.

La carte et le CAD peuvent alors échanger des requêtes de type APDU. Lorsque la carte est déconnectée, l'alimentation électrique est évidemment coupée.

3.3 Le protocole de communication

La communication entre une carte à puce et le CAD est régie par un protocole appelé **APDU** (Application Protocol Data Unit). Selon ce protocole, le CAD envoie des commandes et la carte envoie des réponses.

Une instruction APDU possède toujours un **numéro** codé sur un byte et appartient toujours à une **classe** (également codée sur un byte). Une classe est simplement un moyen de grouper les instructions apparentées : ainsi, les commandes ISO appartiennent à une classe hexadécimale 0x, les commandes connues des GSM à une classe Ax et les commandes propriétaires aux classes 8x (ce sera donc le cas pour les "commandes" Java) – le deuxième digit hexadécimal distingue les classes par le type de sécurité appliquée (0 indique l'absence de sécurité, 4 indique une sécurité propriétaire, etc). Une commande APDU a le format suivant :

header				corps optionnel		
classe de l'instruction	instruction	paramètre 1	paramètre 2	longueur champ de données	champ de données	longueur attendue de la réponse (max=7Fh)

Par exemple, pour sélectionner un fichier (c'est-à-dire l'ouvrir), on va envoyer la commande SELECT FILE, dont le numéro est A4h et la classe 00h. Ici, le premier paramètre permet de spécifier comment le fichier est désigné :

<i>paramètre 1</i>	<i>fichier désigné par</i>
00h	son FID
04h	son nom (pour un DF seulement)
08h	son chemin depuis le MF
09h	son chemin depuis le DF courant

Si le nom du DF est partiel (à la manière des wildcards), plusieurs DF peuvent convenir; le 2^{ème} paramètre permet de spécifier lequel est retenu :

<i>paramètre 2</i>	<i>DF retenu</i>
00h	le premier rencontré
04h	le dernier
08h	le suivant
09h	le précédent

Le contenu de la donnée dépend évidemment du premier paramètre. La commande peut donc ressembler à ceci :

00h	A4h	00h	00h	02h	1Fh 02h	7Fh
-----	-----	-----	-----	-----	---------	-----

Une **réponse APDU** contient des données ou pas, selon la commande à laquelle elle répond. Mais elle comporte toujours deux "status word" qui fourniront le code de retour de l'instruction reçue. Ainsi, **9000h** signifie "succès". La réponse présente donc le format général suivant :

corps optionnel	trailer
champ de données	status word 1 status word 2

Pour notre exemple de commande SELECT FILE, la valeur d'erreur des deux status word peut être en particulier :

<i>status words</i>	<i>signification</i>
6A 82h	fichier non trouvé
6A 86h	paramètres incorrects
6A 87h	longueur du champ de données incompatible avec les données
69 99h	le fichier n'a pu être sélectionné

tandis que les données sont des information de contrôle. Donc, ici

....	90h	00h
------	-----	-----

signifie que le fichier a bien été sélectionné.

Signalons encore que la communication entre une carte à puce et un hôte par l'intermédiaire d'un CAD est **half-duplex** : les données circulent dans un sens ou dans l'autre, mais pas dans les deux simultanément.

4. Diverses spécifications de smart cards

Il existe plusieurs spécifications concernant les cartes à puces, visant des niveaux d'utilisation divers. La spécification fondamentale est **ISO 7816** "Identification cards – Integrated circuits cards with contacts" : c'est bien sûr la norme acceptée au niveau international pour les cartes à puce. Elle est dynamique, c'est-à-dire remise à jour constamment. Il s'agit en fait d'une série de 11 normes traitant de la communication des caractéristiques de la carte utilisée, de ses propriétés physiques et de l'identification des composants et des données de base. Plus précisément :

- ◆ ISO 7816-1 : spécifie les caractéristiques physiques de la carte comme les dimensions, le rayonnement électromagnétique, la résistance mécanique et électrique, la localisation de la puce et de la bande magnétique;
- ◆ ISO 7816-2 : définit l'emplacement et les caractéristiques électriques des contacts;
- ◆ ISO 7816-3 : définit le traitement des signaux électroniques (avec notamment le voltage, la fréquence d'horloge et la vitesse de communication) et les protocoles de transmission (**TPDU** = Transmission Protocol Data Unit); pour rappel, ces derniers peuvent être T=0 (protocole orienté octet) et T=1 (protocole orienté paquet) tandis que T=14 est réservé aux les protocoles propriétaires; on y définit aussi l'**ATR** (Answer To Reset) qui correspond aux données envoyées par la carte immédiatement après la mise sous tension;
- ◆ ISO 7816-4 : définit les commandes de base de lecture, d'écriture et de mise à jour des données de la carte et, pour cela, les messages **APDU** (Application Protocol Data Unit), par lesquels les cartes à puce communiquent avec le lecteur; les échanges s'effectuent en mode client-serveur, le terminal ayant toujours l'initiative de communication;
- ◆ ISO 7816-5 : traite de l'identification des applications qui fonctionnent sur la carte au moyen d'un **AID** (Application IDentifier), qui est un tableau de 5 à 16 bytes. Celui-ci se décompose en un **RID** (Registered Application Provider Identifier ou encore Resource Identifier) qui identifie le *provider* sur 5 octets, et un **PIX** (Proprietary Identifier/Identification eXtension), groupe de 11 bytes maximum qui identifie l'application considérée;
- ◆ ISO 7816-6 : spécifie des éléments de données inter-industrie pour les échanges, tels que le numéro du porteur de carte, sa photo, sa langue, la date d'expiration, etc;
- ◆ ISO 7816-7 : définit le **SCQL** (Card Structured Query Language), langage d'interrogation d'une base de données;
- ◆ ISO 7816-8 : définit les commandes de gestion de la sécurité interne de la carte (comme les techniques de **cryptage** et de **signature**);
- ◆ ISO 7816-9 : comprend des spécifications pour diverses commandes de gestion des cartes;
- ◆ ISO 7816-10 : définit des compléments sur le traitement des signaux électriques;
- ◆ ISO 7816-11 : définit les méthodes d'identification personnelle (comme les mesures biométriques).

Citons encore d'emblée d'autres normes :

- ◆ **GSM** (Global System for Mobile communication) : définie par l'**ETSI** (European Telecommunications Standards Institute);
- ◆ **EMV** (Europay MasterCard and Visa) : basée sur la spécification ISO 7816, avec les aspects financiers en plus;
- ◆ **PS/SC** (Personal Computer/Smart Card) : définie par un consortium "Interoperability Specification for ICCs and Personal Computer Systems";

- ◆ **OCF (Open Card Framework) :**
initialement développée par IBM, elle est à présent gérée par un consortium **OpenCard**, qui regroupe des sociétés aussi connues que Sun, Visa, IBM, Siemens, American Expresss, Gemplus, ...; c'est une spécification incontournable car elle définit les règles à respecter par les fournisseurs de services – nous y reviendrons plus loin;
- ◆ **OP (Open Platform) :**
initialement développée par Visa, elle est à présent gérée par une organisation indépendante sous le nom de GlobalPlatform; son objectif est la création d'une infrastructure normalisée pour le développement, le déploiement et la gestion des cartes à puce; autrement dit, grâce aux bibliothèques fournies par GlobalPlatform, on peut communiquer avec les cartes à puce de n'importe quel fournisseur pour peu que celles-ci respectent les normes ISO-7816 – nous y reviendrons également.



Ces trois dernières normes méritent sans doute quelque attention – cela va venir.

Remarque

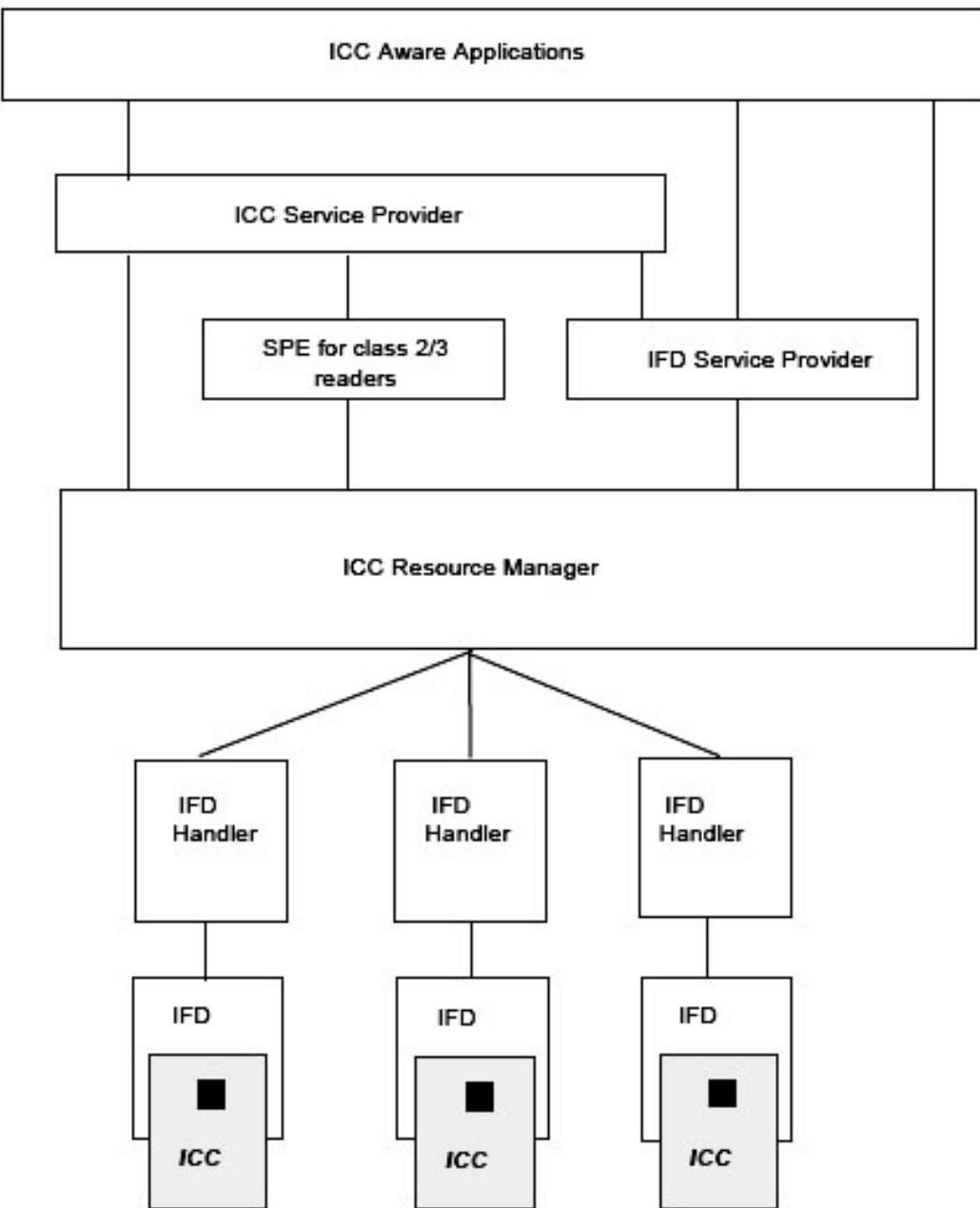
On conçoit sans peine que le développeur "haut niveau" a tout intérêt à se détacher de des normes trop restrictives ou propriétaires pour pouvoir développer des applications faisant intervenir des cartes à puces. C'est évidemment à ce point qu'un langage comme Java est le bienvenu, avec précisément des APIs de "haut niveau" ☺ ...

5. La spécifications PC/SC d'utilisation d'une carte à puce

5.1 Le vocabulaire de la spécification PC/SC

Un groupe nommé le **PC/SC Workgroup** (**Personal Computer/Smart Card Workgroup** - <http://www.pcscworkgroup.com>) a développé une spécification définissant la manière **d'intégrer un système à carte à puces** (un ICC, c'est-à-dire Integrated Circuit Card) **dans un environnement informatique** (typiquement un PC). Cette spécification est à respecter pour garantir l'interopérabilité entre les lecteurs de cartes et les cartes elles-mêmes. Le groupe comporte des sociétés aussi importantes que Toshiba, Gemalto, GemPlus, Schlumberger, Siemens et, surtout, Microsoft. La première version est apparue en 1997. Windows 2000 fut doté du support de cette spécification, et un PC/SC Lite libre pour Linux et Mac OS apparut dans la foulée.

L'architecture prônée est définie par le PS/SC Workgroup selon le schéma suivant :



avec :

1) ICC = la smart card

Dans l'architecture faisant l'objet de cette spécification, la smart card est supposée conforme aux normes ISO 7816-1, 2, 3 et 10.

2) IFD (Interface Device) est en fait le lecteur de carte, c'est-à-dire l'interface physique à travers lequel la carte va communiquer avec le PC (il assure donc le canal d'IO entre eux). C'est lui qui alimente en courant continu le microprocesseur de la carte et qui envoie les signaux d'horloge à celui-ci. Il est relié au PC au moyen d'une ligne USB, PCMCIA, série ou même clavier.

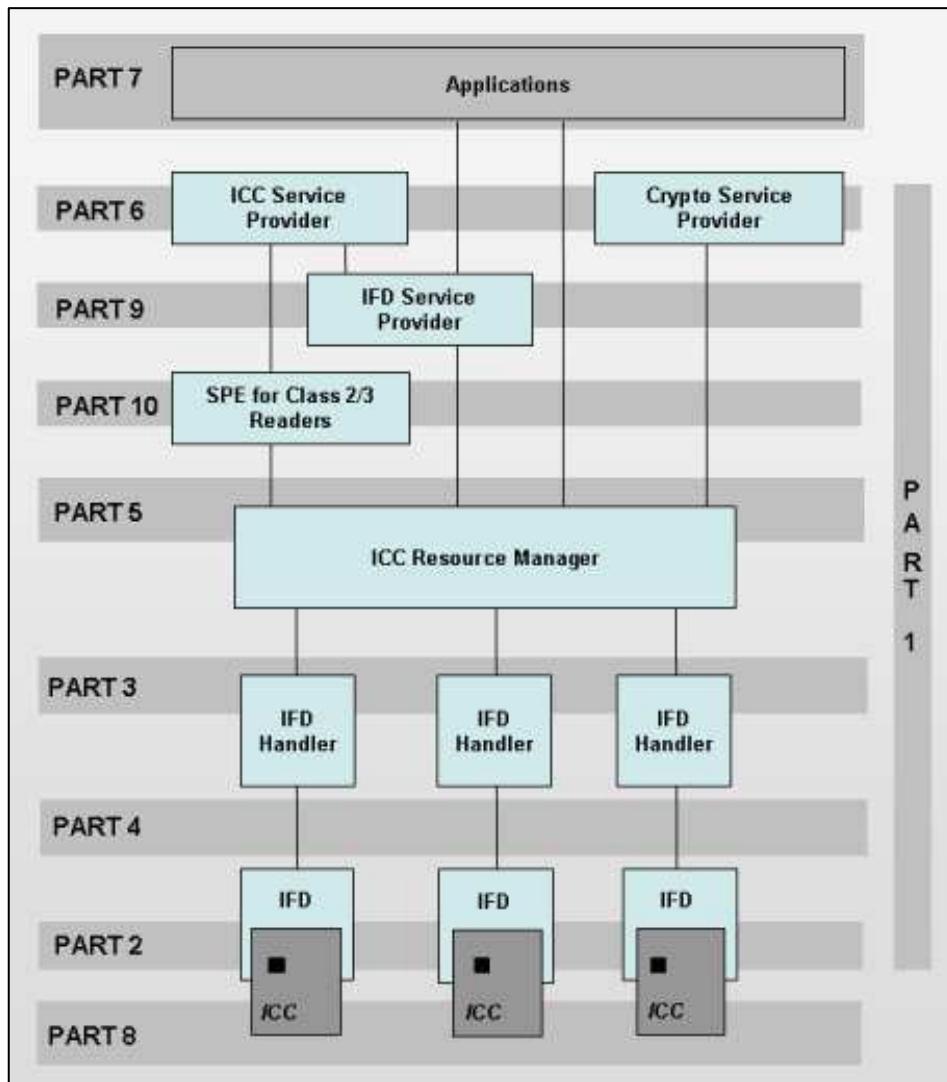
3) IFD Handler = un software bas niveau logé sur le PC capable de recueillir les données venant de l'IFD au moyen du slot associé au dispositif de lecture de cartes.

4) ICC Ressource manager = la pierre d'angle de la spécification, puisqu'il s'agit du composant qui sera fourni par le concepteur du système d'exploitation gérant l'ordinateur : on s'en doute, son rôle est de prendre en charge la gestion et le contrôle des données fournie par les ICCs au travers des IFDs. Avec un tel objectif, on comprend qu'il doit

- ◆ avoir localisé les drivers des divers ICCs utilisables;
- ◆ détecter la présence ou le retrait d'une carte;
- ◆ avoir connaissance des IFDs présents et gérer les handlers associés, notamment en les attribuant de manière partagée ou exclusive aux applications demandeuses, selon par exemple le type de ressource visée sur l'ICC; en résumé, il gère le support des transactions de base pour l'accès aux ressources de la carte.

5.2 La structure globale de la spécification

La dernière spécification PC/SC (v.2.01.4 en mai 2008) comporte 10 parties distinctes décrivant chacune les spécifications des services selon leur niveau dans l'architecture PC/SC :



- ◆ Au sommet (partie 7), on trouve donc les considérations techniques quant à la conception d'applications voulant utiliser des cartes à puce (applications dites "ICC aware" – comme dirait JCVD ;-)).

- ◆ Ensuite (partie 6) sont décrites les interfaces que doivent produire les fournisseurs de service sur cartes à puces (les providers), y compris en ce qui concerne les services cryptographiques.
- ◆ Viennent ensuite (partie 9), dans la foulée, les interfaces du niveau physique, comme par exemple l'introduction d'un code PIN, avec le cas particulier où les fabricants ont établi leur propre mécanisme de contrôle de ce code PIN (partie 10).
- ◆ La partie 5 décrit ce qui touche l'ICC Manager, c'est-à-dire comment il mettra à la disposition des applications les informations provenant des IFDs, donc finalement des ICCs, y compris les informations sur les ICCS eux-mêmes, comme leur type, le fait qu'une carte est insérée ou retirée, etc.
- ◆ Les parties 3 et 4 décrivent les spécifications des lecteurs et de leurs pilotes.
- ◆ La partie 2 traite les communications bas niveau entre les ICCs et les IFDs.
- ◆ Enfin, la partie 8 fournit les spécifications pour les ICCs de manière à permettre les mécanismes d'authentification et d'identification.
- ◆ Quant à la partie 1, elle décrit l'ensemble de l'architecture.

Nous parlerons plus loin de l'autre incontournable spécification : Open Card. Mais pour l'illustrer, nous aurons besoin de Java ...

6. La technologie Java Card

6.1 Java avec peu de mémoire

L'idée est donc de permettre aux développeurs de placer du code Java exécutable sur une carte à puce. Une application écrite pour une Java card est encore appelée, de manière un peu maladroite sans doute, une "**applet**". Ce nom vient du fait que, tout comme une applet HTTP est téléchargée sur un client qui ne la possède pas, une applet pour Java card est téléchargée sur une carte après la fabrication de celle-ci ...

Si le concept est simple, la réalisation pratique l'est certainement beaucoup moins : en effet, ceci sous-entend que la carte à puce possède alors une machine virtuelle. Cette dernière occupe évidemment une certaine place mémoire – or, celle-ci est précisément limitée sur une smart card. Pour contourner ce problème, on a donc imaginé de définir :

- ◆ un langage Java pour smart cards [*Java Card language subset*] qui soit un sous-ensemble du Java standard; la gestion d'un tel langage prend évidemment moins de place;
- ◆ une machine virtuelle propre aux smart cards, la **JCVM** (*Java Card Virtual Machine*) **éclatée en deux parties** : l'une fonctionne effectivement sur la carte, mais l'autre fonctionne sur la machine hôte qui possède le CAD.

6.2 Le sous-ensemble de Java pour les smart cards

Considérant le caractère assez fruste des commandes comprises par les cartes à puce, il a été décidé de conserver dans le Java pour cartes :

- ◆ les types primitifs de petite taille : byte, short, boolean;
- ◆ les tableaux à une dimension;
- ◆ les concepts de base que sont les classes, les interfaces, les packages, les exceptions, la création dynamique des objets et l'héritage.

Mais ne sont plus supportés :

- ◆ les types primitifs de grande taille : long, double, float;
- ◆ les caractères et les Strings;
- ◆ les tableaux à plus d'une dimension;
- ◆ le garbage collector;
- ◆ les threads;
- ◆ la sérialisation et le clonage;
- ◆ le security manager.

Ce dernier point peut paraître surprenant, puisque l'un des arguments massues en faveur des smart cards est précisément la sécurité – mais, en fait, c'est la machine virtuelle elle-même qui gère la sécurité, sans faire appel à la classe SecurityManager de java.lang (cette classe est analysée au chapitre consacré à la sécurité de la plate-forme Java).

6.3 La plate-forme Java Card

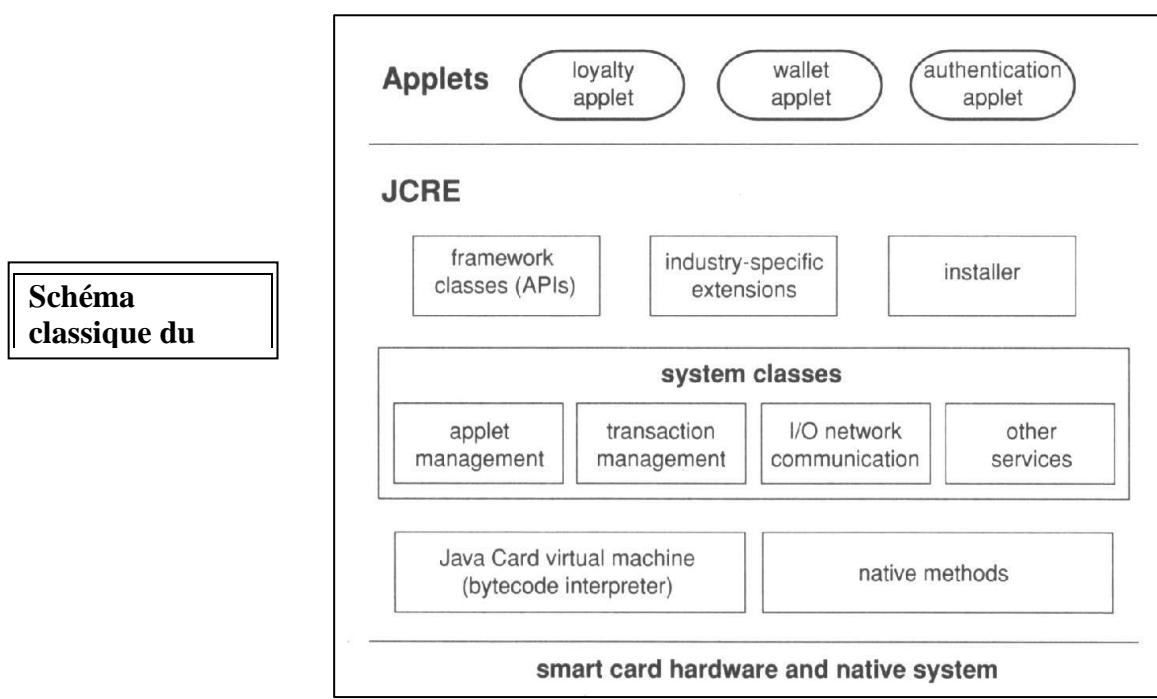
La plate-forme de développement appelée "technologie Java Card" comporte trois éléments, aux spécifications bien établies : la machine virtuelle, l'environnement d'exécution et les APIs. Toutes en sont à la version **2.2**.

La **JCVM** (machine virtuelle des Java cards) comporte donc :

- ◆ une composante sur la carte elle-même, qui comporte l'interpréteur de bytecode;
- ◆ une composante à l'extérieur de la carte, qui est appelée le **convertisseur** [*converter*].

A l'image du traditionnel JRE, cette machine virtuelle appartient au **JCRE** (**J**ava **C**ard **R**untime **E**nvironment) qui comporte :

- ◆ donc la machine virtuelle **JCVM**;
- ◆ les classes systèmes qui gèrent la **cryptographie**, la mémoire, le bas niveau;
- ◆ les **méthodes natives** qui fournissent le support pour la machine virtuelle et le système de classes;
- ◆ les APIs que les applets pour Java card utilisent;
- ◆ l'**installateur** qui permet le téléchargement des applets;
- ◆ des extensions éventuelles, comme par exemple les extensions financières.



Quant aux APIs, elles sont groupées en 4 packages.

7. L'API Java Card

Il s'agit donc ici des APIs que les applets destinées à fonctionner sur des cartes à puce utiliseront. Bien sûr, nous sommes ici dans un monde spartiate, sans GUIs, réseaux, flux, threads, ... Plus précisément, les packages utilisables sont `java.lang` (dans une version limitée), `javacard.framework` (qui fournit l'essentiel pour la programmation des applets) et les deux packages sécuritaires `javacard.security` et `javacardx.crypto`.

7.1 Les 4 packages

1) **java.lang** est un sous-ensemble du `java.lang` du JRE classique; on n'y trouve que les classes **Object** et **Throwable**, avec des classes d'exception permettant la gestion des erreurs qui peuvent survenir dans le monde restreint de la JCVM (`ArithmeticException`, `ArrayIndexOutOfBoundsException`, `NullPointerException`, `SecurityException`, ...).

2) **javacard.framework** est le package fondamental pour le développement Java pour smart cards; signalons d'emblée l'existence

- ◆ de la classe **Applet** : toute applet doit dériver de cette classe;
- ◆ de la classe **APDU**, qui encapsule les commandes du protocole APDU et permet de les manipuler assez simplement;
- ◆ de l'interface **PIN** et de la classe qui l'implémente **OwnerPIN** : elle encapsule le code **PIN** (Personal Identification Number) sous forme d'un short, permettant ainsi l'authentification d'un utilisateur selon le principe du mot de passe;
- ◆ de la classe **JCSYSTEM**, qui remplace la classe `System` classique et qui fournit l'interface avec le système sous-jacent.

Bien sûr, les classes d'exception associées sont fournies également, comme `APDUException`, `ISOException`, etc.

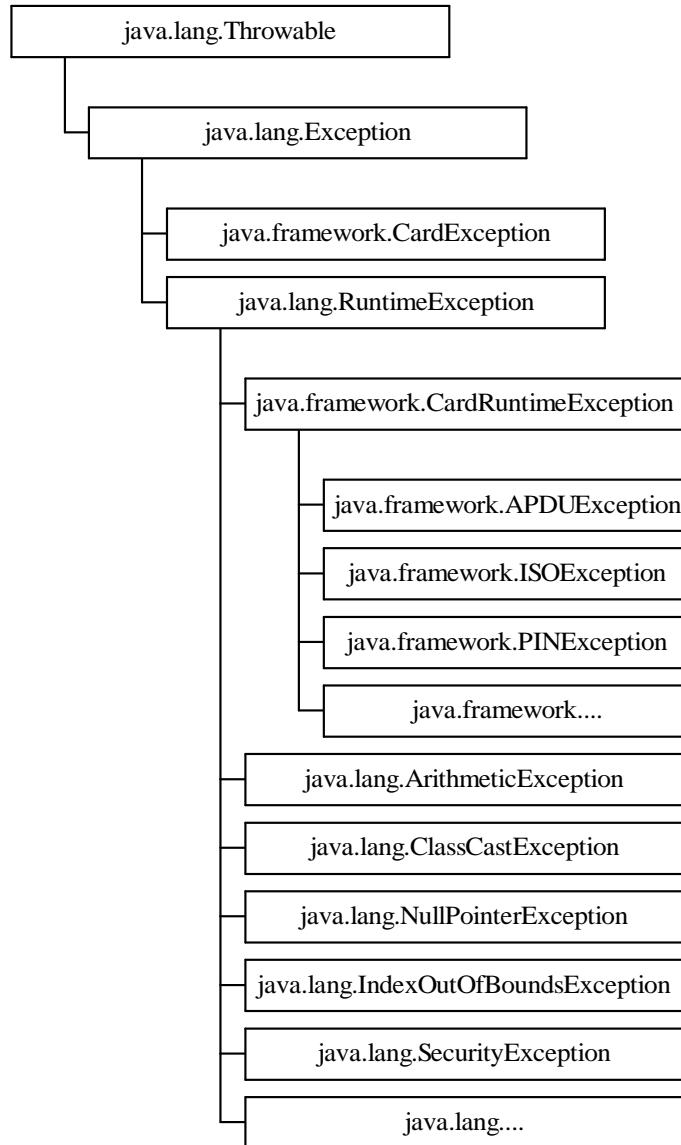
3) **javacard.security** fournit les interfaces et les classes habituels du monde cryptographique, comme

- ◆ `DESKey`, `DSAKey`, `DSAPrivateKey`, `DSAPublicKey`, `Key`, `PrivateKey`, `PublicKey`, `RSAPrivateCrtKey`, `RSAPrivateKey`, `RSA PublicKey`, `SecretKey`
- ◆ `KeyBuilder` (une factory), `MessageDigest`, `RandomData`, `Signature`, `CryptoException`

4) **javacardx.crypto** étend le package précédent en apportant les classes et interfaces propres aux USA, comme la classe `Cipher`.

7.2 Les classes d'exception

Les classes d'exception se positionnent dans une hiérarchie distinguant les exceptions "checkées" (dérivées de la classe CardException) et "non checkées" (dérivées de la classe RuntimeException), comme dans la plate-forme Java habituelle :



On peut remarquer que, comme la classe `String` n'est pas supportée sur la plate-forme Java Card, on ne dispose plus pour les exceptions du message d'erreur auquel nous nous sommes accoutumés dans nos développements Java habituels. L'information sur l'exception sera matérialisée au moyen d'un `short` appelé le "**reason code**", variable membre des classes `CardException` et `CardRuntimeException`. Le réflexe de l'appel de `getMessage()` dans un `catch` sera donc à remplacer par celui de l'appel de

`public short getReason()`.

8. L'exécution d'une applet pour Java Card

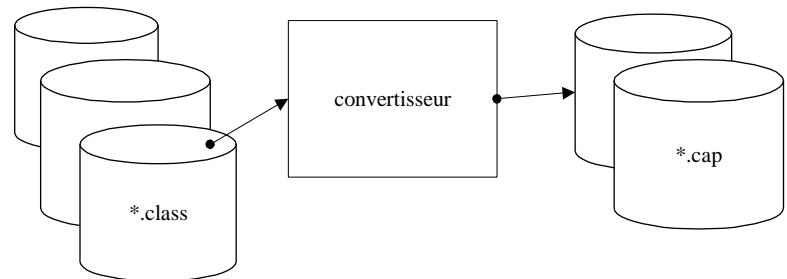
Une applet est donc une classe Java et doit donc être, tout d'abord, compilée pour produire des fichiers bytecode d'extension class. Cependant, les choses se compliquent forcément ensuite, puisque la machine virtuelle qui va gérer ces bytecodes est composée de deux parties.

8.1 Le convertisseur

En fait, les fichiers class ne vont pas être traités un par un, mais regroupés obligatoirement dans un package.

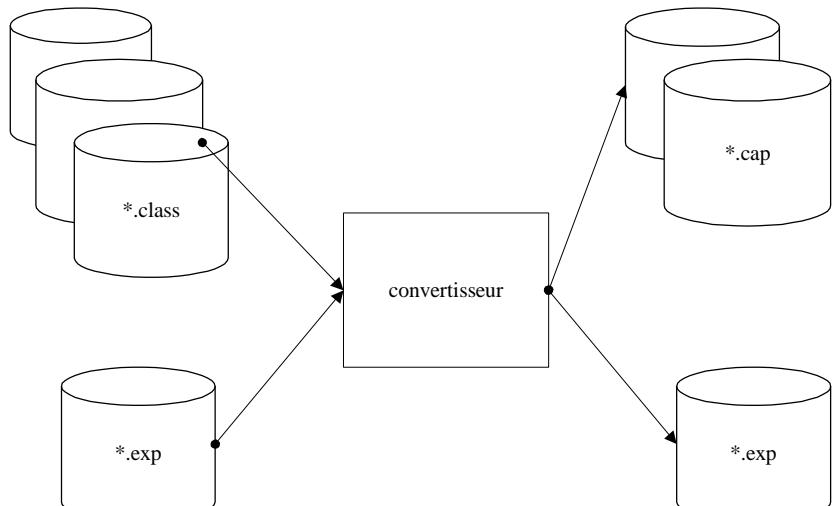
Le convertisseur va transformer le package généré par le compilateur en un fichier **CAP** (**Converted APplet**).

Un fichier CAP est en fait un fichier jar contenant des informations diverses sur l'applet : bytecodes, informations sur la classe, informations de link, etc – bref, tout ce qui est nécessaire pour exécuter l'applet dans un environnement Java Card quelconque. Pour en arriver à cela, le convertisseur réalise des opérations qu'une machine virtuelle classique n'effectuerait



qu'au moment du chargement de la classe : vérification du bytecode et de son adéquation au sous-ensemble de Java pour les smart cards, initialisation des variables statiques, optimisations, allocations des structures décrivant les classes, etc. Donc :

En réalité, le convertisseur utilise et produit également des fichiers export (**EXP**). De tels fichiers ne sont pas téléchargés sur la carte et, donc, ne seront pas utilisés par l'interpréteur. Cependant, ils contiennent toutes les informations publiques sur les packages (en entrée) ou sur l'applet (en sortie), permettant ainsi à tout développeur d'utiliser cette applet sans pour autant en connaître l'implémentation. C'est bien cela : ces fichiers export sont les équivalents des fichiers headers (*.h) du C ...



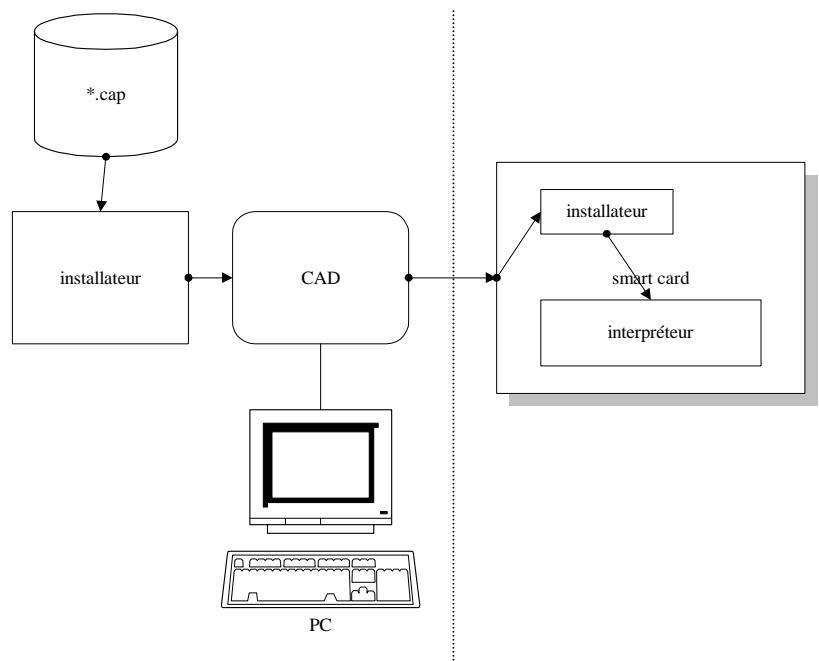
Enfin, le convertisseur peut également produire un fichier **JCA** (Java Card Assembly), fichier texte reflétant le fichier CAP et en permettant l'examen.

8.2 Le téléchargement et l'exécution de l'applet

Une Java Card possède un **installateur** (plus précisément le *on-card-installer*), dont le rôle est de

- ◆ de récupérer, via le CAD, le fichier CAP envoyé par un programme d'installation fonctionnant sur l'hôte (le *off-card installer*) sur lequel le CAD est connecté;
- ◆ d'écrire le fichier CAP dans la mémoire de la carte;
- ◆ de créer les liens éventuels avec d'autres classes déjà présentes sur la carte;
- ◆ d'initialiser les structures utilisées par le JCRC pour l'exécution de l'applet;
- ◆ d'associer à l'applet un "securiy domain", applet particulière destinée à stocker les clés cryptographiques de l'applet.

Il ne reste évidemment plus à l'interpréteur qu'à exécuter l'applet, en contrôlant les allocations mémoires et la sécurité.



9. L'identification d'une applet

Une applet pour Java Card est identifiée au moyen de son **AID** (Application IDentifier), qui est un tableau de 5 à 16 bytes. Pour rappel, cet AID se décompose en

- ◆ un **RID** (Resource IDentifier) : ce groupe de 5 bytes identifie le *provider*, c'est-à-dire la société propriétaire de l'applet; l'ISO régit l'attribution de ces RIDs aux sociétés.
- ◆ un **PIX** (Proprietary Identification eXtension) : ce groupe de 11 bytes maximum identifie l'*applet*; il est attribué par le provider lui-même.

Il en est de même pour un package et c'est au moyen de l'AID que le link avec ce package sera réalisé. Bien sûr, les applets d'un même package partagent le même RID.

10. Les méthodes de base d'une applet pour Java card

Une fois les objectifs de l'applet définis, il s'agit d'en rédiger le code. Comme nous l'avons déjà dit, toute applet hérite de la classe Applet du package javacard.framework. A priori, notre applet devra redéfinir les méthodes suivantes :

1) public static void install(byte[] bArray, short bOffset, byte bLength) throws ISOException

Cette méthode de classe crée une instance de l'applet, éventuellement par constructeur interposé. Le tableau de bytes passé comme paramètre représente les paramètres d'installation tels que définis au sein de l'applet. En fait, **cette méthode est le point d'entrée de l'applet**, une espèce d'équivalent du main() pour une application Java. Elle doit impérativement appeler l'une des méthodes

protected final void register() throws SystemException

mais surtout

```
protected final void register(byte[] bArray, short bOffset, byte bLength) throws
SystemException
```

Cette méthode attribue à l'applet l'AID passé comme paramètre sous forme d'un tableau de bytes.

Il convient de bien remarquer ici qu'il n'y a pas de chargement dynamique de classes sur la carte au moment de l'exécution de l'applet; donc, les classes et les packages auxquels elle fait référence doivent avoir été placés sur la carte au préalable.

2) public boolean **select()**

public void deselect()

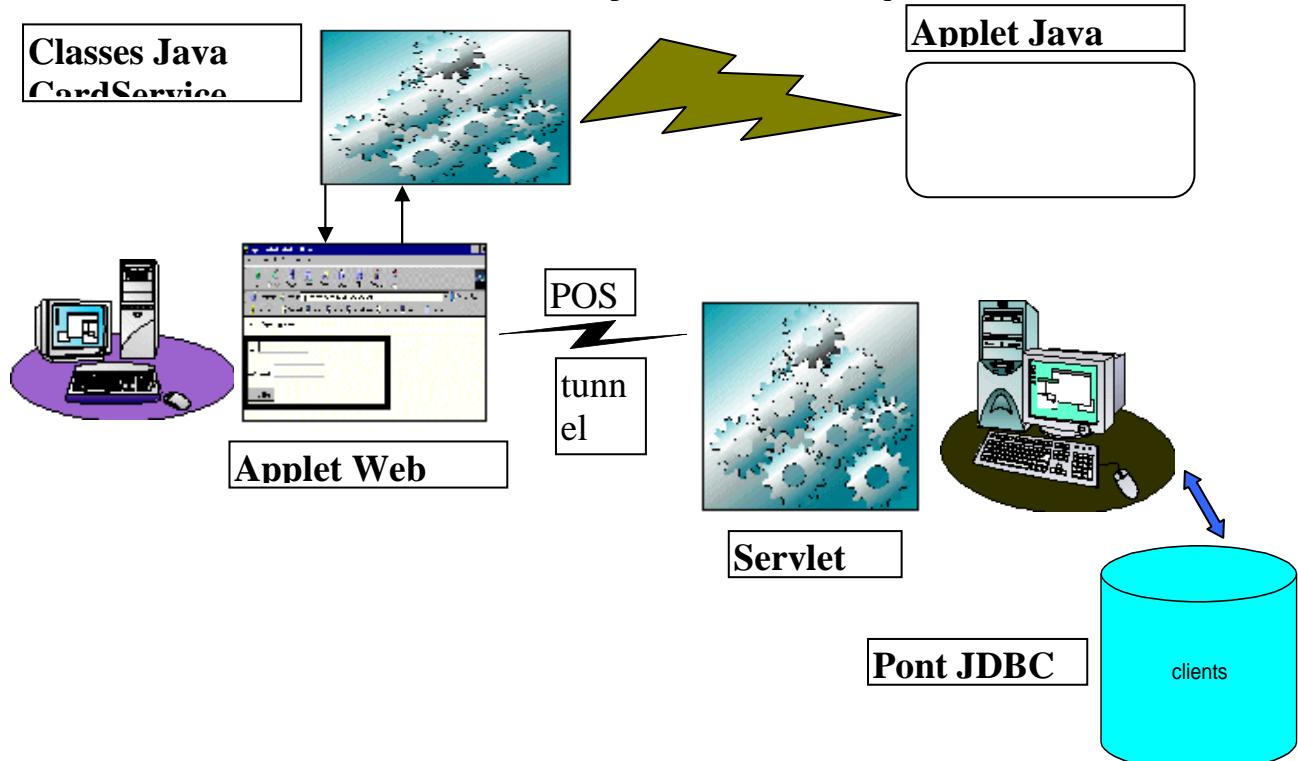
Ces méthodes sont évidemment appelées lorsque l'applet est sélectionnée ou au contraire désélectionnée. On voit donc clairement ici qu'une seule applet est exécutée à la fois (pas de multithread ;-)). Lorsque l'applet est sélectionnée, elle attend typiquement que l'application fonctionnant sur la machine hôte du CAD envoie une commande, ce qui nous conduit à la commande suivante ...

3) public abstract void **process(APDU apdu)** throws ISOException

Cette méthode est évidemment appelée chaque fois qu'une commande APDU est entrée sur la carte. C'est donc au sein de cette méthode que la véritable logique de l'applet se trouve.

11. L'exemple classique du e-commerce : le porte monnaie électronique

Pour fixer les idées, nous allons nous placer dans le contexte typique du e-commerce du futur. Autrement dit, le client télécharge une applet HTTP qui non seulement va dialoguer avec une servlet et lui permettre de remplir un caddie virtuel, mais cette même applet va lui permettre d'utiliser les services de la carte à puce placée dans un CAD raccordé à la machine client, essentiellement l'authentification et les paiements. Schématiquement :



Le dialogue applet-servlet fait partie de nos vieilles connaissances : nous ne nous y attarderons donc pas. Restent

- ◆ les problèmes de communication de l'applet HTTP avec la carte : par défaut, prisonnière de sa sandbox, elle ne pourra pas accéder à la ligne série et il faudra donc en faire une applet signée (voir chapitre XXIV consacré à la sécurité de la plate-forme Java);
- ◆ la création et la mise au point de l'applet Java Card elle-même.

Nous nous étendrons évidemment sur ce dernier point ...

12. La construction de l'applet porte-monnaie

12.1 Les fonctionnalités

Le porte monnaie électronique, que nous allons étudier ici, est l'exemple typique dans le contexte des applications des smart cards: Sun le propose dans son kit de développement (c'est l'applet *Wallet*). Mais, même ainsi, les choses ne sont pas si simples ...

Détaillons donc patiemment les composantes de notre classe applet, finement nommée *appletPorteMonnaie* et qui dérive donc de la classe **Applet** du JCRE :

appletPorteMonnaie.java

```
package eCommerce;  
  
import javacard.framework.*;  
  
public class appletPorteMonnaie extends Applet  
{  
    ...  
}
```

Les fonctionnalités que nous en attendons de cette applet seront simplement :

- ◆ la mémorisation d'une certaine somme d'argent (c'est le contenu du porte-monnaie);
- ◆ le retrait d'une certaine somme;
- ◆ le rechargeement du porte-monnaie;
- ◆ la consultation du compte.

12.2 L'algorithme de sécurité

Avant de pouvoir utiliser la carte (pour un débit ou un rechargeement par exemple), l'applet va appliquer un algorithme de sécurité, qui demande le code **PIN** (**P**ersonal **I**dentity **N**umber) : ce code de 8 chiffres maximum permet l'authentification d'un utilisateur selon le principe du mot de passe : trois essais infructueux bloquent l'utilisation de la carte. En pratique, un vrai porte-monnaie électronique applique cependant des algorithmes plus complexes.

Quoiqu'il en soit, ce code PIN sera défini ici "à la dure" au moment de l'installation et de la création de l'applet. En pratique, encore une fois, il faudrait un utilisateur administrateur chargé de définir les codes PIN des utilisateurs.

12.3 La définition de l'AID

On se souviendra qu'il faut définir un AID pour le package des classes Java et pour l'applet proprement dite et aussi que cet AID se décompose en un **RID** identifiant le provider et un **PIX** identifiant l'applet ou le package. Dans le cas de Sun, le RID à 5 bytes connu de l'ISO est :

0xa0:0x0:0x0:0x0:0x62

Pour nous, ce sera ce que nous voudrons tant que l'usage reste interne, par exemple :

0x00:0x00:0x00:0x00:0x01

Il faut encore définir deux PIX; dans le cas de Sun, ont été choisis :

- ◆ pour l'applet : 0x03:0x01:0x0c:0x06:0x01
- ◆ pour le package : 0x3:0x1:0xC:0x6

Pour nous, disons que ce sera simplement **0x01**.

L'AID de l'applet sera donc en définitive : **0x00:0x00:0x00:0x00:0x01:0x01**.

12.4 Les commandes de la smart card

Notre applet va communiquer avec l'application fonctionnant sur la machine hôte connectée au CAD en utilisant le protocole APDU. Les commandes qui lui sont propres devront appartenir à une classe bien précise, disons par exemple **0xB0**. Elle doit alors

a) nécessairement supporter la commande standard **SELECT FILE** (évoquée au paragraphe 3.2 : numéro=A4h et classe=00h); vu ce qui précède, elle a donc la forme :

0x00	0xA4	0x04	0x00	0x0A	0xa0:0x0:0x0:0x0:0x62:0x03:0x01:0x0c:0x06:0x01	0x7F
------	------	------	------	------	--	------

b) nécessairement supporter une commande **VERIFY**, dont le numéro est 20h et la classe B0h. Cette commande permet notamment la vérification du PIN évoquée ci-dessus. Ici, le premier paramètre est laissé à 0 tandis que le 2^{ème} paramètre permet de spécifier à quelle donnée il est fait référence :

<i>paramètre 2</i>	<i>donnée utilisée</i>
0x00	pas de référence
0x0X	référence à une donnée globale de référence X
0x8X	référence à une donnée locale de référence X

Elle a donc la forme :

0xB0	0x20	0x00	0x00	0x<longueur code PIN>	<code PIN>	0x7F
------	------	------	------	-----------------------	------------	------

La réponse spécifie dans ses deux "status word" le code de retour de l'instruction

<i>status words</i>	<i>signification</i>
0x6300	la vérification a échoué
0x63CX	la vérification a échoué et il reste X essais
0x6983	authentification bloquée
0x6A86	paramètres incorrects
0x6A88	données référencées non trouvées

sachant que 0x9000 signifie toujours "succès".

c) implémenter des commandes APDU en rapport avec ses fonctionnalités; dans notre cas, il y aura donc 3 commandes, respectivement pour le retrait (commande APDU **DEBIT**), le rechargeement (commande APDU **CREDIT**) et la consultation (commande APDU **GET BALANCE**). Comme ces commandes sont propres à l'application, leur définition est la discréption du développeur (SAUF 0xB0 0x60 pris par l'ISO). Seul le code de succès 0x9000 est conservé comme succès pour toutes les commandes. Ici :

CREDIT :

0xB0	0x30	0x00	0x00	0x01	<montant du rechargeement>	0x7F
------	------	------	------	------	----------------------------	------

<i>status words</i>	<i>signification</i>
0x6301	la vérification du PIN est nécessaire avant une transaction
0x6A83	montant invalide (ou négatif)
0x6A85	montant retiré trop important pour le porte-monnaie

DEBIT :

0xB0	0x40	0x00	0x00	0x01	<montant du retrait>	0x7F
------	------	------	------	------	----------------------	------

<i>status words</i>	<i>signification</i>
0x6301	la vérification du PIN est nécessaire avant une transaction
0x6A83	montant invalide (ou négatif)
0x6A84	montant trop important pour le porte-monnaie

GET BALANCE :

0xB0	0x50	0x00	0x00	?	?	0x02
------	------	------	------	---	---	------

Le montant du solde se trouvera dans le champ de données de la réponse – pas de code particulier.

12.5 La définition des codes d'instructions

Bien sûr, toutes ces valeurs concernant les instructions et les status words vont être définies au sein de l'applet comme des constantes de classe :

appletPorteMonnaie.java (2)

```
package eCommerce;
```

```
import javacard.framework.*;
```

```

public class appletPorteMonnaie extends Applet
{
    /* Définition des constantes de classe */
    // code of CLA byte in the command APDU header
    final static byte PorteMonnaie_CLA =(byte)0xB0;

    // codes of INS byte in the command APDU header
    final static byte VERIFY =(byte) 0x20;
    final static byte CREDIT =(byte) 0x30;
    final static byte DEBIT =(byte) 0x40;
    final static byte GET_BALANCE =(byte) 0x50;

    // Valeurs de status words
    // signal that the PIN verification failed
    final static short SW_VERIFICATION_FAILED = 0x6300;
    // signal the the PIN validation is required for a credit or a debit transaction
    final static short SW_PIN_VERIFICATION_REQUIRED = 0x6301;
    // signal invalid transaction amount > MAX_TRANSACTION_AMOUNT or < 0
    final static short SW_INVALID_TRANSACTION_AMOUNT = 0x6A83;
    // signal that the balance exceed the maximum
    final static short SW_EXCEED_MAXIMUM_BALANCE = 0x6A84;
    // signal the the balance becomes negative
    final static short SW_NEGATIVE_BALANCE = 0x6A85;

    // Pour les contrôles
    // maximum balance
    final static short MAX_BALANCE = 0x7FFF;
    // maximum transaction amount
    final static byte MAX_TRANSACTION_AMOUNT = 127;
    // maximum number of incorrect tries before the PIN is blocked
    final static byte PIN_TRY_LIMIT =(byte)0x03;
    // maximum size PIN
    final static byte MAX_PIN_SIZE =(byte)0x04;
    ...
}

```

On y a ajouté les constantes permettant de réaliser divers contrôles, comme le crédit maximum ou le montant maximum d'une transaction. La taille maximale du code PIN a été fixée à 4, ce qui est une taille assez courante.

12.6 L'instanciation de l'applet

La méthode de classe **install()** est donc celle qui est appelée à la fin de l'installation de l'applet sur la carte (dont nous détaillerons les opérations plus loin). Classiquement, cette méthode instancie l'applet en en appelant le constructeur, qui est d'ailleurs **privé**. Celui-ci, tout naturellement, instancie les objets dont l'applet aura besoin et les initialise; ensuite, il enregistre l'applet auprès du JCRE en appelant la méthode **register()**.

Ici, le constructeur reçoit le tableau de bytes passés comme paramètre à la méthode **install**. Ce tableau contient des paramètres dont l'interprétation dépend entièrement du bon vouloir du programmeur – on peut donc tout imaginer. Ici, le tableau de bytes tels que les développeurs de Sun l'on imaginé, se décrypte ainsi :

bArray

longueur de l'AID	AID	longueur des informations	informations sur l'applet	longueur des données	données – ici, le code PIN
-------------------	-----	---------------------------	---------------------------	----------------------	----------------------------

Le plus urgent, pour pouvoir manipuler l'applet, est d'enregistrer le code PIN qui sert de sésame. Tout d'abord, il faut instancier la variable membre qui instancie la classe **OwnerPIN**, dont le rôle semble assez clair. Le constructeur

public **OwnerPIN**(byte tryLimit, byte maxPINSIZE) throws PINException

réclame simplement le nombre d'essais maximum qui conduit à un blocage de la carte et la taille maximale du PIN (au moins 1). C'est ensuite le rôle de la méthode

public void **update** (byte[] pin, short offset, byte length) throws PINException

d'enregistrer localement le code PIN, de réinitialiser le compteur d'essais et de repositionner un flag interne de validation. En principe, elle se procure le PIN dans le tableau donné en paramètre, à partir de la position indiquée. Mais ce code PIN doit provenir d'une commande de l'administrateur - pour ne pas nous égarer, comme déjà précisé, nous allons fixer ce code PIN à la dure avec la valeur "1234". La méthode est susceptible de lancer une exception PINException avec le reason code PINException.ILLEGAL_VALUE dans le cas où la taille du PIN proposé dépasse la taille maximale prévue dans la configuration.

Le fait de créer l'applet, dans la méthode **install()**, au moyen de l'opérateur **new** assure que cet objet ainsi créé est **persistent** : ceci signifie que

- ◆ ses valeurs sont placées dans l'EEPROM et sont donc conservées entre deux sessions CAD;
- ◆ la mise à jour d'une variable membre est **atomique**, c'est-à-dire qu'en cas de défaillance de l'alimentation durant l'opération, l'ancienne valeur est restaurée.

Donc :

appletPorteMonnaie.java (3)

```
package eCommerce;
import javacard.framework.*;
public class appletPorteMonnaie extends Applet
{
    /* Définition des constantes de classe */
    ...
    /* Définition des variables d'instance */
    OwnerPIN pin;
    short sold; // 2 bytes

    /* Constructeur */
    private appletPorteMonnaie (byte[] bArray,short bOffset,byte bLength)
    {
        pin = new OwnerPIN(PIN_TRY_LIMIT, MAX_PIN_SIZE);
```

```

// En principe :
/*byte iLen = bArray[bOffset]; // aid length
bOffset = (short) (bOffset+iLen+1);
byte cLen = bArray[bOffset]; // info length
bOffset = (short) (bOffset+cLen+1);
byte aLen = bArray[bOffset]; // applet data length
*/
bArray[0]=0x01; bArray[1]=0x02; bArray[2]=0x03; bArray[3]=0x04;

pin.update(bArray, (short)(MAX_PIN_SIZE), aLen);
solde = 0;
register();
}

/* Point d'entrée de l'applet */
public static void install(byte[] bArray, short bOffset, byte bLength)
{
    new appletPorteMonnaie (bArray, bOffset, bLength);
}
...
}

```

Point n'est sans doute besoin d'expliquer que la variable membre solde représente la somme contenue dans le porte-monnaie – oups ! trop tard ;-) ...

Remarque

Il est possible de rendre des objets "transient", c'est-à-dire non persistants et mémorisés dans la RAM, au moyen d'APIs de la classe JCSysyem. De telles données temporaires sont remises à zéro lors de la désélection de l'applet ou lors d'un reset. Ces APIs travaillent toutes sur des tableaux et se nomment

```
public static boolean[] makeTransientXXXArray(short length, byte event)
throws SystemException
```

où

- ◆ XXX peut être boolean, short, byte et Object - ceci peut présenter de l'intérêt, par exemple, pour les codes PIN;
- ◆ le 1^{er} paramètre donne la longueur du tableau (qu'il appartiendra au développeur de remplir);
- ◆ le 2^{ème} paramètre peut être l'une des deux valeurs :

```
public static final byte CLEAR_ON_DESELECT
public static final byte CLEAR_ON_RESET
```

12.7 La méthode de réponse à la sélection

L'applet ainsi créée est inactive. Elle ne démarrera que lorsque l'applet sera sélectionnée au moyen de la commande APDU SELECT paramétrée avec l'AID de cette applet. Lorsque le JCRC reçoit une telle commande, il invoque la méthode select() de l'applet. Celle-ci renvoie true ou false selon que l'applet est prête à traiter une commande APDU (via la méthode process()) ou pas. Pour décider de cela, un minimum est sans doute de se demander si l'applet n'est pas bloquée, ce qui peut se savoir au moyen de la méthode de PIN implémentée dans OwnerPIN :

```
public byte getTriesRemaining()
```

qui retourne le nombre d'essais qui peuvent être encore tentés avant blocage. Donc :

appletPorteMonnaie.java (4)

```
package eCommerce;

import javacard.framework.*;

public class appletPorteMonnaie extends Applet
{
    /* Définition des constantes de classe */
    ...
    /* Définition des variables d'instance */
    ...
    /* Constructeur */
    ...
    /* Point d'entrée de l'applet */
    ...
    /* En cas de sélection */
    public boolean select()
    {
        if ( pin.getTriesRemaining() == 0 ) return false;
        return true;
    }
}
```

Inutile de dire que si une autre applet est active, elle sera désélectionnée avant que la nouvelle applet ne soit rendue active.

12.8 La méthode de reconnaissance des commandes

Lorsqu'une applet est sélectionnée, toute commande APDU autre que SELECT reçue par le JCRC provoque l'appel immédiat de la méthode process() (pour une commande SELECT qui a réussi, il y a auparavant, comme nous le savons, appel de la méthode select()). Celle-ci reçoit un objet créé par le JCRC et instance de la classe **APDU** qui encapsule une commande du même nom. Cet objet mémorise la trame APDU dans un tableau de bytes interne que l'on appelle le buffer APDU (logique ;-)). Plus précisément :

- ♦ le JCRC n'écrit tout d'abord dans le buffer que le header de la commande avec la longueur des données éventuelles pour ensuite appeler la méthode process() de l'applet active;

- ◆ si la commande possède effectivement des données, l'applet peut les demander par l'intermédiaire de l'objet APDU et elles seront disponibles dans le buffer;
- ◆ si l'applet veut renvoyer des données, elle les place dans le buffer de l'objet APDU; le JCRE les enverra dans la réponse.

Il reste maintenant à décoder la trame APDU pour reconnaître la commande reçue, enclencher les opérations associées et construire une réponse. L'interface **ISO7816** possède toutes les constantes nécessaires pour réaliser les déplacements et les reconnaissances dans le buffer, que ce soit pour lire (commande) ou écrire (réponse). Ainsi, on peut y trouver les constantes de déplacement :

```
public static final byte OFFSET_CLA = 0
public static final byte OFFSET_INS = 1
public static final byte OFFSET_P1 = 2
public static final byte OFFSET_P2 = 3
public static final byte OFFSET_LC = 4
public static final byte OFFSET_CDATA = 5
```

et aussi les constantes définissant les status words de la réponse :

```
public static final short SW_INS_NOT_SUPPORTED = 0x6D00
public static final short SW_CLA_NOT_SUPPORTED = 0x6E00
public static final short SW_NO_ERROR = (short)0x9000
public static final short SW_BYTRES_REMAINING_00 = 0x6100
public static final short SW_WRONG_LENGTH = 0x6700
public static final short SW_SECURITY_STATUS_NOT_SATISFIED = 0x6982
public static final short SW_FILE_INVALID = 0x6983
public static final short SW_DATA_INVALID = 0x6984
public static final short SW_CONDITIONS_NOT_SATISFIED = 0x6985
public static final short SW_COMMAND_NOT_ALLOWED = 0x6986
public static final short SW_APPLET_SELECT_FAILED = 0x6999;
public static final short SW_WRONG_DATA = 0x6A80
public static final short SW_FUNC_NOT_SUPPORTED = 0x6A81
public static final short SW_FILE_NOT_FOUND = 0x6A82
public static final short SW_RECORD_NOT_FOUND = 0x6A83
public static final short SW_INCORRECT_P1P2 = 0x6A86
public static final short SW_WRONG_P1P2 = 0x6B00
public static final short SW_CORRECT_LENGTH_00 = 0x6C00
public static final short SW_UNKNOWN = 0x6F00
public static final short SW_FILE_FULL = 0x6A84
```

Concrètement, les opérations à prévoir sont les suivantes.

a) Il faut donc tout d'abord se procurer le buffer au moyen de la méthode de la classe APDU :

```
public byte[] getBuffer()
```

soit ici :

```
public void process(APDU apdu)
{
    byte[] buffer = apdu.getBuffer();
    ...
}
```

b) On peut alors tester la classe de la commande en allant chercher le byte correspondant au moyen de

buffer[ISO7816.OFFSET_CLA]

Ainsi, pour reconnaître la commande APDU SELECT, on peut programmer :

```
if ((buffer[ISO7816.OFFSET_CLA] == 0) &&
    (buffer[ISO7816.OFFSET_INS] == (byte)(0xA4))) ...
```

Cependant, si il ne s'agit que de distinguer que l'on a affaire à la commande SELECT, il suffit d'utiliser la méthode

protected final boolean selectingApplet()

qui fait le travail à notre place ... Dans l'affirmative, la méthode process() ne fera rien.

c) Il faut ensuite vérifier que la commande reçue est bien une commande connue de l'applet, donc de sa classe

```
if (buffer[ISO7816.OFFSET_CLA] != PorteMonnaie_CLA) ...
```

Si ce n'est pas le cas, la commande n'est pas supportée. Comment le signaler ? En envoyant une exception qui permettra au JCRC de fabriquer une réponse APDU avec un status word qui est le reason code de l'exception : ceci s'obtient avec la méthode de la classe ISOException :

public static void throwIt (short sw)

d) Si il s'agit bien d'une commande de l'applet, il reste à la reconnaître et à enclencher le traitement correspondant :

```
switch (buffer[ISO7816.OFFSET_INS])
{
    case CREDIT: credit(apdu); return;
    ...
}
```

chaque méthode de traitement (comme ici credit()) restant à écrire ... Donc :

appletPorteMonnaie.java (5)

```

package eCommerce;
import javacard.framework.*;
public class appletPorteMonnaie extends Applet
{
    /* Définition des constantes de classe */
    ...
    /* Définition des variables d'instance */
    ...
    /* Constructeur */
    ...
    /* Point d'entrée de l'applet */
    ...
    /* En cas de sélection */
    ...
    /* Traitement d'une commande APDU */
    public void process(APDU apdu)
    {
        byte[] buffer = apdu.getBuffer();

        // check SELECT APDU command
        if (selectingApplet())
            return;

        /* buffer[ISO7816.OFFSET_CLA] =
           (byte)(buffer[ISO7816.OFFSET_CLA] & (byte)0xFC);
        if ((buffer[ISO7816.OFFSET_CLA] == 0) &&
            (buffer[ISO7816.OFFSET_INS] == (byte)(0xA4)))
        return; */

        // command of the class of the applet ?
        if (buffer[ISO7816.OFFSET_CLA] != PorteMonnaie_CLA)
            ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
        // Ok : which command ?
        switch (buffer[ISO7816.OFFSET_INS])
        {
            case GET_BALANCE: getBalance(apdu);
                return;
            case DEBIT: debit(apdu);
                return;
            case CREDIT: credit(apdu);
                return;
            case VERIFY: verify(apdu);
                return;
            default: ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
        }
    }
}

```

12.9 L'envoi et la réception des données

Nous en arrivons ainsi aux méthodes de l'applet qui correspondent à ses fonctionnalités. Cependant, auparavant, outre la logique interne, deux opérations sont à bien définir.

a) Dans certains cas, des données sont à lire et la taille de ces données se trouve dans le 5^{ème} byte disponible. Or, pour rappel, ces données ne se trouvent pas dans le buffer. Mais on peut se les procurer au moyen de la méthode de la classe APDU

```
public short setIncomingAndReceive() throws APDUEException
```

Cette méthode

- ◆ place le JCRC en mode de réception (on se souviendra en effet que la communication carte-CAD est half duplex);
- ◆ indique au JCRC que le 5^{ème} byte du buffer donne le nombre de bytes à recevoir;
- ◆ demande au JCRC de recevoir ces bytes et de les placer dans le buffer APDU à partir de la position OFFSET_CDATA – autrement dit, à la suite des bytes qui s'y trouvent déjà;
- ◆ retourne le nombre de bytes lus.

Dans la plupart des cas, la taille du buffer APDU sera suffisante. Mais si ce n'est pas le cas (on peut le savoir en comparant le nombre de bytes lus au nombre de bytes prévus dans le 5^{ème} byte), on appelle alors autant de fois que nécessaire une autre méthode de la classe APDU :

```
public short receiveBytes(short bOff) throws APDUEException
```

b) Dans d'autres cas, des données sont à envoyer vers l'hôte. On s'en doute, il faut d'abord placer le JCRC en mode d'envoi au moyen de la méthode

```
public short setOutgoing() throws APDUEException
```

La valeur rentrée est le nombre de bytes attendus par l'hôte en réponse à la commande qu'il a envoyée. Evidemment, il se peut que l'applet n'envoie pas exactement ce nombre; il faut donc préciser le nombre exact de bytes effectivement envoyés (car l'hôte suppose que c'est 0 par défaut !) au moyen de la méthode

```
public void setOutgoingLength (short len) throws APDUEException
```

Enfin, on peut envoyer les données dans le buffer APDU au moyen de la méthode

```
public void sendBytes(short bOff, short len) throws APDUEException
```

J'oubliais ;-) : tout ceci peut se faire avec la seule méthode

```
public void setOutgoingAndSend(short bOff, short len) throws APDUEException
```

12.10 Les méthodes de traitement des commandes

Il s'agit donc ici des méthodes propres à l'applet et qui correspondent à ses fonctionnalités.

a) la vérification du PIN

Une fois le PIN lu, il suffit de le comparer au PIN connu au moyen de la méthode définie dans l'interface PIN et implémentée dans la classe OwnerPIN :

```
public boolean check(byte[] pin, short offset, byte length)
```

En cas de succès, le flag de validation est positionné et le nombre d'essais possibles est initialisé au maximum. Donc :

appletPorteMonnaie.java (6)

```
package eCommerce;

import javacard.framework.*;

public class appletPorteMonnaie extends Applet
{
    /* Définition des constantes de classe */
    ...
    /* Définition des variables d'instance */
    ...
    /* Constructeur */
    ...
    /* Point d'entrée de l'applet */
    ...
    /* En cas de sélection */
    ...
    /* Traitement d'une commande APDU */
    ...
    /* Vérification du PIN */
    private void verify(APDU apdu)
    {
        byte[] buffer = apdu.getBuffer();
        // lecture du PIN entré
        byte byteRead = (byte)(apdu.setIncomingAndReceive());

        if ( pin.check(buffer, ISO7816.OFFSET_CDATA, byteRead) == false )
            ISOException.throwIt (SW_VERIFICATION_FAILED);
    }
}
```

b) le rechargement du porte-monnaie

Tout comme pour les opérations de paiement et de solde, il sera d'abord vérifié que le code PIN a été introduit avec succès au moyen de la méthode de l'interface PIN :

```
public boolean isValidated()
```

La suite est logique : on lit le montant de la recharge qui figure dans la commande APDU CREDIT et on met à jour le solde, en vérifiant

- ◆ que ce montant est valide (c'est-à-dire pas trop grand ou négatif);
- ◆ qu'avec une telle recharge, le solde ne dépasse pas la limite de capacité admise du porte-monnaie.

appletPorteMonnaie.java (7)

```
package eCommerce;
import javacard.framework.*;
public class appletPorteMonnaie extends Applet
{
    /* Définition des constantes de classe */
    ...
    /* Définition des variables d'instance */
    ...
    /* Constructeur */
    ...
    /* Point d'entrée de l'applet */
    ...
    /* En cas de sélection */
    ...
    /* Traitement d'une commande APDU */
    ...
    /* Vérification du PIN */
    ...
    /* Recharge du porte-monnaie */
    private void credit(APDU apdu)
    {
        if ( ! pin.isValidated() )
            ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
        byte[] buffer = apdu.getBuffer();
        // récupération de la longueur des données
        byte numBytes = buffer[ISO7816.OFFSET_LC];
        // lecture de ces données
        byte byteRead = (byte)(apdu.setIncomingAndReceive());
        // a-t-on tout lu ?
        if ( ( numBytes != 1 ) || (byteRead != 1) )
            ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
        // montant de la recharge
        byte montant = buffer[ISO7816.OFFSET_CDATA];
        if ( ( montant > MAX_TRANSACTION_AMOUNT ) || ( montant < 0 ) )
            ISOException.throwIt(SW_INVALID_TRANSACTION_AMOUNT);
        if ( (short)(solde + montant) > MAX_BALANCE )
            ISOException.throwIt(SW_EXCEED_MAXIMUM_BALANCE);
        solde = (short)(solde + montant);
    }
}
```

c) le paiement

Cette méthode est bien sûr tout à fait analogue à la précédente :

appletPorteMonnaie.java (8)

```
package eCommerce;

import javacard.framework.*;

public class appletPorteMonnaie extends Applet
{
    /* Définition des constantes de classe */
    ...
    /* Définition des variables d'instance */
    ...
    /* Constructeur */
    ...
    /* Point d'entrée de l'applet */
    ...
    /* En cas de sélection */
    ...
    /* Traitement d'une commande APDU */
    ...
    /* Vérification du PIN */
    ...
    /* Recharge du porte-monnaie */
    ...
    /* Paiement */
    private void debit(APDU apdu)
    {
        if ( ! pin.isValidated() )
            ISOException.throwIt (SW_PIN_VERIFICATION_REQUIRED);
        byte[] buffer = apdu.getBuffer();
        byte numBytes = (byte)(buffer[ISO7816.OFFSET_LC]);
        byte byteRead = (byte)(apdu.setIncomingAndReceive());
        if ( ( numBytes != 1 ) || (byteRead != 1) )
            ISOException.throwIt (ISO7816.SW_WRONG_LENGTH);

        // montant du paiement
        byte montant = buffer[ISO7816.OFFSET_CDATA];
        if ( ( montant > MAX_TRANSACTION_AMOUNT ) || ( montant < 0 ) )
            ISOException.throwIt (SW_INVALID_TRANSACTION_AMOUNT);
        if ( (short)( solde - montant ) < (short)0 )
            ISOException.throwIt(SW_NEGATIVE_BALANCE);
        solde = (short) (solde - montant);
    }
}
```

d) la visualisation du contenu du porte-monnaie

Cette fois, il n'y a pas de données à lire dans la commande, mais il y a une réponse à retourner : on utilisera donc les méthodes liées à l'envoi de bytes.

appletPorteMonnaie.java (9)

```
package eCommerce;

import javacard.framework.*;

public class appletPorteMonnaie extends Applet
{
    /* Définition des constantes de classe */
    ...
    /* Définition des variables d'instance */
    ...
    /* Constructeur */
    ...
    /* Point d'entrée de l'applet */
    ...
    /* En cas de sélection */
    ...
    /* Traitement d'une commande APDU */
    ...
    /* Vérification du PIN */
    ...
    /* Recharge du porte-monnaie */
    ...
    /* Paiement */
    ...
    /* Visualisation du solde */
    private void getBalance(APDU apdu)
    {
        byte[] buffer = apdu.getBuffer();

        // nombre de bytes prévus par l'hôte pour la réponse
        short le = apdu.setOutgoing();
        if ( le < 2 ) ISOException.throwIt (ISO7816.SW_WRONG_LENGTH);
        apdu.setOutgoingLength((byte)2);

        /* écriture dans le buffer APDU */
        // assembler-like :-(
        // buffer[0] = (byte)(solde >> 8); buffer[1] = (byte)(solde & 0xFF);
        // moins douloureux :
        Util.setShort (buffer, (short)0, solde);
        // écriture dans le buffer APDU
        apdu.sendBytes((short)0, (short)2);
    }
}
```

On aura remarqué l'utilisation d'une méthode de la classe Util : cette classe, qui ne comporte que des méthodes de classe, est destinée à fournir des outils de traitement des tableaux de bytes et des entiers short. Ici, ce qui nous intéresse est :

```
public static final short setShort(byte[] bArray, short bOff, short sValue)
    throws TransactionException
```

- cette méthode place les deux bytes constitutifs du short dans le tableau cible.

12.11 La méthode de désélection

Lorsqu'une applet est sélectionnée, l'applet active doit être désactivée (puisque une Java card n'est pas multithread). La méthode

```
public void deselect()
```

est appelée par le JCRC dans ce cas, ce qui permet de réaliser les opérations de terminaison et de nettoyage de l'applet – une espèce de destructeur. Dans notre cas, la seule opération consiste à invalider le PIN courant, ce qui se fait avec la méthode de PIN :

```
public void reset()
```

qui remet le flag de validation à false. Et pour terminer :

appletPorteMonnaie.java (10)

```
package eCommerce;
import javacard.framework.*;

public class appletPorteMonnaie extends Applet
{
    /* Définition des constantes de classe */
    ...
    /* Définition des variables d'instance */
    ...
    /* Constructeur */
    ...
    /* Point d'entrée de l'applet */
    ...
    /* En cas de sélection */
    ...
    /* Traitement d'une commande APDU */
    ...
    /* Vérification du PIN */
    ...
    /* Recharge du porte-monnaie */
    ...
    /* Paiement */
    ...
    /* Visualisation du solde */
    ...
}
```

```

/* Désélection */
public void deselect()
{
    pin.reset();
}

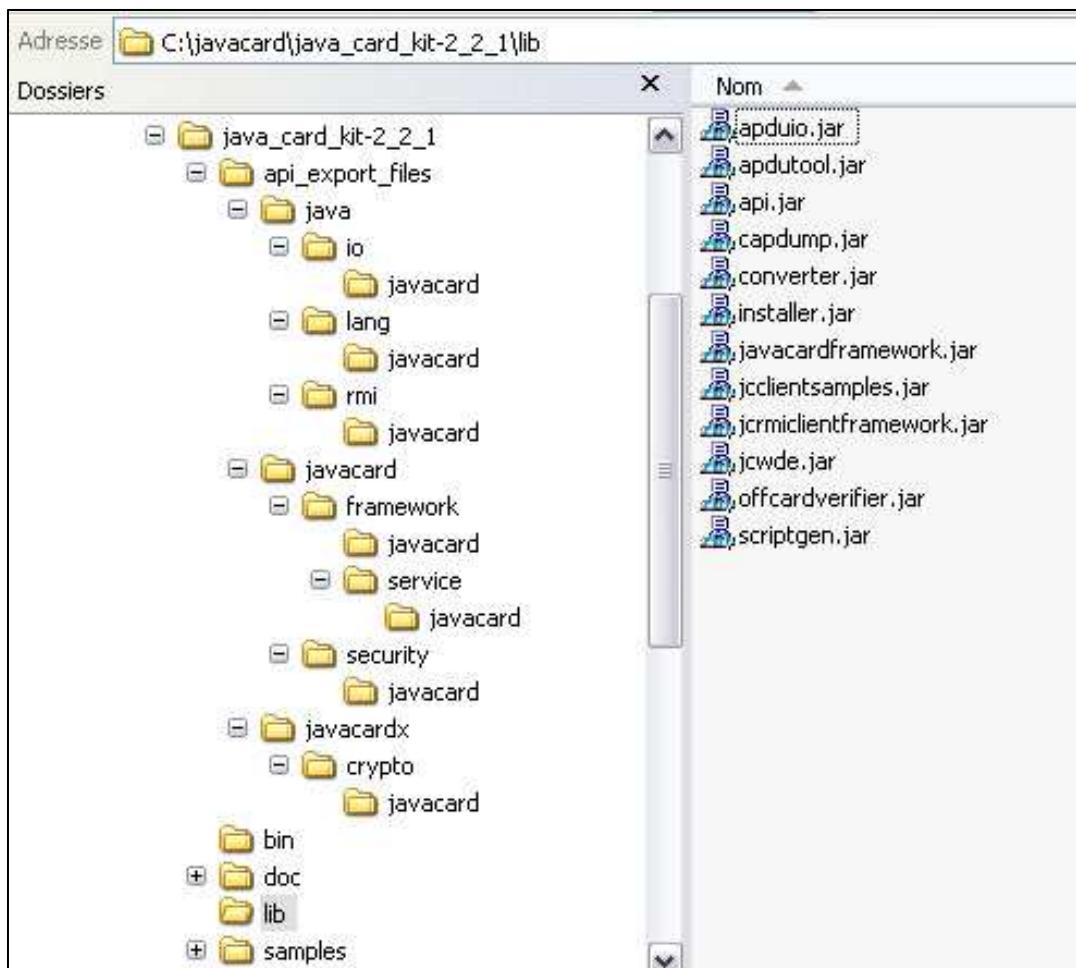
```

Cela a pris un certain temps, mais voici notre applet prêté ☺ ! Reste à la faire fonctionner ...

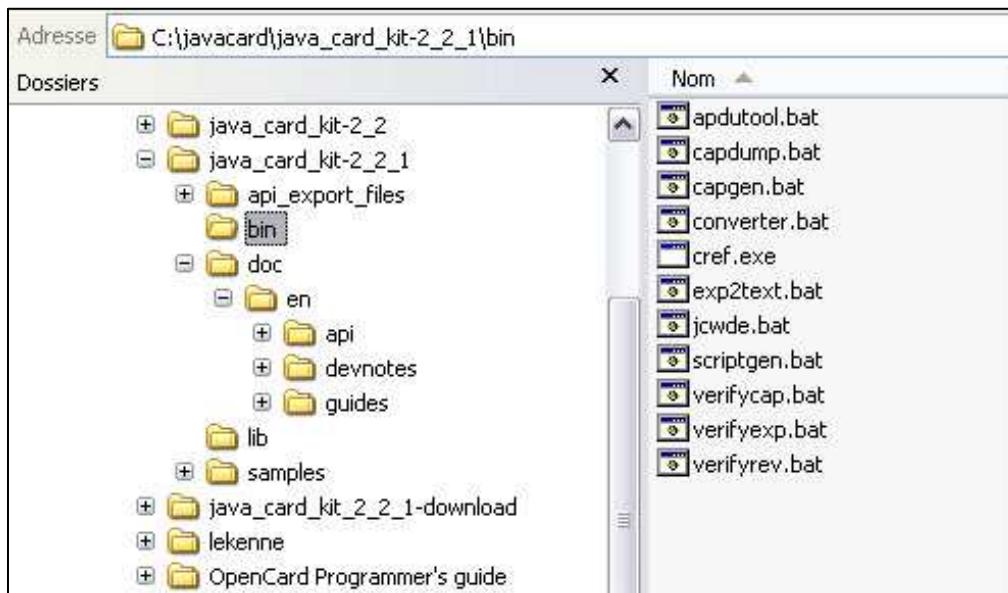
13. Les outils de développement Java

Nous allons travailler ici avec une plate-forme J2SE ou J2EE version 1.3 et suivantes (mais les versions postérieures à 1.4 n'intègrent tous les outils nécessaires de la même manière). Les éléments additionnels nécessaires à l'utilisation des Java cards sont les suivants.

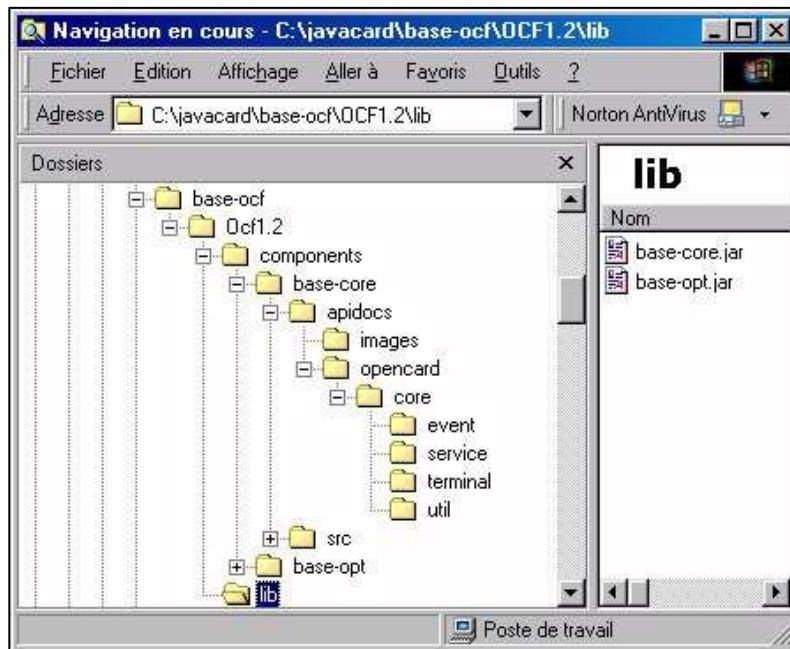
a) le kit de développement des Java cards; le fichier compressé (à destination des machines Windows) `java_card_kit-2_2_01-win-gl.zip`, que l'on peut télécharger sur le site de Sun, est à décompresser, par exemple dans un répertoire nommé `javacard`. On y obtient ainsi un répertoire **java_card_kit-2_2-1** :



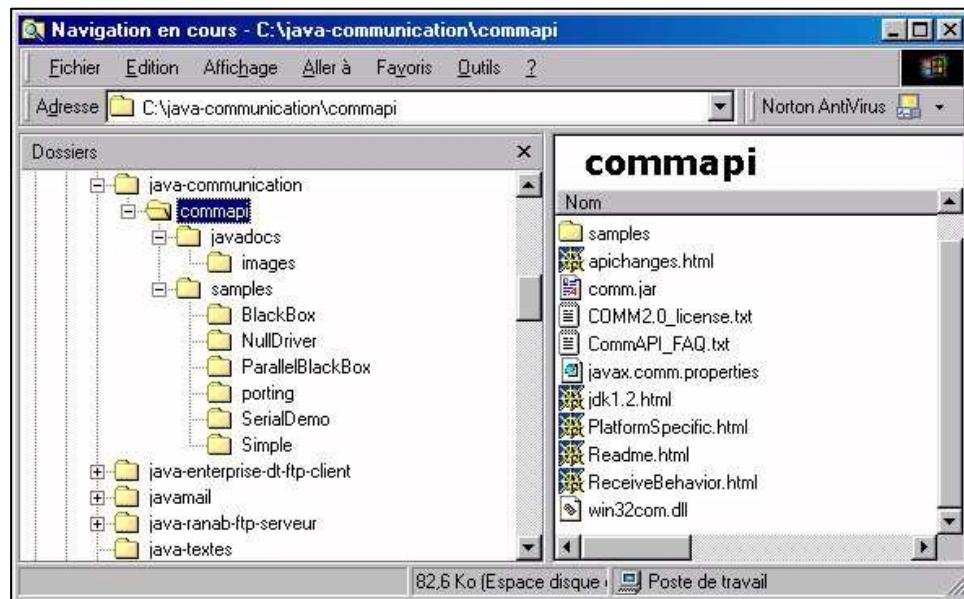
Outre les fichiers jar prévisibles, comportant les packages de développement pour Java card, on trouve aussi dans le répertoire les utilitaires de mise au point de l'applet : convertisseur en fichier CAP, émulateurs de CAD, interpréteur et générateur de commandes APDU, etc :



b) le framework (donc les APIs) de l'OpenCard; le fichier compressé BaseOCF.zip, que l'on peut télécharger sur le site de l'OpenCard, est également à décompresser, par exemple dans le même répertoire javacard (son utilisation sera expliquée plus loin). On y obtient ainsi un répertoire **Ocf1.2** :



c) éventuellement, mais elles sont vouées à la disparition, les APIs de communication série et parallèle – en effet, le lecteur de cartes est parfois encore (rarement) connecté à la machine hôte sur le port série (au lieu du plus moderne port USB); le fichier compressé (à destination des machines Windows) javacom20-win32.zip, que l'on peut télécharger sur le site de Sun, est à décompresser, par exemple dans un répertoire nommé java-communication. On y obtient ainsi un répertoire **commapi** :



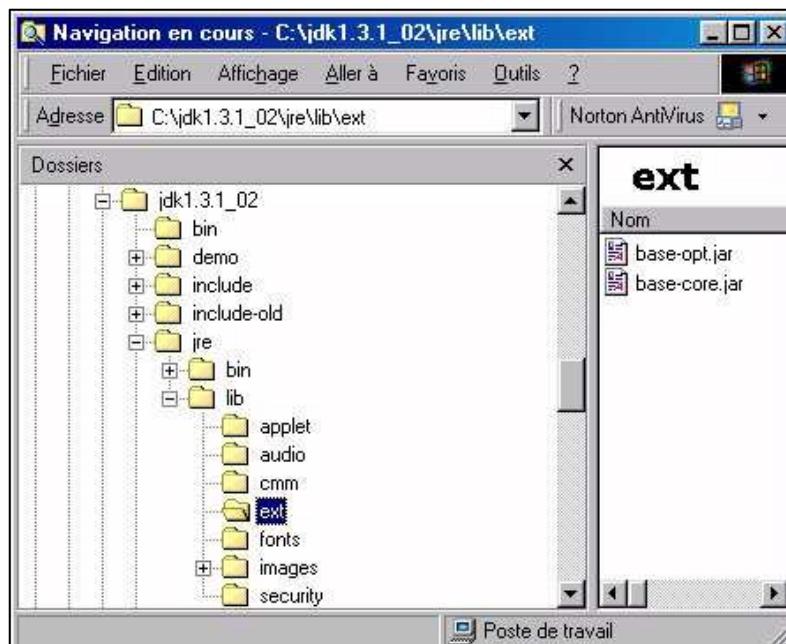
d) les variables d'environnement JC_HOME (chemin pour accéder au kit Java Card) et JAVA_HOME (chemin pour accéder au JDK); pour la facilité, on peut aussi ajouter au PATH la chemin des exécutables Java; il faut aussi compléter le CLASSPATH des chemins permettant de trouver les jar de l'API Java card et de l'API Java communication; c'est le rôle de fichier batch suivant :

jc_env.bat

```
set JC_HOME=C:\javacard\java_card_kit-2_2_1
set JAVA_HOME=C:\j2sdk1.4.2_03
set PATH=%JC_HOME%\bin;%PATH%
set CLASSPATH=C:\javacard\java_card_kit-2_2_1\lib\javacardframework.jar;C:\j2sdk1.4.2_03\jre\lib\*.jar;%CLASSPATH%
```

Pour la facilité, on peut encore copier une bonne fois pour toutes :

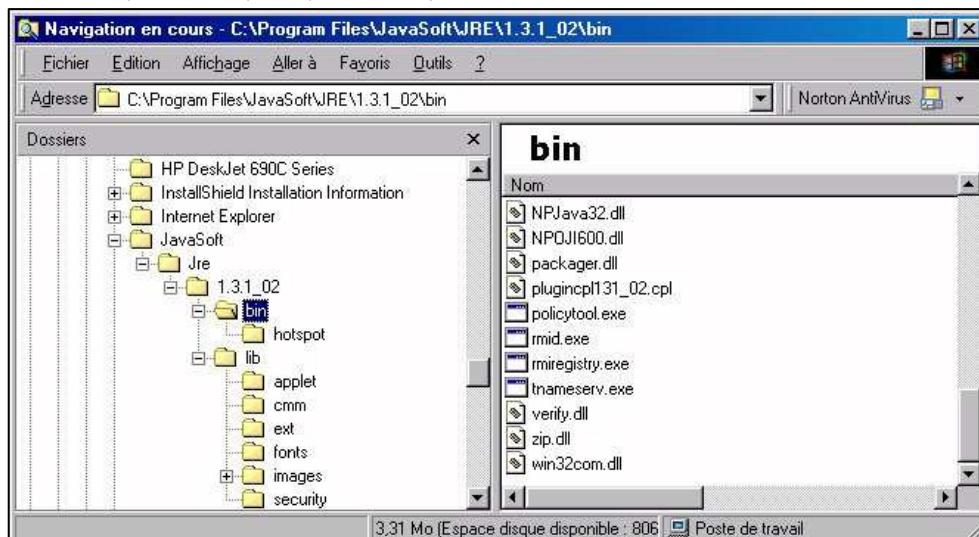
- ◆ les **fichiers jar de l'OCF** dans le répertoire `\jre\lib\ext` du JDK – par exemple :



- ♦ si nécessaire, les fichiers de communication série orientés Java **comm.jar** et **javax.comm.properties** dans le répertoire C:\Program Files\JavaSoft\JRE\1.3.1_02\lib



- ♦ le fichiers de communication série pour Windows **win32com.dll** dans le répertoire C:\Program Files\JavaSoft\JRE\1.3.1_02\bin



14. La mise au point de l'applet porte-monnaie

14.1 La compilation

Au préalable, il s'agit de compiler notre applet appletPorteMonnaie.java :

```

package eCommerce;
import javacard.framework.*;
/*
 * @author Vilvens
 * @version
 */
public class appletPorteMonnaie extends Applet
{
    /* Définition des constantes de classe */
}

```

Nous obtenons ainsi un fichier appletPorteMonnaie.class dans un répertoire eCommerce.

14.2 La conversion en fichier CAP

Nous allons à présent utiliser le convertisseur évoqué au paragraphe 7. En fait, il s'agit d'un fichier batch qui utilise une classe Java :

converter.bat

```
@echo off
REM
REM Copyright © 2002 Sun Microsystems, Inc. All rights reserved.
REM SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
REM

if "%OS%" == "Windows_NT" setlocal
if not "%JAVA_HOME%" == "" goto check_tool
    echo Please set the JAVA_HOME environment variable.
    goto end

:check_tool
if not "%JC_HOME%" == "" goto doit
    echo Please set the JC_HOME environment variable.
    goto end

:doit
set
_CLASSES=%JC_HOME%\lib\apduio.jar;%JC_HOME%\lib\apdutool.jar;%JC_HOME%\lib\jc
wde.jar;%JC_HOME%\lib\converter.jar;%JC_HOME%\lib\scriptgen.jar;%JC_HOME%\lib\offca
rdverifier.jar;%JC_HOME%\lib\api.jar;%JC_HOME%\lib\capdump.jar;%JC_HOME%\samples\
classes;%CLASSPATH%;

%JAVA_HOME%\bin\java -classpath %_CLASSES% com.sun.javacard.converter.Converter
%*
goto end

:end
if "%OS%" == "Windows_NT" endlocal
```

Ce convertisseur s'utilise donc en ligne de commandes selon la syntaxe :

converter [options] <nom du package> <aid du package> <version x.y>

avec donc un certain nombre d'options dont nous retiendrons :

-out : pour spécifier les fichiers attendus en sortie (leur type est à écrire en majuscules); ici, il s'agira de CAP, EXP et JCA;

-exportpath : pour indiquer où trouver les fichiers d'export du kit des Java Cards

-applet : pour donner l'AID de l'applet et du package avec la version selon la syntaxe :

applet <AID applet> <nom complet applet>

-classdir : pour fournir le chemin vers le fichier .class de l'applet, si l'on ne se trouve pas dans le répertoire convenable.

-config : permet d'utiliser les options mémorisées dans un fichier texte, qui doit avoir l'extension .opt.

Nous allons ici exploiter cette dernière possibilité avec le fichier suivant, qui contient même tous les arguments passés au convertisseur :

appletPorteMonnaie.opt

```
-out EXP JCA CAP  
-exportpath C:\javacard\java_card_kit-2_2_1\api_export_files  
-applet 0x00:0x00:0x00:0x00:0x01:0x01 eCommerce.appletPorteMonnaie  
eCommerce 0x00:0x00:0x00:0x00:0x01 1.0
```

Si nous nous plaçons dans le répertoire père du répertoire eCommerce du package de l'applet, nous pouvons lancer le convertisseur par :

```
C:\java-netbeans-application\AppletPorteMonnaie\build\classes>converter -config  
appletPorteMonnaie.opt
```

ou

```
C:\java- netbeans-application\AppletPorteMonnaie\build\classes>converter -applet  
0x00:0x00:0x00:0x00:0x01:0x01 eCommerce.appletPorteMonnaie eCommerce  
0x00:0x00:0x00:0x00:0x01 1.0
```

ce qui donne en écho :

```
Java Card 2.2.1 Class File Converter, Version 1.3  
Copyright 2003Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.  
  
Conversion completed with 0 errors and 0 warnings.
```

En pratique, le convertisseur a créé dans le répertoire eCommerce un **répertoire javacard** dans lequel il a généré, si on l'a demandé, trois fichiers eCommerce.cap, eCommerce.exp et eCommerce.jca.

14.3 L'émulateur de lecteur de cartes

Sun fournit, parmi les divers outils de mise au point des applications Java card, un émulateur d'environnement de carte à puces – c'est fort pratique pour la mise au point préalable des applets. Cet émulateur s'appelle **C-JCRE** et est présenté comme un exécutable appelé cref.exe :

```
C:\javacard>cref  
Java Card 2.2.1 Reference Implementation Simulator (version 0.41)  
Copyright 2003Sun Microsystems, Inc. All rights reserved.
```

Memory configuration

Type	Base	Size	Max Addr
RAM	0x0	0x500	0x4ff
ROM	0x1000	0x8000	0x8fff
E2P	0x9020	0x3fe0	0xcfff
ROM Mask size = 0x4a31 = 18993 bytes			
Highest ROM address in mask = 0x5a30 = 23088 bytes			
Space available in ROM = 0x35cf = 13775 bytes			

Mask has now been initialized for use

On peut constater que l'émulateur attend ses instructions sur le port TCP **9025** de la machine hôte. On peut aussi le lancer avec comme option de ligne de commande :

- i <fichier d'entrée> : pour placer les données du fichier dans l'EEPROM simulée;
- o <fichier de sortie> : pour sauver le contenu de l'EEPROM dans le fichier précisé et ainsi rendre effectivement les données persistantes, comme sur une vraie carte.

Cet émulateur utilisera les fichiers cap et connaît donc d'emblée les AIDs des applets et des packages. Il attend essentiellement des commandes APDU qui lui parviendront au moyen de l'outil suivant ...

14.4 Les scripts APDU

Un script APDU est essentiellement un fichier contenant des commandes APDU sous forme hexadécimale; il a l'extension **.scr**. Plus précisément,

- ◆ il commence toujours par "powerup" (pour alimenter la carte en courant) et se termine toujours par "powerdone" (pour couper l'alimentation);
- ◆ une ligne débutant par "0x" est supposée être un commande au format APDU;
- ◆ une ligne débutant par "//" est un commentaire;
- ◆ on peut réaliser un affichage sur la sortie standard par la commande "echo" suivi du texte entre guillemets;
- ◆ chaque ligne doit se terminer par ";".

Par exemple, pour sélectionner l'installer d'applet dont l'AID est 0xA0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01, nous écrirons une commande SELECT dont nous savons (voir paragraphe 3.2) que le header est du type

0x00 0xA4 0x04 0x00

Le 5^{ème} byte de la commande contiendra la longueur de l'AID (soit ici 9). Le fichier script correspondant aura donc l'aspect suivant :

selectInstaller.scr

```
powerup;

echo "Select the installer applet";
0x00 0xA4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01 0x7F;
// 9000 = success

powerdown;
```

L'utilitaire **apdutool.bat** sait utiliser un tel fichier script pour envoyer les commandes au JCRC (réel ou simulé par C-JCRC) sur la socket sur laquelle le JCRC attend et récupérer les codes de retour correspondants.

14.5 L'installation de l'applet sur la carte

Il s'agit en fait de réaliser deux opérations : le téléchargement de l'applet sous une forme utilisable pour la Java card et son instantiation.

Le package contenant notre applet se trouve pour l'instant dans un fichier CAP. Mais la carte ne peut recevoir avec succès que des commandes APDUs. C'est le rôle d'un autre utilitaire fourni dans le kit Java card, utilitaire nommé **scriptgen**, de générer à partir d'un fichier CAP un fichier SCR contenant la séquence des commandes APDU correspondant aux fonctionnalités de notre applet. En fait, ce fichier batch exécute la méthode main() de la classe Main se trouvant dans le fichier scrptgen.jar – on peut le voir dans le texte du fichier bat :

```
scriptgen.bat
@echo off
REM
REM Copyright © 2002 Sun Microsystems, Inc. All rights reserved.
REM SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
REM

if "%OS%" == "Windows_NT" setlocal
if not "%JAVA_HOME%" == "" goto check_tool
    echo Please set the JAVA_HOME environment variable.
    goto end

:check_tool
if not "%JC_HOME%" == "" goto doit
    echo Please set the JC_HOME environment variable.
    goto end

:doit
set
_CLASSES=%JC_HOME%\lib\apduio.jar;%JC_HOME%\lib\apdutool.jar;%JC_HOME%\lib\jcwde.jar;%JC_HOME%\lib\converter.jar;%JC_HOME%\lib\scriptgen.jar;%JC_HOME%\lib\offcardverifier.jar;%JC_HOME%\lib\api.jar;%JC_HOME%\lib\capdump.jar;%JC_HOME%\samples\classes;%CLASSPATH%;

%JAVA_HOME%\bin\java -classpath %_CLASSES% com.sun.javacard.scriptgen.Main
%*
goto end

:end
if "%OS%" == "Windows_NT" endlocal
```

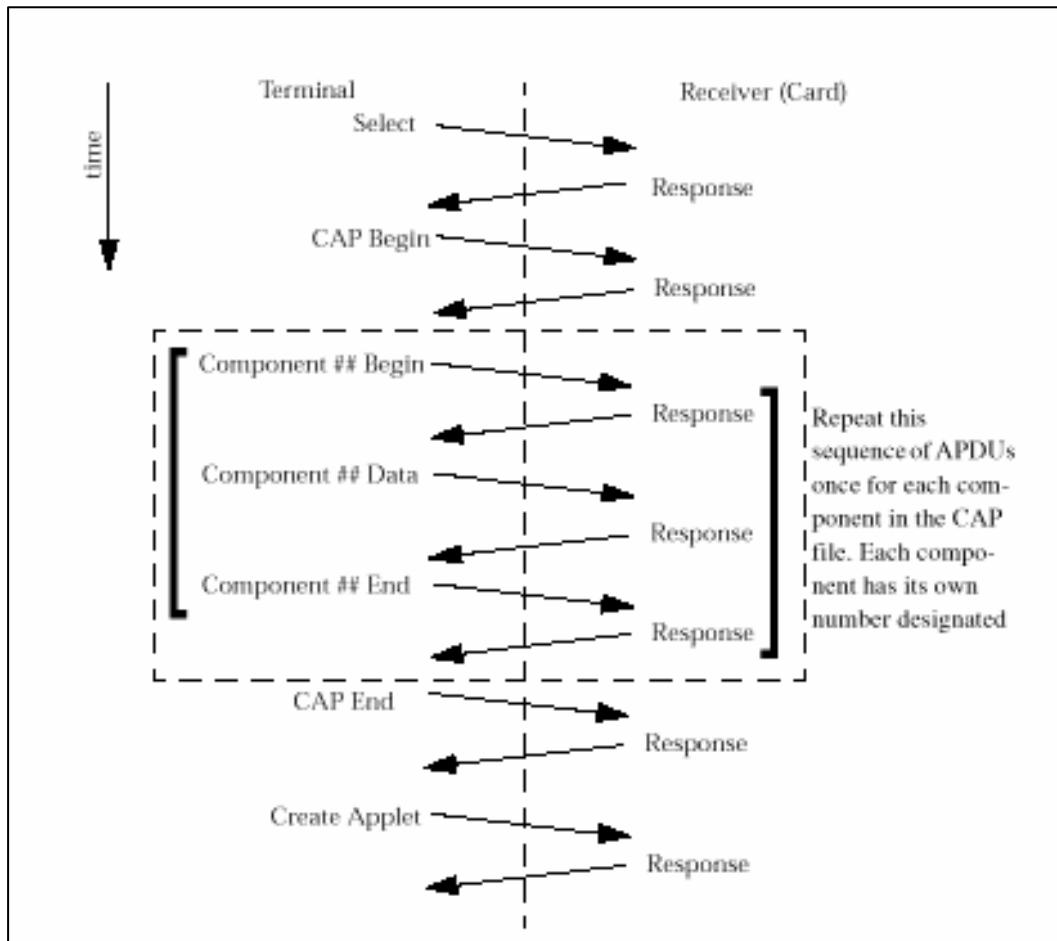
L'utilitaire est utilisé selon la syntaxe

scriptgen [options] <fichier CAP>

où les options qui nous intéressent sont seulement :

- o <fichier de sortie> : pour sauver le résultat de la conversion dans le fichier précisé;
 - package <nom du package> pour préciser le nom du fichier contenant le fichier CAP.
-

Le protocole d'installation peut être vu comme suit :



L'appel de l'utilitaire scriptgen :

```
C:\java-forte-application\AppletPorteMonnaie\eCommerce\javacard> scriptgen -o charge.scr
eCommerce.cap
Java Card 2.2.1 Script Generator, Version 1.3
Copyright 2003 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.
APDU script file for CAP file download generated.
```

fournit un fichier presqu'utilisable : il faut simplement

- a) ajouter au début "PowerUp" (si il n'a pas été généré automatiquement) et la commande APDU de sélection de l'installateur on-card (voir paragraphe 13.4);
- b) ajouter en fin de fichier la commande d'instanciation proprement dite (sinon, l'applet aura été téléchargée, mais pas instanciée, et ne sera donc pas utilisable), qui fournit comme données l'AID de l'applet à installer :

0x80 0xB8 0x00 0x00 0x08 0x06 0x00:0x00:0x00:0x01:0x01 0x00 0x7F;

le byte complémentaire 0x00 étant un paramètre inutile pour nous (il est d'ailleurs supprimé dans la nouvelle version 2-2-2 !), mais qui doit exister dans la version que nous utilisons ici (fallait le savoir ☺ ...).

Cette commande se place entre les commandes générées :

- ♦ de lancement de l'installation, commande sans paramètres et sans données :

0x80 0xB0 0x00 0x00 0x00 0x00 0x7F;

- ♦ de fin de l'installation, commande sans paramètres et sans données :

0x80 0xBA 0x00 0x00 0x00 0x00 0x7F;

c) ajouter à la fin "PowerDown" (si il n'a pas été généré automatiquement).

Donc finalement :

charge.scr

powerup;

echo "Selection de l'installer d'applet";

0x00 0xA4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01 0x7F;

0x80 0xB0 0x00 0x00 0x00 0x7F;

// eCommerce/javacard/Header.cap

0x80 0xB2 0x01 0x00 0x00 0x7F;

0x80 0xB4 0x01 0x00 0x12 0x01 0x00 0x0F 0xDE 0xCA 0xFF 0xED 0x01 0x02 0x04 0x00 0x01 0x05 0x00 0x00 0x00 0x01 0x7F;

0x80 0xBC 0x01 0x00 0x00 0x7F;

// eCommerce/javacard/Directory.cap

0x80 0xB2 0x02 0x00 0x00 0x7F;

...

// eCommerce/javacard/Import.cap

0x80 0xB2 0x04 0x00 0x00 0x7F;

0x80 0xB4 0x04 0x00 0x0E 0x04 0x00 0x0B 0x01 0x02 0x01 0x07 0xA0 0x00 0x00 0x00 0x62 0x01 0x01 0x7F;

0x80 0xBC 0x04 0x00 0x00 0x7F;

// eCommerce/javacard/Applet.cap

0x80 0xB2 0x03 0x00 0x00 0x7F;

0x80 0xB4 0x03 0x00 0x0D 0x03 0x00 0x0A 0x01 0x06 0x00 0x00 0x00 0x00 0x01 0x02 0x00 0x34 0x7F;

0x80 0xBC 0x03 0x00 0x00 0x7F;

// eCommerce/javacard/Class.cap

0x80 0xB2 0x06 0x00 0x00 0x7F;

0x80 0xB4 0x06 0x00 0x15 0x06 0x00 0x12 0x00 0x80 0x03 0x02 0x00 0x01 0x04 0x04

```
0x00 0x00 0x01 0x9E 0xFF 0xFF 0x00 0x42 0x00 0x4F 0x7F;  
0x80 0xBC 0x06 0x00 0x00 0x7F;  
  
// eCommerce/javacard/Method.cap  
0x80 0xB2 0x07 0x00 0x00 0x7F;  
0x80 0xB4 0x07 0x00 0x20 0x07 0x01 0xA6 0x00 0x05 0x40 0x18 0x8C 0x00 0x03 0x18  
0x8F 0x00 0x05 0x3D 0x06 0x07 0x8C 0x00 0x02 0x87 0x00 0x19 0x03 0x04 0x38 0x19  
0x04 0x05 0x38 0x19 0x05 0x7F;  
...  
  
// eCommerce/javacard/StaticField.cap  
0x80 0xB2 0x08 0x00 0x00 0x7F;  
0x80 0xB4 0x08 0x00 0x0D 0x08 0x00 0x0A 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00  
0x00 0x00 0x7F;  
0x80 0xBC 0x08 0x00 0x00 0x7F;  
  
// eCommerce/javacard/ConstantPool.cap  
0x80 0xB2 0x05 0x00 0x00 0x7F;  
0x80 0xB4 0x05 0x00 0x20 0x05 0x00 0x66 0x00 0x19 0x02 0x00 0x00 0x00 0x02 0x00  
0x00 0x01 0x06 0x80 0x09 0x00 0x06 0x80 0x03 0x00 0x03 0x80 0x09 0x08 0x01 0x80  
0x09 0x00 0x03 0x80 0x03 0x7F;  
...  
  
// eCommerce/javacard/RefLocation.cap  
0x80 0xB2 0x09 0x00 0x00 0x7F;  
0x80 0xB4 0x09 0x00 0x20 0x09 0x00 0x3C 0x00 0x0F 0x12 0x12 0x0A 0x17 0x6A 0x13  
0x41 0x11 0x05 0x05 0x41 0x0E 0x05 0x20 0x0F 0x00 0x29 0x05 0x04 0x06 0x1A 0x08  
...  
0x80 0xBC 0x09 0x00 0x00 0x7F;  
  
0x80 0xBA 0x00 0x00 0x00 0x7F;  
  
0x80 0xB0 0x00 0x00 0x00 0x7F;  
0x80 0xB8 0x00 0x00 0x08 0x06 0x00 0x00 0x00 0x00 0x01 0x01 0x00 0x7F;  
0x80 0xBA 0x00 0x00 0x00 0x7F;  
  
powerdown;
```

Nous disposons à présent du matériel nécessaire pour l'installation effective de l'applet, provisoirement avec une Java card émulée. Nous avons donc besoin de deux fenêtres DOS, chacune avec les variables d'environnement habituelles :

a) la première va nous permettre de lancer l'émulateur de carte C-JCRE, avec l'option -o pour lui désigner un fichier qui lui servira d'EEPROM (appelons-le finement eeprom) :

```
C:\javacard>cref -o eeprom  
Java Card 2.2.1 C Reference Implementation Simulator (version 0.41)  
32-bit Address Space implementation - with cryptography support  
Copyright 2003 Sun Microsystems, Inc. All rights reserved.
```

Memory configuration

Type	Base	Size	Max Addr
RAM	0x0	0x500	0x4ff
ROM	0x1000	0x8000	0x8fff
E2P	0x9020	0x3fe0	0xcfff

ROM Mask size = 0x4a31 = 18993 bytes
 Highest ROM address in mask = 0x5a30 = 23088 bytes
 Space available in ROM = 0x35cf = 13775 bytes

EEPROM will be saved in file "eprom"

Mask has now been initialized for use

b) la deuxième va nous permettre de lancer l'apdutool pour lui faire exécuter notre script d'installation, en lui précisant dans une option -o un fichier résultat qui contiendra les commandes envoyées avec leur réponse :

```
C:\java-netbeans-application\AppletPorteMonnaie\eCommerce\javacard>apdutool -o res.log charge.scr
Java Card 2.2.1 APDU Tool, Version 1.3
Copyright 2003 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.
Opening connection to localhost on port 9025.
Connected.
Received ATR = 0x3b 0xf0 0x11 0x00 0xff 0x00
```

La dernière ligne fait référence à l'"**Answer To Reset**" –pour rappel, une information envoyée par la smartcard qui a été réinitialisée ("reset") et mise sous tension.

Le fichier res.log nous informe sur la manière dont le script a été exécuté - pour rappel, un code de retour 9000 signifie "succès" :

res.log	
Selection de l'installer d'applet	
CLA:	00, INS: a4, P1: 04, P2: 00, Lc: 09, a0, 00, 00, 00, 62, 03, 01, 08, 01, Le: 00, SW1: 90, SW2: 00
CLA:	80, INS: b0, P1: 00, P2: 00, Lc: 00, Le: 00, SW1: 90, SW2: 00
CLA:	80, INS: b2, P1: 01, P2: 00, Lc: 00, Le: 00, SW1: 90, SW2: 00
CLA:	80, INS: b4, P1: 01, P2: 00, Lc: 12, 01, 00, 0f, de, ca, ff, ed, 01, 02, 04, 00, 01, 05, 00, 00, 00, 01, Le: 00, SW1: 90, SW2: 00
CLA:	80, INS: bc, P1: 01, P2: 00, Lc: 00, Le: 00, SW1: 90, SW2: 00
CLA:	80, INS: b2, P1: 02, P2: 00, Lc: 00, Le: 00, SW1: 90, SW2: 00
CLA:	80, INS: b4, P1: 02, P2: 00, Lc: 20, 02, 00, 1f, 00, 0f, 00, 1f, 00, 0a, 00, 0b, 00, 66, 00, 12, 01, b7, 00, 0a, 00, 3c, 00, 00, 00, db, 00, 00, 00, 00, 00, 01, Le: 00, SW1: 90, SW2: 00
CLA:	80, INS: b4, P1: 02, P2: 00, Lc: 02, 01, 00, Le: 00, SW1: 90, SW2: 00
...	
CLA:	80, INS: b8, P1: 00, P2: 00, Lc: 08, 06, 00, 00, 00, 00, 01, 01, 00, Le: 06, 00, 00, 00, 00, 01, 01, SW1: 90, SW2: 00
CLA:	80, INS: ba, P1: 00, P2: 00, Lc: 00, Le: 00, SW1: 90, SW2: 00

A ce stade, l'applet est prête à être employée puisque installée dans le fichier eeprom ...

14.6 Le test de l'applet

Nous pouvons à présent rédiger un script APDU qui, après avoir sélectionné notre applet, va la faire fonctionner avec des commandes de test :

testAppletPorteMonnaie.scr

```
powerup;

// Sélection de l'applet
0x00 0xA4 0x04 0x00 0x06 0x00 0x00 0x00 0x00 0x01 0x01 0x7F;
// 90 00 = SW_NO_ERROR

// Vérification du PIN
0xB0 0x20 0x00 0x00 0x04 0x01 0x02 0x03 0x04 0x7F;
//90 00 = SW_NO_ERROR

// Lecture du solde
0xB0 0x50 0x00 0x00 0x00 0x02;
//0x00 0x00 0x00 0x90 0x00 = Balance = 0 and SW_ON_ERROR

// Tentative de débit sur un compte vide
0xB0 0x40 0x00 0x00 0x01 0x64 0x7F;
//0x6A85 = SW_NEGATIVE_BALANCE

// Versement de 100 EUR sur la carte
0xB0 0x30 0x00 0x00 0x01 0x64 0x7F;
//0x9000 = SW_NO_ERROR

// Lecture du solde
0xB0 0x50 0x00 0x00 0x00 0x02;
//0x00 0x64 0x9000 = Balance = 100 and SW_NO_ERROR

// Prélèvement de 50 EUR
0xB0 0x40 0x00 0x00 0x01 0x32 0x7F;
//0x9000 = SW_NO_ERROR

// Lecture du solde
0xB0 0x50 0x00 0x00 0x00 0x02;
//0x00 0x32 0x9000 = Balance = 50 and SW_NO_ERROR

// Versement de 128 EUR
0xB0 0x30 0x00 0x00 0x01 0x80 0x7F;
//0x6A83 = SW_INVALID_TRANSACTION_AMOUNT

// Lecture du solde
0xB0 0x50 0x00 0x00 0x00 0x02;
//0x00 0x32 0x9000 = Balance = 50 and SW_NO_ERROR

// Prélèvement de 51 EUR
0xB0 0x40 0x00 0x00 0x01 0x33 0x7F;
```

```
//0x6A85 = SW_NEGATIVE_BALANC

// Lecture du solde
0xB0 0x50 0x00 0x00 0x00 0x02;
//0x00 0x32 0x9000 = Balance = 50 and SW_NO_ERROR

// Prélèvement de 128 EUR
0xB0 0x40 0x00 0x00 0x01 0x80 0x7F;
//0x6A83 = SW_INVALID_TRANSACTION_AMOUNT

// Lecture du solde
0xB0 0x50 0x00 0x00 0x00 0x02;
//0x00 0x32 0x9000 = Balance = 50 and SW_NO_ERROR

// Lecture du solde avec une longueur erronée
0xB0 0x50 0x00 0x00 0x00 0x01;
//0x6700 = ISO7816.SW_WRONG_LENGTH

// Lecture du solde
0xB0 0x50 0x00 0x00 0x00 0x02;
//0x00 0x32 0x9000 = Balance = 50 and SW_NO_ERROR

powerdown;
```

A nouveau, nous allons ouvrir deux fenêtres DOS, chacune avec les variables d'environnement habituelles :

a) la première va toujours nous permettre de lancer l'émulateur de carte C-JCRE, avec l'option -i pour lui désigner le fichier qui lui servira d'EEPROM et qui lui fournira les valeurs mémorisées et l'option -o pour mémoriser les nouvelles données dans un autre fichier (appelons-le finement eeprombis) :

```
| C:\javacard>cref -i eeprom -o eeprombis
```

b) la deuxième va nous permettre d'exécuter notre script de commandes :

```
| C:\java-netbeans-application\AppletPorteMonnaie\eCommerce\javacard >apdutool -o rest.log
| testAppletPorteMonnaie.scr
```

Ce qui donne comme résultat :

```
| Java Card 2.2.1 ApduTool (version 0.20)
| Copyright 2003 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.
| Opening connection to localhost on port 9025.
| Connected.
| Received ATR = 0x3b 0xf0 0x11 0x00 0xff 0x00
```

Nous pouvons examiner dans le fichier rest.log le résultat des diverses commandes envoyées à notre applet :

rest.log

```

CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 06, 00, 00, 00, 01, 01, Le: 00, SW1: 90, SW2: 00
CLA: b0, INS: 20, P1: 00, P2: 00, Lc: 04, 01, 02, 03, 04, Le: 00, SW1: 90, SW2: 00
CLA: b0, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 00, SW1: 90, SW2: 00
CLA: b0, INS: 40, P1: 00, P2: 00, Lc: 01, 64, Le: 00, SW1: 6a, SW2: 85
CLA: b0, INS: 30, P1: 00, P2: 00, Lc: 01, 64, Le: 00, SW1: 90, SW2: 00
CLA: b0, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 64, SW1: 90, SW2: 00
CLA: b0, INS: 40, P1: 00, P2: 00, Lc: 01, 32, Le: 00, SW1: 90, SW2: 00
CLA: b0, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 32, SW1: 90, SW2: 00
CLA: b0, INS: 30, P1: 00, P2: 00, Lc: 01, 80, Le: 00, SW1: 6a, SW2: 83
CLA: b0, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 32, SW1: 90, SW2: 00
CLA: b0, INS: 40, P1: 00, P2: 00, Lc: 01, 33, Le: 00, SW1: 6a, SW2: 85
CLA: b0, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 32, SW1: 90, SW2: 00
CLA: b0, INS: 40, P1: 00, P2: 00, Lc: 01, 80, Le: 00, SW1: 6a, SW2: 83
CLA: b0, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 32, SW1: 90, SW2: 00
CLA: b0, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 00, SW1: 67, SW2: 00
CLA: b0, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 32, SW1: 90, SW2: 00

```

Eh ben ;-) ... En fait, un tel fichier est très facile à lire ;-). On peut ainsi remarquer qu'une tentative de débit sur une carte vide donne une erreur 6a85.

On peut au sein du script des commandes

echo "<chaîne de caractères"

afin de voir dans le fichier log la progression des opérations; par exemple:

testAppletPorteMonnaie.scr (2)

```

powerup;
echo "Sélection de l'applet"
// Sélection de l'applet
0x00 0xA4 0x04 0x00 0x06 0x00 0x00 0x00 0x00 0x01 0x01 0x01 0x7F;
// 90 00 = SW_NO_ERROR

echo "Vérification du PIN"
// Vérification du PIN
0xB0 0x20 0x00 0x00 0x04 0x01 0x02 0x03 0x04 0x7F;
//90 00 = SW_NO_ERROR

// Lecture du solde
0xB0 0x50 0x00 0x00 0x00 0x02;
//0x00 0x00 0x00 0x00 0x90 0x00 = Balance = 0 and SW_ON_ERROR
...
powerdown;

```

ce qui donne :

restecho.log

```

Sélection de l'applet
CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 06, 00, 00, 00, 00, 01, 01, Le: 00, SW1: 90, SW2: 00
Vérification du PIN
CLA: b0, INS: 20, P1: 00, P2: 00, Lc: 04, 01, 02, 03, 04, Le: 00, SW1: 90, SW2: 00
...

```

15. La simplification : l'Open Card Framework

15.1 L'objectif

Nous venons de le voir, la conception et la mise en place d'une applet sur une Java card n'est pas vraiment chose facile. Et, même ainsi, l'utilisation en reste assez rébarbative puisqu'il faut créer un script APDU ad hoc, ce qui implique une connaissance convenable des classes et numéros d'instruction ainsi que des codes de retour ☺ ...

On conçoit donc sans peine que l'on ait imaginé de faciliter le travail d'un développeur d'applications utilisant une Java card. Pour cela, OCF propose à ce dernier de passer par une classe de haut niveau, appelée **CardService**, où chaque fonctionnalité de l'applet correspondra à une méthode : chaque méthode portera un nom suffisamment clair et enverra la commande, se chargeant également de transformer en texte compréhensible le code de retour éventuel. La programmation est donc bien séparée de l'implémentation et de l'envoi des commandes. De plus, cette classe CardService est capable de réaliser des conversions des types familiers aux développeurs Java comme int ou String en les seuls types connus des applets, c'est-à-dire short ou byte(s).

Précisons encore qu'une telle classe CardService sera obtenue au moyen d'une classe factory **CardServiceFactory**, pour les raisons habituelles ;-) ...

15.2 L'architecture Open Card Framework

Selon cette architecture, l'utilisation d'une carte à puces au sein d'une application comporte cinq intervenants, qui sont dans l'ordre d'une abstraction décroissante :

- 5) le développeur d'applications (Application Developer)** : il se contente de connaître l'existence de l'applet et se sert des méthodes du **CardService** associé;
- 4) le fabricant de cartes (Card Issuer)** : il peut fournir un système de gestion des applets sur la carte et/ou un **CardServiceFactory**; cette dernière, si elle permet d'utiliser des fonctions avancées de la carte, n'en est pas moins attachée au fabricant et manque donc de généralité;
- 3) le développeur de cartes (Card Developer)**: comme son nom le laisse croire, il va développer l'applet Java card et sera supposé fournir des classes **CardService** et **CardServiceFactory** associées, dérivées des classes du même nom fournies par l'OpenCard; c'est à ce niveau que se situe le premier fondement même de l'architecture, puisque *le développeur d'applications s'appuiera sur ces classes*;
- 2) le fabricant de lecteur de cartes (Reader Device Provider)** : il va être tenu de fournir les classes **CardTerminal** et **CardTerminalFactory**, dérivées des classes du même nom de l'OpenCard et implémentant ainsi des méthodes de forme standard; *le développeur de cartes s'appuiera donc sur ces méthodes*;
- 1) l'OCF lui-même**, qui fournit les classes de base : outre les classes abstraites déjà évoquées (**CardService(Factory)** et **CardTerminal(Factory)**), on trouvera encore les trois classes :

◆ public final class **SmartCard** extends java.lang.Object
Définie dans le package opencard.core.service, elle représente le point d'entrée de l'utilisation d'une applet en fournissant le CardService par

```
public CardService getCardService(java.lang.Class clazz, boolean block)
throws java.lang.ClassNotFoundException, CardServiceException
```

Son constructeur

```
public SmartCard(CardServiceScheduler scheduler, CardID cid)
```

indique bien qui sont les deux autres classes ...

- ◆ public class **CardID** extends java.lang.Object

Définie dans le package opencard.core.terminal, elle représente l'**ATR** (Answer To Reset) de la carte, c'est-à-dire un identifiant du type de la carte, ainsi que le numéro du slot dans lequel la carte est insérée.

- ◆ public final class **CardServiceScheduler** extends java.lang.Object implements CTListener

Définie dans le package opencard.core.service, elle fournit les accès physiques à la carte et gère les CardService : autrement dit, elle permet à plusieurs CardService de travailler en même temps sur la même carte. L'interface **CTListener**, défini dans le package opencard.core.event, définit le comportement de tout objet gérant une carte, avec les méthodes prévisibles :

```
public void cardInserted(CardTerminalEvent ctEvent) throws CardTerminalException  
public void cardRemoved(CardTerminalEvent ctEvent) throws CardTerminalException
```

où la classe événement **CardTerminalEvent** contient des méthodes utiles comme :

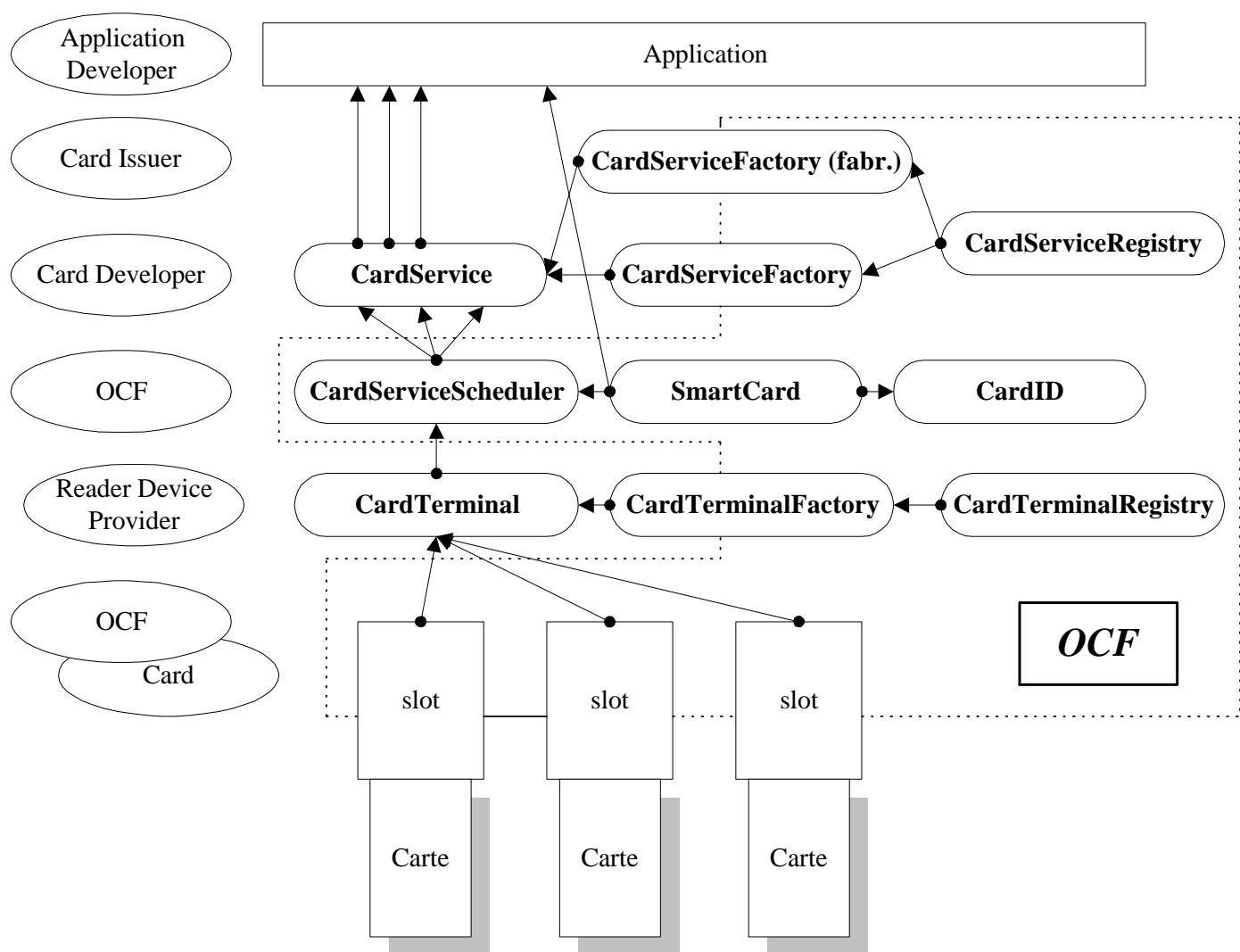
```
public CardTerminal getCardTerminal()
```

ou

```
public int getSlotID()
```

Il faut encore signaler l'existence des classes registry (CardServiceRegistry et CardTerminalRegistry) dont le rôle est évidemment de retrouver les différents éléments de l'architecture afin de pouvoir les faire communiquer.

Donc, schématiquement, l'architecture OCF se résume ainsi :



15.3 La programmation d'un CardService

Nous avons vu que la création des **CardService** et **CardServiceFactory** est du ressort du développeur de cartes. Il ne faut donc pas s'étonner de l'aspect bas niveau de la programmation exposée ici, puisque l'objectif final est précisément d'occulter ce bas niveau pour les développeurs d'applications.

Un **CardService** dérive toujours de la classe du même nom du package `opencard.core.service`. Le constructeur se contente d'appeler celui de la classe mère et instancie la variable membre représentant une commande APDU qui sera créée et envoyée par les autres méthodes; cette variable est une instance de la classe **CommandAPDU** dérivée de la classe abstraite **APDU** au sein du package `opencard.core.terminal`. Le constructeur utilisé ici est :

```
public CommandAPDU(int size)
```

où le paramètre fixe la taille du buffer qui contiendra effectivement la commande. Donc :

appletPorteMonnaieCardService.java

```
package PorteMonnaieCardService;

import opencard.core.service.*;
import opencard.core.terminal.*;
import opencard.core.util.*;

public class appletPorteMonnaieCardService extends CardService
{
    private CommandAPDU cmd;

    /* Constructeur */
    public appletPorteMonnaieCardService ()
    {
        super();
        cmd = new CommandAPDU(100);
    }
    ...
}
```

Les autres méthodes ont donc pour rôle d'implémenter les fonctionnalités de l'applet associée. Chacune d'entre elles suit **le même schéma** :

1) allocation d'un canal de communication vers la carte au moyen de la méthode de CardService :

```
protected void allocateCardChannel() throws InvalidCardChannelException
```

2) fabrication de la commande APDU en plaçant patiemment dans la commande APDU les bytes qui la constituent, ceci au moyen de la méthode héritée :

```
public void append(byte b) throws java.lang.IndexOutOfBoundsException
```

3) envoi de la commande et récupération de la réponse, en utilisant le canal de communication, alloué précédemment, au moyen de la méthode

```
public final CardChannel getCardChannel()
```

L'objet instance de **CardChannel** (défini dans l'habituel opencard.core.service) possède la méthode d'envoi :

```
public ResponseAPDU sendCommandAPDU(CommandAPDU cmdAPDU)
                                      throws InvalidCardChannelException, CardTerminalException
```

La réponse est donc une instance de la classe **ResponseAPDU**, laquelle permet de récupérer les status words par les méthodes :

```
public final byte sw1()
public final byte sw2()
```

4) analyse de la réponse et affichage sous une forme compréhensible; la classe **HexString** (du package opencard.core.util) est dédiée aux transformations byte-short/int-String et fournit notamment une méthode :

public static java.lang.String **hexify**(byte[] data)

dont on devine le rôle ...

5) libération du canal de communication au moyen de la méthode héritée :

protected void **releaseCardChannel()** throws InvalidCardChannelException

En résumé, par exemple, la méthode de sélection de notre applet s'écrit ainsi :

appletPorteMonnaieCardService.java (2)

```
package PorteMonnaieCardService;
import opencard.core.service.*;
import opencard.core.terminal.*;
import opencard.core.util.*;
import opencard.opt.service.*;
public class appletPorteMonnaieCardService extends CardService
{
    private CommandAPDU cmd;
    /* Constructeur */
    ...
    /* Sélection */
    public void Select() throws CardTerminalException,
        CardServiceUnexpectedResponseException
    {
        allocateCardChannel();
        cmd = new CommandAPDU(100);
        byte data[] = { (byte) 0xA0,(byte) 0x00,(byte) 0x00,(byte) 0x00,
            (byte) 0x62,(byte) 0x03,(byte) 0x01,(byte) 0x08,(byte) 0x01 };

        cmd.setLength(0);          // Taille de la commande APDU
        cmd.append((byte) 0x00);   // Classe d'instruction
        cmd.append((byte) 0xA4);   // Instruction
        cmd.append((byte) 0x04);   // Parametre 1
        cmd.append((byte) 0x00);   // Parametre 2
        cmd.append((byte) 0x06);   // Lc
        cmd.append((byte) 0x00);   // Données : AID de l'applet
        cmd.append((byte) 0x00);
        cmd.append((byte) 0x00);
        cmd.append((byte) 0x00);
        cmd.append((byte) 0x01);
        cmd.append((byte) 0x01);
        cmd.append((byte) 0x00);   // Longueur attendue en réponse
```

```

System.out.println("\nEnvoi de la commande SELECT au JCRE : ");
System.out.println(cmd.toString());
ResponseAPDU resp = getCardChannel().sendCommandAPDU(cmd);

String reponse = new
    String(HexString.hexify(resp.sw1())+HexString.hexify(resp.sw2()));
System.out.println("Response to 'SELECT' command: "+reponse);

if (reponse.compareTo("9000")!=0)
    throw new CardServiceUnexpectedResponseException(reponse);
System.out.println("-----");
releaseCardChannel();
}
}

```

On remarquera l'usage de l'exception **CardServiceUnexpectedResponseException** du package `opencard.opt.service` qui est l'héritière d'une longue lignée :

```

+--opencard.core.OpenCardException
|
+--opencard.core.service.CardServiceException
|
+--opencard.core.service.CardServiceImplementationException
|
+--opencard.opt.service.CardServiceUnexpectedResponseException

```

dont la classe fondatrice `OpenCardException` est dérivée de `java.io.IOException`.

Les autres méthodes s'écrivent manière similaire (l'applet travaille ici sur 3 bytes au lieu de 2, ce qui était le cas de l'exemple précédent) :

appletPorteMonnaieCardService.java (2)

```

package PorteMonnaieCardService;

import opencard.core.service.*;
import opencard.core.terminal.*;
import opencard.core.util.*;
import opencard.opt.service.*;

public class appletPorteMonnaieCardService extends CardService
{
    private CommandAPDU cmd;

    // Pour les conversions décimal-hexadécimal
    byte ref[] = { 0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,
                   0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F };

    /* Constructeur */
    ...
    /* Sélection */

```

```

...
public void VerifyUser(String PIN)
    throws CardTerminalException, CardServiceUnexpectedResponseException
{
    allocateCardChannel();
    cmd = new CommandAPDU(100);
    if (PIN.length()!=4)
        throw new CardServiceUnexpectedResponseException
            ("La longueur du pin doit être de 4");

    cmd.setLength(0);
    cmd.append((byte) 0xB0);
    cmd.append((byte) 0x20);
    cmd.append((byte) 0x00);
    cmd.append((byte) 0x00);
    cmd.append((byte) 0x04);
    cmd.append(charToHexaByte(PIN.charAt(0)));
    cmd.append(charToHexaByte(PIN.charAt(1)));
    cmd.append(charToHexaByte(PIN.charAt(2)));
    cmd.append(charToHexaByte(PIN.charAt(3)));
    cmd.append((byte) 0x00);

    System.out.println("\nEnvoi de la commande VERIFY_USER au JCRE : ");
    System.out.println(cmd.toString());
    ResponseAPDU resp = getCardChannel().sendCommandAPDU(cmd);

    String reponse = new
        String(HexString.hexify(resp.sw1())+HexString.hexify(resp.sw2()));
    System.out.println("Response to 'VERIFY_USER' command: "+reponse);

    if (reponse.compareTo("9000")!=0)
        throw new CardServiceUnexpectedResponseException(reponse);
    System.out.println("-----");
    releaseCardChannel();
}

public double GetBalance()
    throws CardTerminalException, CardServiceUnexpectedResponseException
{
    allocateCardChannel();
    cmd = new CommandAPDU(100);
    cmd.setLength(0);
    cmd.append((byte)0xB0);
    cmd.append((byte)0x50);
    cmd.append((byte)0x00);
    cmd.append((byte)0x00);
    cmd.append((byte)0x00);
    cmd.append((byte)0x03);

    System.out.println("\nEnvoi de la commande GET BALANCE au JCRE : ");
}

```

```
System.out.println(cmd.toString());
ResponseAPDU resp = getCardChannel().sendCommandAPDU(cmd);

String reponse = new
    String(HexString.hexify(resp.sw1())+HexString.hexify(resp.sw2()));
System.out.println("Response to 'GET BALANCE' command: "+reponse);

if (reponse.compareTo("9000")!=0)
    throw new CardServiceUnexpectedResponseException(reponse);

byte[] buffer = resp.data();      // méthode de ResponseAPDU
double montant = buffer[0]*100;
montant += buffer[1];
montant += buffer[2]*0.01;

System.out.println("Montant : "+montant);
System.out.println("Res : "+HexString.hexify(buffer));

System.out.println("-----");
releaseCardChannel();

return montant;
}

public void Credit(double montant)
    throws CardTerminalException, CardServiceUnexpectedResponseException
{
    double solde = GetBalance();
    cmd = new CommandAPDU(100);
    allocateCardChannel();

    if (montant<0)
        throw new CardServiceUnexpectedResponseException
            ("Le montant transféré ne peut être négatif");

    if (montant+solde> 150)
        throw new CardServiceUnexpectedResponseException
            ("La carte peut contenir au maximum 150 Euro");
    System.out.println("Montant a traiter : "+montant);

    // On crée trois short contenant la partie centaines, entiere et decimale du
    //montant
    montant *= 100;

    cmd.setLength(0);
    cmd.append((byte)0xB0);
    cmd.append((byte)0x30);
    cmd.append((byte)0x00);
    cmd.append((byte)0x00);
    cmd.append((byte)0x03);
```

```

cmd.append(shortToHexaByte((short)(montant/10000)));
cmd.append(shortToHexaByte((short)((montant%10000)/100)));
cmd.append(shortToHexaByte((short)(montant%100)));
cmd.append((byte)0x00);

System.out.println("\nEnvoi de la commande CREDIT au JCRE : ");
System.out.println(cmd.toString());
ResponseAPDU resp = getCardChannel().sendCommandAPDU(cmd);

String reponse = new
    String(HexString.hexify(resp.sw1())+HexString.hexify(resp.sw2()));
System.out.println("Response to 'CREDIT' command: "+reponse);

if (reponse.compareTo("9000")!=0)
    throw new CardServiceUnexpectedResponseException(reponse);
System.out.println("-----");
releaseCardChannel();
}

public void Debit(double montant)
    throws CardTerminalException, CardServiceUnexpectedResponseException
{
    allocateCardChannel();
    System.out.println("Montant à traiter : "+montant);
    montant *= 100;

    cmd.setLength(0);
    cmd.append((byte)0xB0);
    cmd.append((byte)0x40);
    cmd.append((byte)0x00);
    cmd.append((byte)0x00);
    cmd.append((byte)0x03);
    cmd.append(shortToHexaByte((short)(montant/10000)));
    cmd.append(shortToHexaByte((short)((montant%10000)/100)));
    cmd.append(shortToHexaByte((short)(montant%100)));
    cmd.append((byte)0x00);

    System.out.println("\nEnvoi de la commande DEDIT au JCRE : ");
    System.out.println(cmd.toString());
    ResponseAPDU resp = getCardChannel().sendCommandAPDU(cmd);

    String reponse = new
    String(HexString.hexify(resp.sw1())+HexString.hexify(resp.sw2()));
    System.out.println("Response to 'DEDIT' command: "+reponse);

    if (reponse.compareTo("9000")!=0)
        throw new CardServiceUnexpectedResponseException(reponse);
    System.out.println("-----");
    releaseCardChannel();
}

```

```

public void Deconnect()
    throws CardTerminalException, CardServiceUnexpectedResponseException
{
    allocateCardChannel();
    cmd = new CommandAPDU(100);
    cmd.setLength(0);
    cmd.append((byte)0xC0);
    cmd.append((byte)0xDB);
    cmd.append((byte)0x00);
    cmd.append((byte)0x00);
    cmd.append((byte)0x00);
    cmd.append((byte)0x00);

    System.out.println("\nEnvoi de la commande DECONNECT au JCREE : ");
    System.out.println(cmd.toString());
    ResponseAPDU resp = getCardChannel().sendCommandAPDU(cmd);

    String reponse = new
        String(HexString.hexify(resp.sw1())+HexString.hexify(resp.sw2()));
    System.out.println("Response to 'DECONNECT' command: "+reponse);

    if (reponse.compareTo("9000")!=0)
        throw new CardServiceUnexpectedResponseException(reponse);
    System.out.println("-----");
    releaseCardChannel();
}

/* Méthodes utilitaires */
protected byte charToHexaByte(char c)
{
    byte b = (byte) c;
    b= (byte) (b-0x30);
    return b;
}

protected byte shortToHexaByte(short s)
{
    byte b;

    b= ref[(short)(s/16)];
    b*=16;
    b+=ref[(short)(s%16)];

    return b;
}
}

```

15.4 La programmation d'une CardServiceFactory

Un objet de type "factory pour card service" a pour rôle d'**instancier un CardService correspondant à un type particulier de cartes à puces**. Typiquement, cette classe doit dériver de la classe abstraite **CardServiceFactory**, classe appartenant au package opencard.core.service. Outre un constructeur par défaut, elle se doit d'implémenter les deux méthodes :

1) protected abstract CardType getCardType(CardID cid, CardServiceScheduler scheduler)
throws CardTerminalException

Elle permet de savoir si la factory est capable de fournir un CardService compatible avec la carte insérée dans le CAD (ou plutôt avec son OS). Dans l'affirmative, on obtient comme valeur de retour un objet **CardType** (du même package) qui contient assez d'informations pour permettre l'utilisation de la méthode

protected java.lang.Class getClassFor(java.lang.Class clazz, CardType type)

dont on devine qu'elle permet de localiser la définition de la classe de type CardService passée en paramètre. Dans la négative, on obtient le CardType signifiant qu'il n'est pas possible de fournir un CardService, soit la variable de classe de la classe CardType :

public static CardType UNSUPPORTED

2) protected abstract java.util.Enumeration getClasses(CardType type)

Elle permet évidemment d'obtenir les CardServices compatibles avec la carte éventuellement promis par la méthode précédente ...

Concrètement, notre factory sera écrite avec

- ◆ une variable de classe nommée listeServices, de type Vector, chargée bien sûr de contenir les CardService retenus – ici, ce sera le nôtre;
- ◆ un constructeur par défaut qui place notre CardService dans le vecteur;
- ◆ la surcharge de la méthode getCardType(): en fait, nous allons essayer dans cette méthode de réaliser une commande APDU SELECT; le succès ou l'échec de cette opération nous renseignera sur la compatibilité (en cas de succès, nous décrèterons la carte comme étant de type 1, nombre que nous passerons au constructeur); on utilisera pour cette commande la classe ISOCommandAPDU du package opencard.opt.terminal, avec le constructeur

public **ISOCommandAPDU(int size, byte classByte, byte instruction, byte p1, byte p2, byte[] data, int le)**

- ◆ la surcharge de la méthode getClasses(), qui se contente de parcourir listeServices.
Donc :

appletPorteMonnaieCardServiceFactory.java

package PorteMonnaieCardService;

```
/*
 * appletPorteMonnaieCardServiceFactory.java
 *
 * Created on 12 septembre 2003, 9:14
 */
```

```

import java.util.*;
import opencard.core.service.*;
import opencard.core.terminal.*;
import opencard.core.util.*;
import opencard.opt.terminal.*;

public class appletPorteMonnaieCardServiceFactory extends CardServiceFactory
{
    private static Vector listeServices = new Vector();

    public appletPorteMonnaieCardServiceFactory ()
    {
        super();
        listeServices.addElement(appletPorteMonnaieCardService.class);
    }

    protected CardType getCardType(CardID cid, CardServiceScheduler sched)
        throws CardTerminalException
    {
        byte data[] = { (byte) 0x00,(byte) 0x00,(byte) 0x00,(byte) 0x00,
                       (byte) 0x01,(byte) 0x01 };

        ISOCommandAPDU cmd = new ISOCommandAPDU
        (12,                                // Taille de la commande APDU
         (byte) 0x00,  // Classe d'instruction
         (byte) 0xA4,  // Instruction
         (byte) 0x04,  // P1
         (byte) 0x00,  // P2
         data,                      // Données
         (byte) 0x00); // Longueur attendue en réponse

        System.out.println("Envoi de la commande SELECT au JCRE : ");
        System.out.println(cmd.toString()+"\n");
        ResponseAPDU resp = sched.getSlotChannel().sendAPDU(cmd);

        String reponse =
            new String(HexString.hexify(resp.sw1())+HexString.hexify(resp.sw2())));
        System.out.println("Response to 'SELECT' command: "+reponse);

        if (reponse.compareTo("9000")==0) return new CardType(1);
        else return CardType.UNSUPPORTED;
    }

    protected Enumeration getClasses (CardType type)
    {
        return listeServices.elements();
    }
}

```

15.5 L'utilisation d'une CardServiceFactory et d'un CardService

Pour obtenir notre CardService, il nous faut d'abord notre factory. Selon une méthode maintenant bien connue, un fichier properties va être le dépositaire des informations nécessaires pour trouver cette factory. Ce fichier se nomme **opencard.properties** et sera recherché, dans l'ordre, dans les répertoires suivants :

- ◆ le répertoire lib du JRE;
- ◆ le répertoire au nom de l'utilisateur dans Documents and Settings (donc le user.home), mais sous le nom de .opencard.properties;
- ◆ sur le bureau (cas habituel);
- ◆ sur le bureau mais sous le nom de .opencard.properties;
- ◆ dans le user.dir de l'utilisateur (ici : c:\java-sun-application\AppletPorteMonnaieCardService).

Ce fichier comporte en fait au moins les deux entrées du type suivant¹ :

opencard.properties

```
OpenCard.services = PorteMonnaieCardService.applePorteMonnaieCardServiceFactory
OpenCard.terminals = com.gemplus.opencard.terminal.apduio.ApduIOCardTerminalFactory
|mySim|Socket|localhost:9025
```

La première ligne permet évidemment de localiser la factory tandis que la deuxième spécifie la classe utilisée pour obtenir le terminal, ici celle de l'émulateur C-JCRE.

Nous voilà à présent prêts à utiliser notre CardService. Le point d'entrée dans ce mécanisme est matérialisé par la classe **SmartCard** (de l'éternel package opencard.core.service), qui va nous permettre d'utiliser les fonctionnalités développées dans notre CardService; en fait, l'objet SmartCard est étroitement lié à un objet CardServiceScheduler qui, pour rappel, gère les canaux d'accès à la carte et les différents CardServices. La première méthode à en utiliser est

```
public static void start()
    throws OpenCardPropertyLoadingException, java.lang.ClassNotFoundException,
           CardServiceException, CardTerminalException
```

Cette méthode, en quelque sorte, met en place le décor, c'est-à-dire que l'OpenCard Framework est initialisé et que le fichier opencard.properties est recherché. Si celui-ci est trouvé, sa lecture permettra de remplir les objets **CardTerminalRegistry** et **CardServiceRegistry**: le nom de ces deux classes est assez explicite pour comprendre ce qu'elles sont sensées enregistrer (encore que la deuxième sert à enregistrer les factories de CardService plutôt que les CardService eux-mêmes) – mais, à l'évidence, c'est un point de passage obligé.

La deuxième méthode à invoquer est celle qui permet d'obtenir un objet SmartCard :

```
public static SmartCard waitForCard (CardRequest req) throws CardTerminalException
```

Comme son nom l'indique, elle attend qu'une carte soit insérée dans le terminal ou encore que l'émulateur soit lancé. Elle utilise un objet **CardRequest** qui permet de spécifier quel type de carte l'application attend. On construit un tel objet en utilisant le constructeur

¹ il n'y a bien que deux lignes – la seconde est ici coupée pour des raisons typographiques

```
public CardRequest(int waitBehavior, CardTerminal terminal, Class cardServiceClass)
```

Seul le premier paramètre est obligatoire : il permet de spécifier si la carte éventuellement présente peut être prise en compte ou si au contraire il faut en attendre une nouvelle, ceci au moyen des constantes de classe :

```
public static final int ANYCARD  
public static final int NEWCARD
```

Le deuxième paramètre permet de spécifier le terminal sur lequel la carte doit être attendue (null signifie "sur tous les terminaux présents") tandis que le troisième précise quel type de CardService devrait être disponible pour la carte (null signifie que tout convient à priori).

Nous pouvons alors enregistrer notre factory au moyen de la méthode de CardServiceRegistry :

```
public void add (CardServiceFactory factory)
```

l'objet registry du framework en lui-même nous étant fourni au moyen de la méthode de classe de cette même classe :

```
public static CardServiceRegistry getRegistry()
```

La factory étant à présent enregistrée, nous pouvons enfin obtenir notre CardService au moyen de l'objet SmartCard qui appelle sa méthode :

```
public CardService getCardService(java.lang.Class clazz, boolean block)  
throws java.lang.ClassNotFoundException, CardServiceException
```

A partir de ce moment, si la carte est valide (donc, si la référence de l'objet SmartCard n'est pas nulle), nous pouvons utiliser toutes les méthodes du CardService. Nous terminerons l'application en appelant la méthode de SmartCard :

```
public static void shutdown() throws CardTerminalException
```

qui, bien sûr, nettoie tout l'environnement de travail.

En résumé :

applicPorteMonnaie.java

```
/*  
 * applicPorteMonnaie.java  
 * Created on 12 septembre 2003, 15:30  
 */  
  
import opencard.core.terminal.*;  
import opencard.core.util.*;  
import opencard.core.service.*;  
import opencard.opt.util.*;  
import opencard.opt.service.*;  
import PorteMonnaieCardService.*;
```

```

public class applicPorteMonnaie
{
    public static void main(String[] args)
    {
        SmartCard smart_card;

        try
        {
            SmartCard.start();

            CardRequest cr = new CardRequest (CardRequest.ANYCARD,null,null);
            smart_card = SmartCard.waitForCard (cr);

            CardServiceRegistry.getRegistry().add(
                new appletPorteMonnaieCardServiceFactory() );

            appletPorteMonnaieCardService cs =
                (appletPorteMonnaieCardService)smart_card.
                getCardService(appletPorteMonnaieCardService.class,true);

            if (smart_card != null)
            {
                CardID cardID = smart_card.getCardID ();
                System.out.println("ATR : "+cardID.toString());
            }
            else
            {
                System.out.println ("La carte n'est pas valide");
            }

            cs.Select();
            cs.VerifyUser("4321");
            cs.GetBalance();
            double nb = 101.3;
            cs.Credit(nb);
            cs.GetBalance();
            double nb1 = 60.1;
            cs.Debit(nb1);
            cs.GetBalance();
            cs.Credit(130);
        }
        catch(OpenCardPropertyLoadingException e)
        { System.out.println("OpenCardPropertyLoadingException: "+e.getMessage()); }
        catch(CardServiceImplementationException e)
        { System.out.println("CardServiceImplementationException: "+e.getMessage()); }
        catch(CardServiceInsufficientMemoryException e)
        { System.out.println("CardServiceInsufficientMemoryException:
            "+e.getMessage()); }
        catch(CardServiceObjectNotAvailableException e)
        { System.out.println("CardServiceObjectNotAvailableException:
            "+e.getMessage()); }
    }
}

```

```
        "+e.getMessage()); }  
    catch(CardServiceOperationFailedException e)  
    { System.out.println("CardServiceOperationFailedException: "+e.getMessage()); }  
    catch(CardServiceResourceNotFoundException e)  
    { System.out.println("CardServiceResourceNotFoundException:  
        "+e.getMessage()); }  
    catch (ClassNotFoundException e)  
    { System.out.println("ClassNotFoundException: "+e.getMessage()); }  
    catch(CardServiceException e)  
    { System.out.println("CardServiceException: "+e.getMessage()); }  
    catch(CardTerminalException e)  
    { System.out.println("CardTerminalException: "+e.getMessage()); }  
    finally  
    {  
        try  
        {  
            SmartCard.shutdown();  
        }  
        catch (Exception e)  
        {  
            System.out.println("Exception : "+e.getMessage());  
        }  
    }  
}  
}
```

16. Remplacer l'émulateur par un lecteur série

Lorsque l'on travaille sur un véritable terminal, soit un lecteur de cartes à puce du type Gemplus, connecté non pas sur un port USB mais sur un port série, le fichier opencard.properties prend l'aspect suivant :

opencard.properties (lecteur Gemplus – Windows)

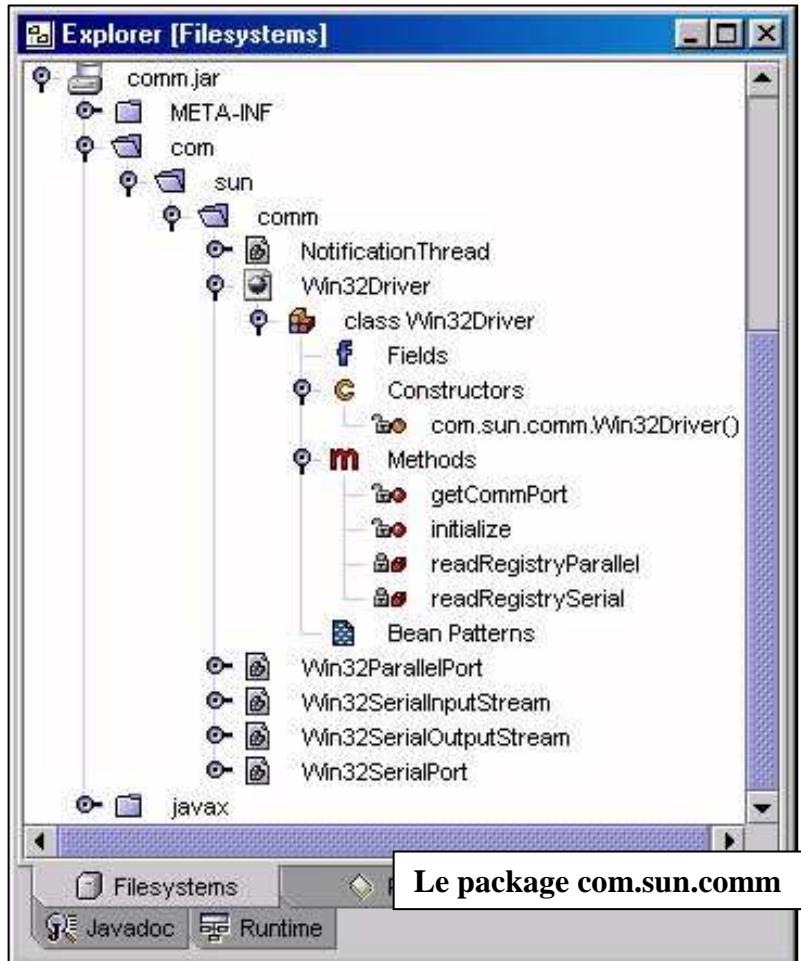
```
OpenCard.services = PorteMonnaieCardService.applePorteMonnaieCardServiceFactory  
OpenCard.terminals =  
com.gemplus.opencard.terminal.GemplusCardTerminalFactory|mygcr|GCR410COM1
```

ou

opencard.properties (lecteur Gemplus – Unix)

```
OpenCard.services = PorteMonnaieCardService.applePorteMonnaieCardServiceFactory  
OpenCard.terminals =  
com.gemplus.opencard.terminal.GemplusCardTerminalFactory|mygcr|GCR410|/dev/ttyS1
```

Mais, surtout, il s'agira d'utiliser la ligne série au moyen des packages ad hoc. Pour cela, nous utiliserons les classes de comm.jar, qui se trouve dans le répertoire commapi évoqué plus haut (paragraphe 12 c) :



Avant d'acquérir le CardService comme indiqué pour l'émulateur de carte, il nous faudra tout d'abord initialiser le driver. Celui-ci est une implémentation de l'interface CommDriver, du package javax.comm, implémentation variable selon les machines et les systèmes d'exploitation. Pour notre part, c'est la classe Win32Driver, du package com.sun.comm, qui fera l'affaire. Une fois une instance obtenue (par newInstance()), il restera à l'initialiser en appelant la méthode

```
public abstract void initialize()
```

qui charge les librairies natives éventuelles et enregistre les noms des ports série présents.

Il nous faudra donc procéder de la manière suivante :

applicPorteMonnaie.java (version pour lecteur Gemplus sur port série)

```
import opencard.core.service.*;
...
import javax.comm.*;
import PorteMonnaieCardService.*;

public class applicPorteMonnaie
{
    public static void main(String[] args)
    {
        String nomDuDriverSerie = "com.sun.comm.Win32Driver";
        try
        {
            Class laClasseDuDriverSerie = Class.forName(nomDuDriverSerie);
            CommDriver driverSerie =
                (CommDriver) laClasseDuDriverSerie.newInstance();
            driverSerie.initialize();
        }
        catch (Throwable exc)
        { System.out.println(exc.getMessage() + " -- mais on s'en f..."); }

        SmartCard smart_card;
        try
        {
            SmartCard.start();
            CardRequest cr = new CardRequest(CardRequest.ANYCARD,null,null);
            smart_card = SmartCard.waitForCard (cr);
            ... // comme avant
        }
    }
}
```

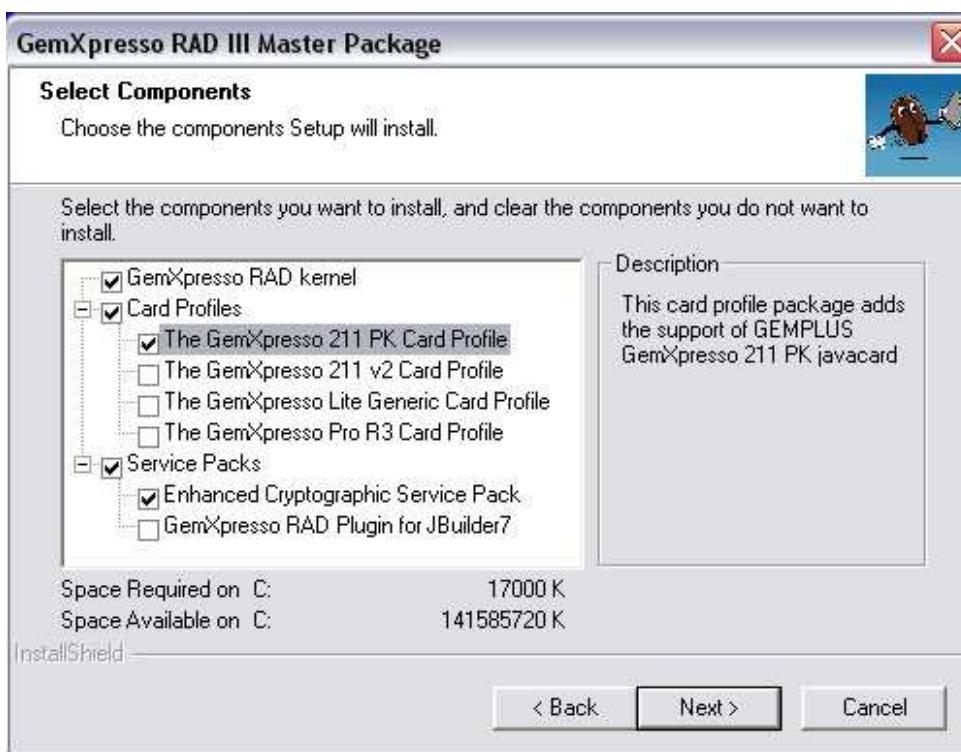
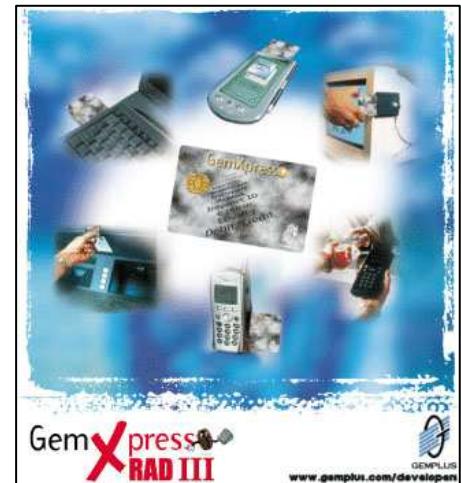
L'initialiation du driver donne une exception sous Windows (pas sous Linux ;-)) – mais ce n'est qu'un bug que l'on peut ignorer ...

17. Le développement avec le simulateur Gemplus

17.1 Les composants logiciels de base

Le développement d'applications pour cartes à puces est donc, on vient de le constater, un travail assez délicat. Si le monde des smartcards ne peut être que fruste de par sa nature même, on peut cependant trouver des logiciels facilitant certaines phases de développement. Tel est le cas du Kit GemXpresso RAD III, produit par la société Gemplus : il apporte une aide appréciable au développement d'applets pour cartes à puce et d'applications associées.

L'installation du kit ne pose guère de difficultés. On peut juste signaler que le profil de carte qui nous intéressera ici est card profile PK et aussi que l'on peut installer le service de cryptographie : même si les cartes que nous utilisons ici ne la supportent pas, l'émulateur peut par contre en faire usage.



On a ainsi mis en place divers softwares, dont essentiellement pour nous :

- ♦ un simulateur de cartes à puces;
- ♦ l'interface JCardManager qui permet de charger des applets et de les tester en leur envoyant des commandes APDU.

Petit détail complémentaire : ***il faut installer le JDK 1.2*** ! Il est en effet le JDK utilisé pour développer les Java Communication API (Comm.jar). Cela peut sembler totalement obsolète, mais cela s'explique par le fait que l'espace restreint des cartes à puce n'autorise qu'un bytecode très basique.

17.2 Les variables d'environnement

Il conviendra de définir les diverses variables d'environnement suivantes :

jc_env_gemplus.bat

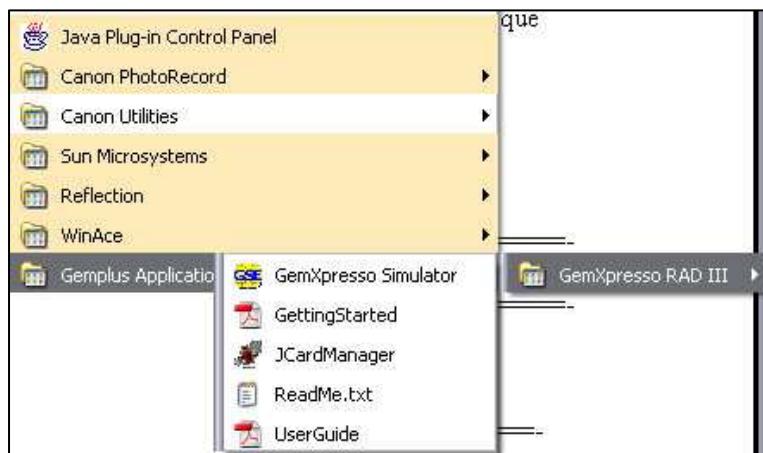
```
set JC_HOME=C:\javacard\java_card_kit-2_2_1
set JAVA_HOME=C:\jdk1.2.2
set PATH=%JC_HOME%\bin;%JAVA_HOME%\bin;%PATH%
set CLASSPATH=PorteMonnaieCardService;C:\javacard\java_card_kit-
2_2_1\lib\javacardframework.jar;C:\jdk1.2.2\jre\lib\*.jar
```

Nous pouvons compiler l'applet porte-monnaie développée précédemment :

```
C:\java-sun-application\AppletPorteMonnaie\eCommerce>javac appletPorteMonnaie.java
C:\java-sun-application\AppletPorteMonnaie\eCommerce>
```

17.3 Le développement avec simulateur : GemXpresso Simulator

Nous allons ensuite lancer le simulateur de cartes à puce depuis le menu Démarrer :



ce qui lance un service :

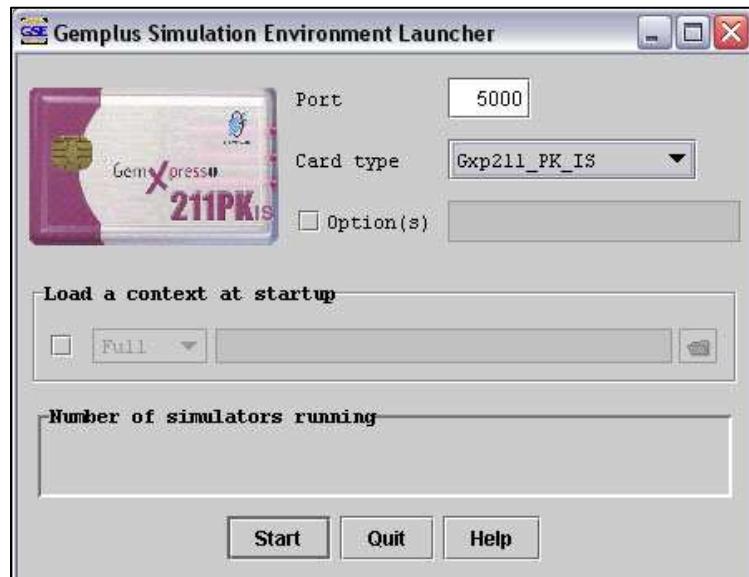
Gemplus GemXpresso RAD III Initialization batch file

- set the GSE GUI classpath
- set the Cryptographic component classpath
- set the Jhall library classpath

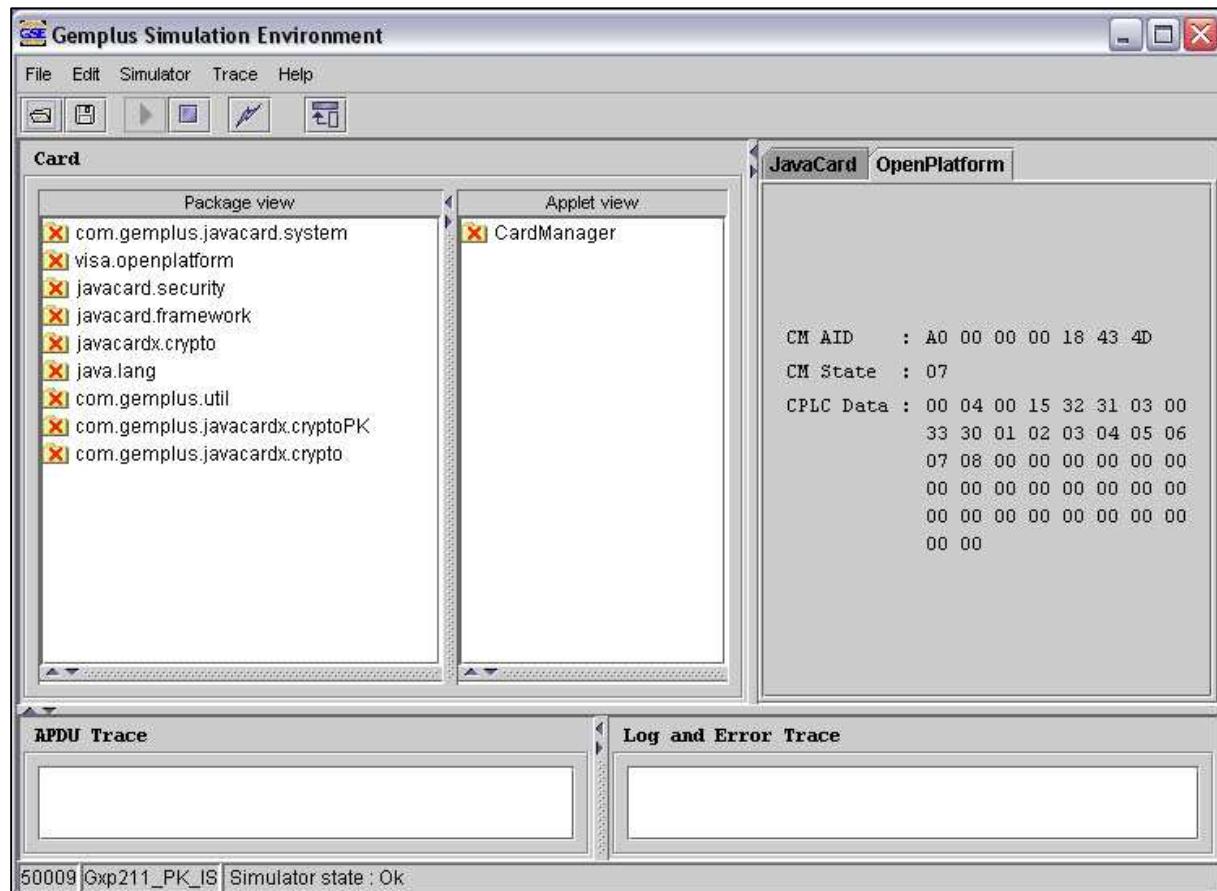
GUI of the GSE: GemXpresso Simulation Environment Interface

```
C:\Gemplus\GemXpresso.rad3>"C:\jdk1.2.2\bin\java" com.gemplus.javacard.gse.gui.G
SEGuiLauncher -noposition -conf "C:\Gemplus\GemXpresso.rad3"\conf\gse-gui.proper
ties
```

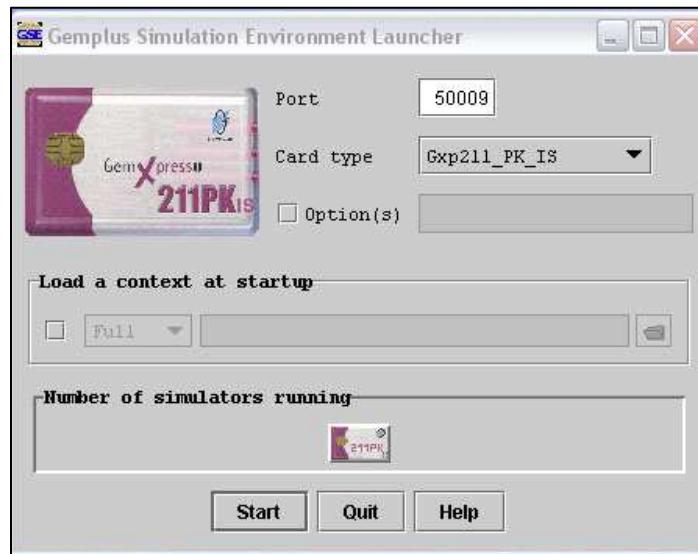
On obtient :



Dans cet interface, il faut choisir le port d'écoute du simulateur (disons 50009) et le type de carte utilisé (pour nous, GXP_211_PK_IS). La lancement proprement dit se fait par appui sur Start. On obtient alors Gemplus Simulation Environment :



où l'on voit les packages se trouvant sur la carte simulée : java.lang, javacard.security, etc avec leur "Security Domain"; une applet non modifiable (CardManager) est également déjà présente. La fenêtre du launcher subsiste avec le compteur figuré à 1



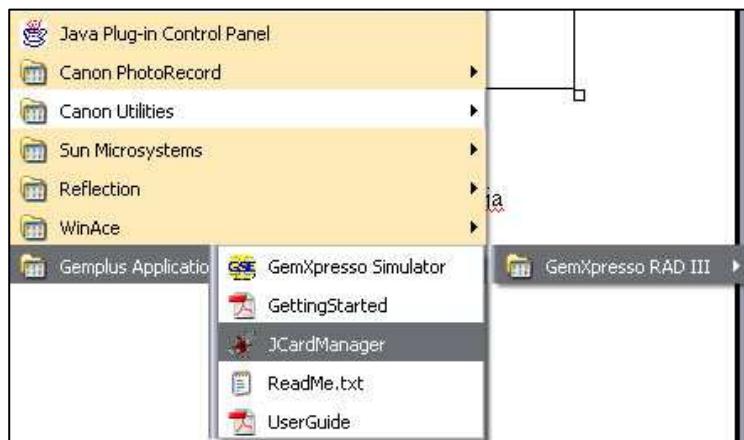
17.4 Le JCardManager

Avant d'utiliser nos outils, qui sont basés sur l'OpenCard, il ne faut pas oublier de préciser le port utilisé dans le fichier opencard.properties qui se trouve dans le sous-répertoire ad hoc du répertoire où GemPlus a été installé, soit C:\Gemplus\GemXpresso.rad3\conf :

opencard.properties

```
OpenCard.terminals = \
com.gemplus.opencard.terminal.GemplusRadCardTerminalFactory|Simulator| \
SOCKETJC21SIMULATOR|127.0.0.1:50009 | \
com.ibm.opencard.terminal.pcsc10.Pcsc10CardTerminalFactory
```

Nous pouvons à présent lancer le JCardManager, et lui faire prendre en compte cette configuration de terminal, depuis le menu Démarrer :



ce qui lance en fait un service :

```
=====
Gemplus GemXpresso RAD III Initialization batch file
=====
- set the Jhall library classpath
- set the OCF Core classpath
- set the Gemplus GSE Card Service classpath
- set the OCF default Card Terminals and Gemplus Card Terminals classpath
```

- set the GemXpresso library classpath
- set the SAP Converter tool classpath

...

- set the GSE GUI classpath

...

GemXpresso JCardManager

```
C:\Gemplus\GemXpresso.rad3>"C:\jdk1.2.2\bin\java.exe" -Dgemplus.gemxpresso.rad.home="C:\Gemplus\GemXpresso.rad3" com.gemplus.tools.gemxpresso.pilot.JCardManager JFrame
```

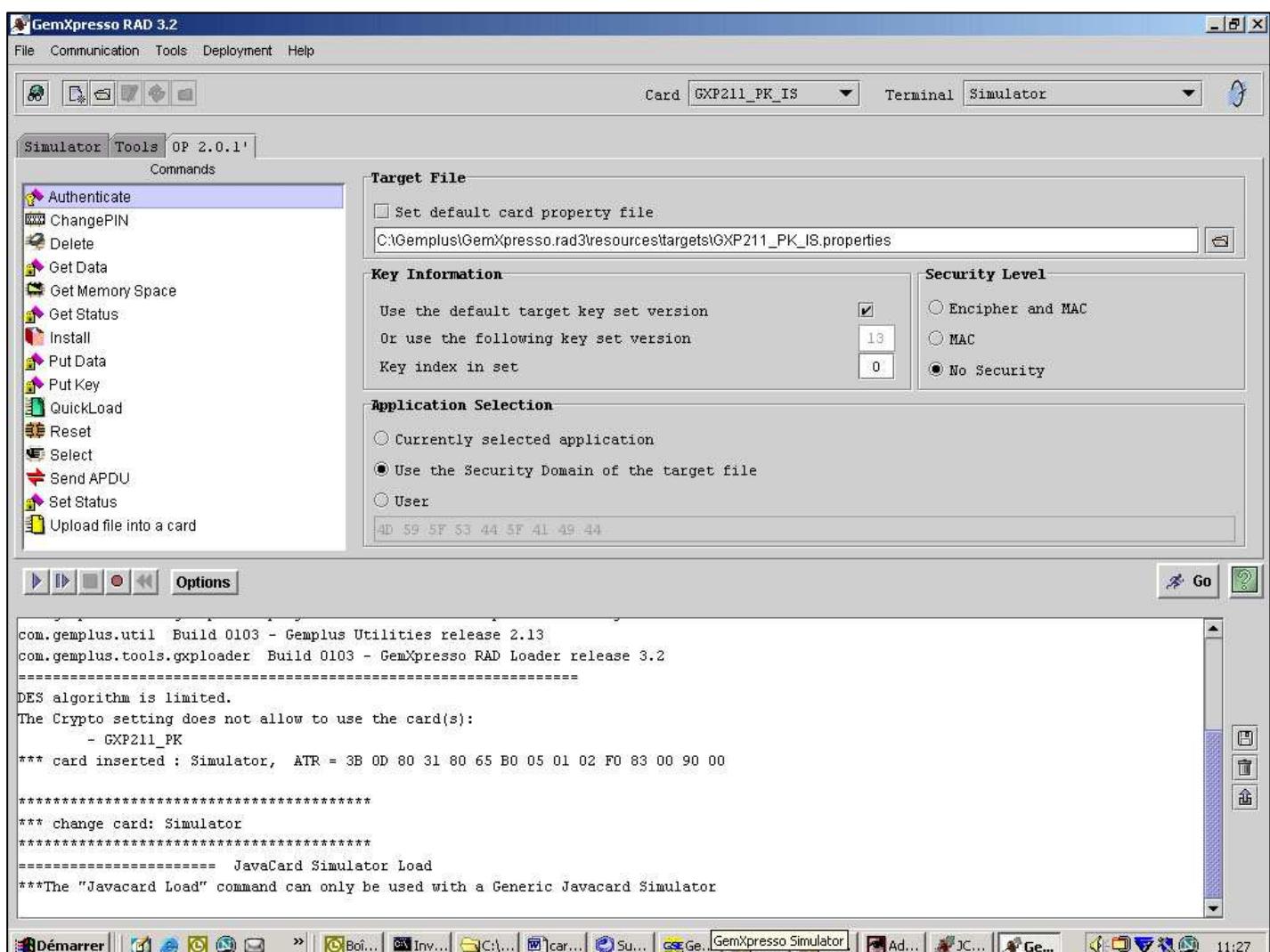
GemXpresso RAD 3.2 Build 0103 - JCardManager release 3.5

Trace file is C:\Gemplus\GemXpresso.rad3\traces.htm

TPDU Not Used

TPDU Not Used

On se retrouve devant l'interface graphique de GemXpresso RAD 3.2 dans lequel on commence par sélectionner l'onglet OP 2.0.1' (l'Open Platform Card Service qui se trouve dans l'implémentation Gemplus de l'OCF) :

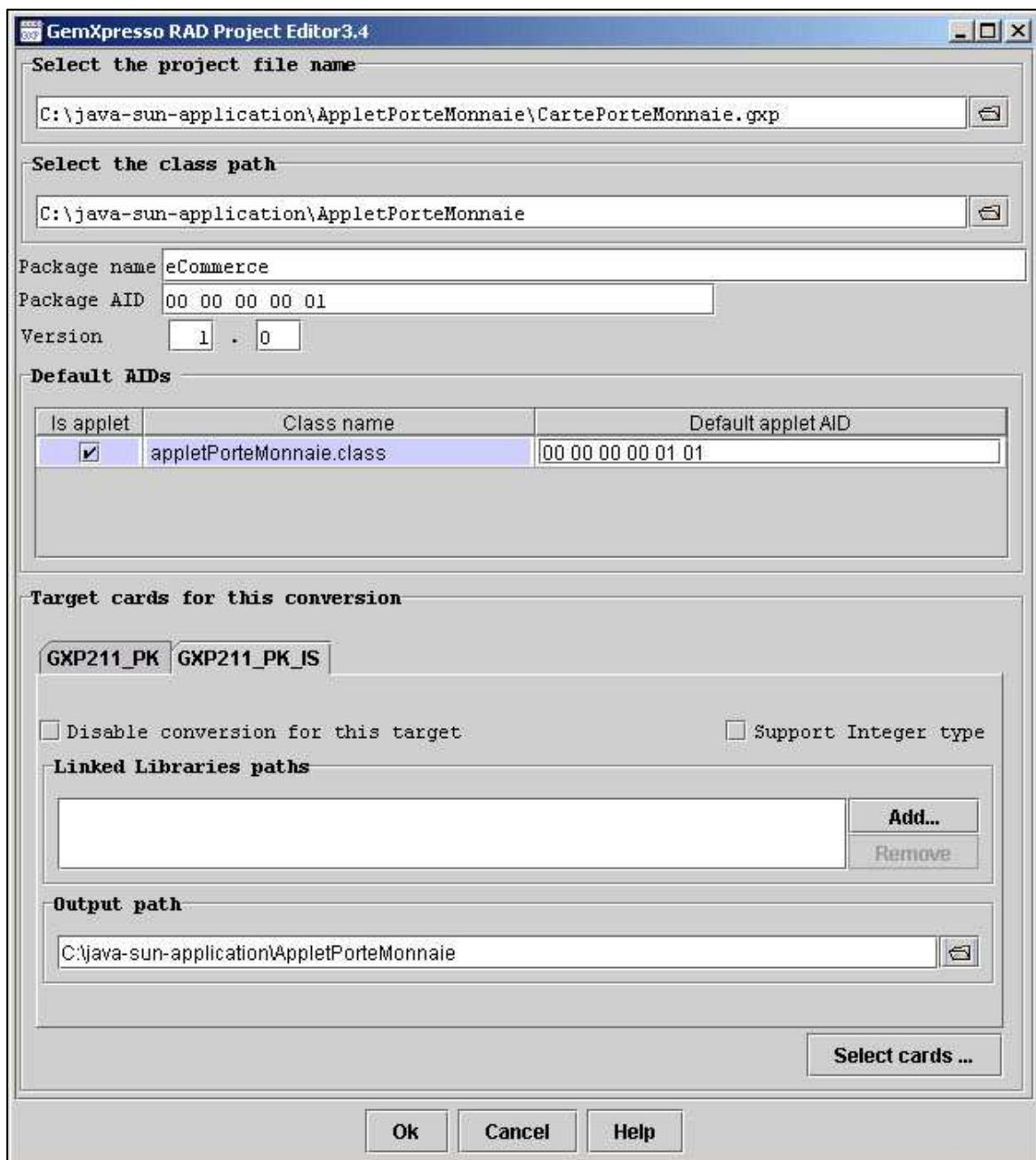


17.5 Le déploiement d'une applet de base

Supposons à présent vouloir déployer notre applet appletPorteMonnaie. Nous allons pour cela utiliser le RAD de Gemplus. Pour cela, nous sélectionnons dans le menu du JCardManager :

File → New project

Dans l'interface obtenu, nous fournissons toutes les informations concernant l'applet :



et un appui sur Ok conduit à la création d'un fichier CartePorteMonnaie.gxp. Ce fichier gxp (GemXpresso Project) contient les définitions d'un projet avec une applet, un choix de carte, etc - tout se passe comme si, dans un contexte de développement classique, on avait défini un

projet qu'il faudrait à présent compiler. Nous passons donc ici à la phase équivalente, soit la génération des fichiers CAP, EXP et JCA, ceci au moyen de

File → Convert

dans le GemXpresso RAD – ce qui donne :

Converter JC2.1 SUN Converter (version 1.0) Result on card GXP211_PK :

Java Card 2.1 Class File Converter (version 1.0)

Copyright (c) 1999 Sun Microsystems, Inc. All rights reserved.

parsing C:\java-sun-application\AppletPorteMonnaie\eCommerce\appletPorteMonnaie.class
converting eCommerce.appletPorteMonnaie

parsing C:\Gemplus\GemXpresso.rad3\resources\exportfiles\Gxp211_PK\javacard\
framework\javacard\framework.exp

writing C:\java-sun-application\AppletPorteMonnaie\oncard\GXP211_PK\eCommerce\
javacard\commerce.exp

writing C:\java-sun-application\AppletPorteMonnaie\oncard\GXP211_PK\eCommerce\
javacard\commerce.jca

conversion completed with 0 errors and 0 warnings.

JC2.1 SUN Converter (version 1.0) process is completed

JCASM program result on card GXP211_PK :

Java Card 2.1 CAP File Builder (version 1.0)

Copyright (c) 1999 Sun Microsystems, Inc. All rights reserved.

JCAsm process is completed

Converter SAP Converter (version 1.0) Result on card GXP211_PK :

SAP file : C:\java-sun-application\AppletPorteMonnaie\oncard\GXP211_PK\eCommerce\
javacard\commerce.sap done

SAP Converter (version 1.0) process is completed

Le résultat de tout ceci se traduit par 4 fichiers :

 eCommerce.exp	1 Ko	Fichier EXP
 eCommerce.jar	3 Ko	JavaSoft archive
 eCommerce.jca	12 Ko	Fichier JCA
 eCommerce.sap	2 Ko	Fichier SAP

Outre les fichiers connus exp et jca, on a donc un fichier **sap**, qui est un jar-cap propriétaire spécialement dédié au simulateur, et un jar qui contient une série de fichiers cap.

17.6 Le déploiement d'une applet

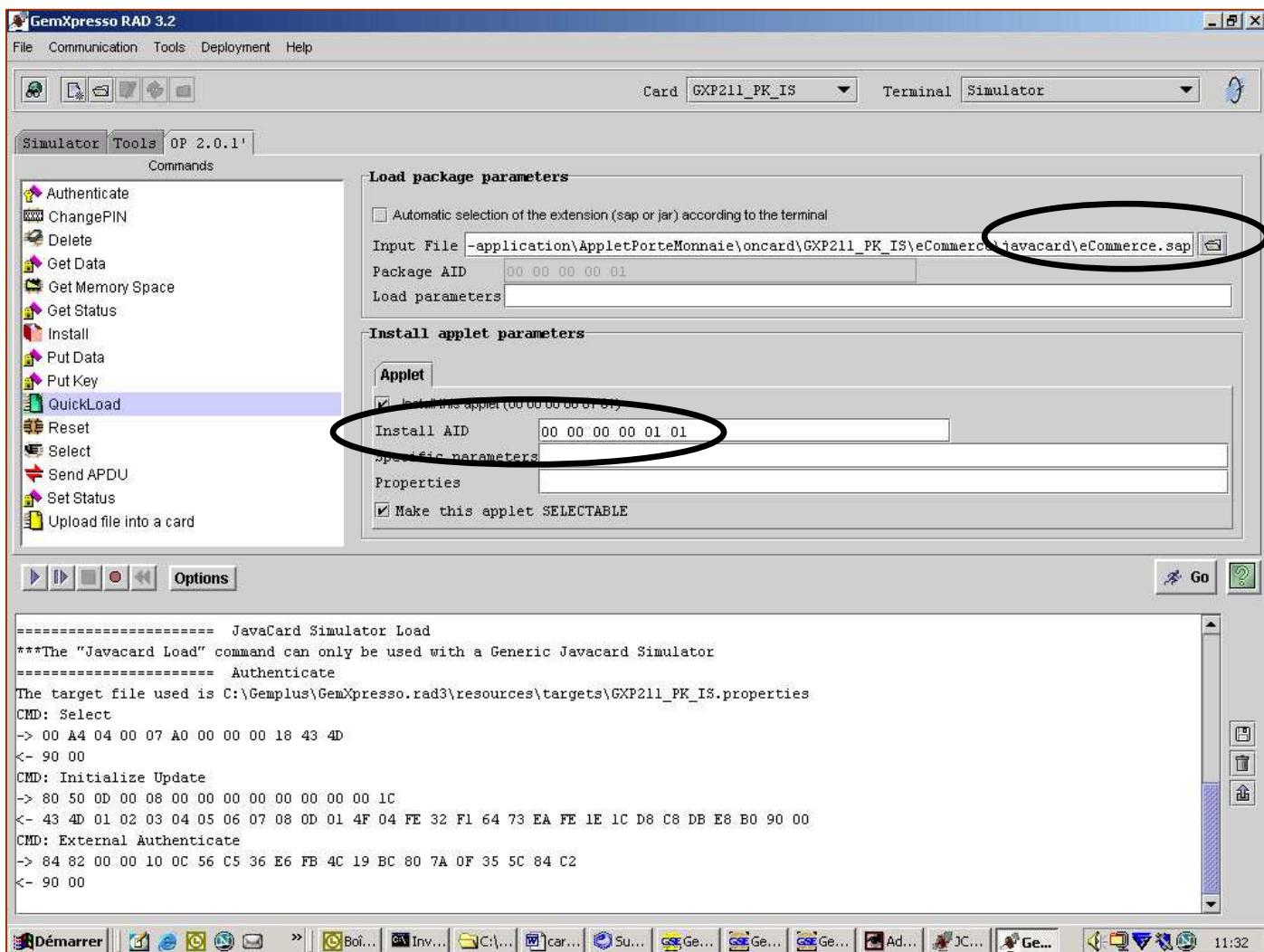
Nous pouvons à présent tester notre applet dans cet environnement certainement plus attractif que le monde fruste du simulateur CRef. Nous travaillerons exclusivement ici dans l'onglet OP 2..0.1' du JCardManager GemXpresso RAD 3.2. Dans les commandes disponibles dans le fenêtre de gauche, nous choisirons successivement les commandes suivantes :

- 1) **Authenticate** : pour trouver les caractéristiques de la carte utilisée

```
===== Authenticate
The target file used is
C:\Gemplus\GemXpresso.rad3\resources\targets\GXP211_PK_IS.properties
CMD: Select
-> 00 A4 04 00 07 A0 00 00 00 18 43 4D
<- 90 00
CMD: Initialize Update
-> 80 50 0D 00 08 00 00 00 00 00 00 00 1C
<- 43 4D 01 02 03 04 05 06 07 08 0D 01 4F 04 FE 32 F1 64 73 EA FE 1E 1C D8 C8 DB E8
B0 90 00
CMD: External Authenticate
-> 84 82 00 00 10 0C 56 C5 36 E6 FB 4C 19 BC 80 7A 0F 35 5C 84 C2
<- 90 00
```

- 2) **Quickload** : cette commande permet de charger le fichier SAP sur la "carte" simulée de l'installer avec l'applet d'installation on-card. Pour cela, il nous faut :

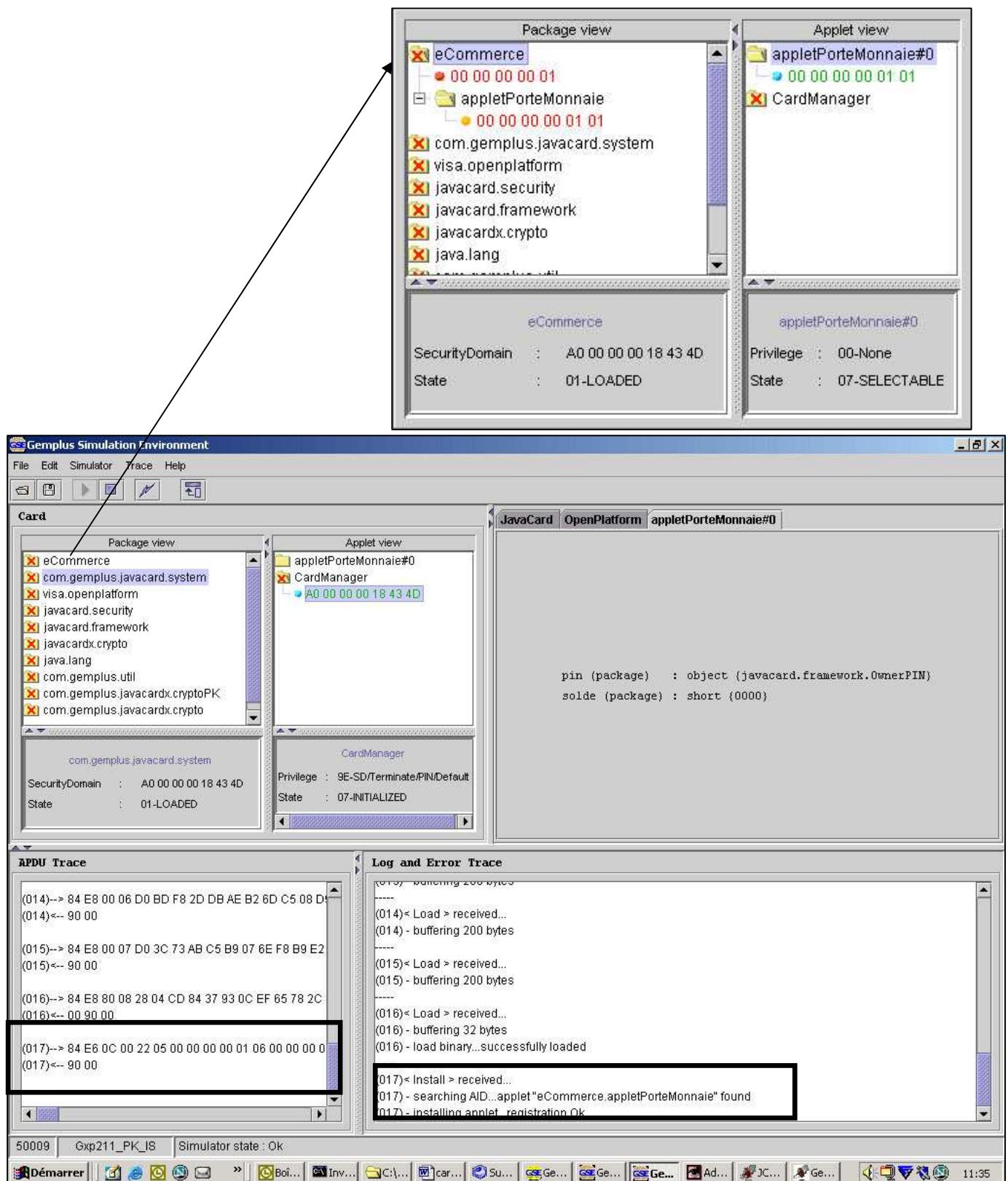
- ◆ désigner le fichier .sap dans la zone d'entrée "Input file";
- ◆ préciser l'AID de l'applet à installer.



Résultat →

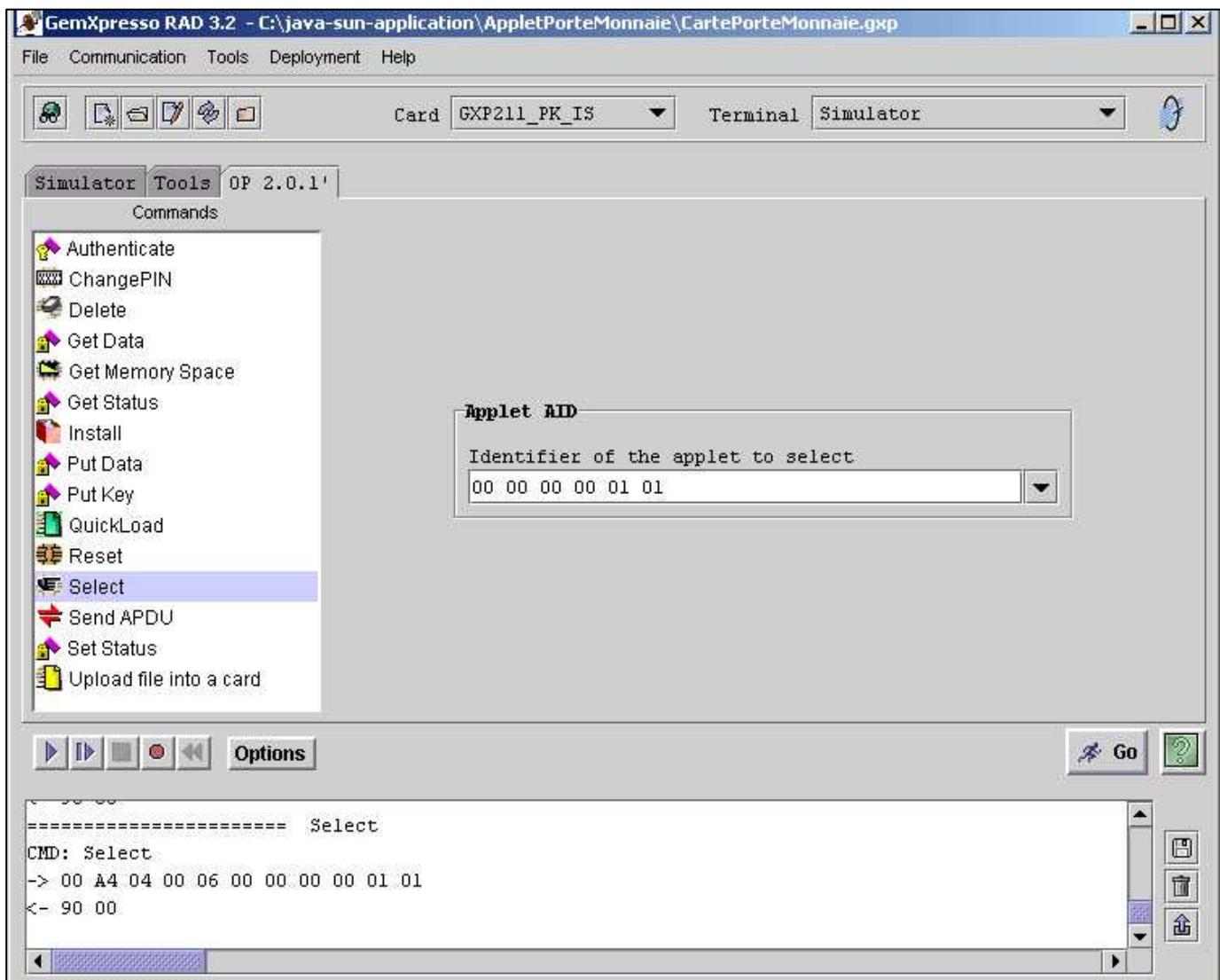
```
===== QuickLoad
CMD: Install for load
-> 84 E6 02 00 12 05 00 00 00 00 01 00 00 00 00 C9 F8 37 FF DC B0 99 2E 00
<- 90 00
9 blocks to load ...
CMD: Load File
-> 84 E8 00 00 D4 C4 82 06 60 BA CA CA FE 20 01 09 65 43 6F 6D 6D 65 72 63 65 05 00
00 00 00 01 31 66 69 6C 65 3A 2F 43 3A 2F 6A 61 76 61 2D 73 75 6E 2D 61 70 70 6C 69 63
... DF 59 1E 89 96 F6 A7 66 26 DF DE EF DC C1 A4 18 28 EB 4D AF 78
<- 90 00
block 1 loaded
CMD: Load File
...
block 9 loaded
package loaded in 1 s
===== QuickLoad
CMD: InstallApplication
-> 84 E6 0C 00 22 05 00 00 00 00 01 06 00 00 00 00 01 01 06 00 00 00 00 01 01 01 00 02 C9
00 00 F1 2D F9 A3 8C EA D8 1E 00
<- 90 00
```

On peut voir l'exécution de l'applet d'installation dans le Gemplus Simulation Environment et remarquer la présence de notre applet dans la fenêtre "Applet view" :



On peut ainsi visualiser l'exécution des commandes APDU d'installation (écran en bas à gauche) et les écritures correspondantes dans le fichier de log (en bas à droite) ☺ !

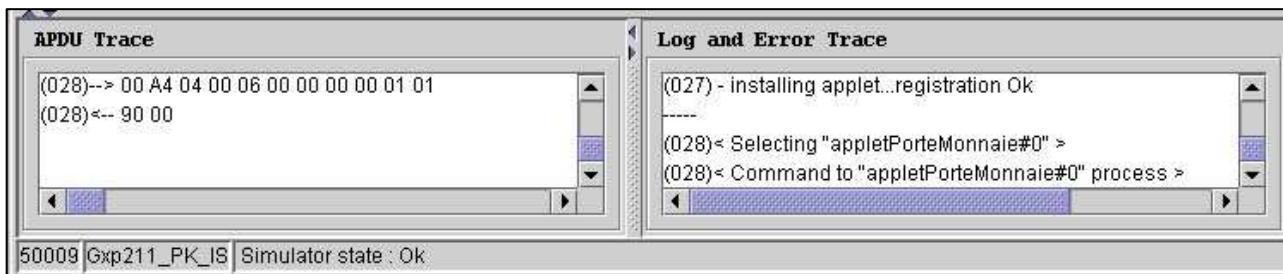
3) **Select** : Notre applet étant ainsi installée sur la "carte", on peut la sélectionner. Pour cela, on revient sur le RAD et on choisit la commande select qui sélectionne l'applet (⊕ on entre l'AID à la main !) :



Résultat →

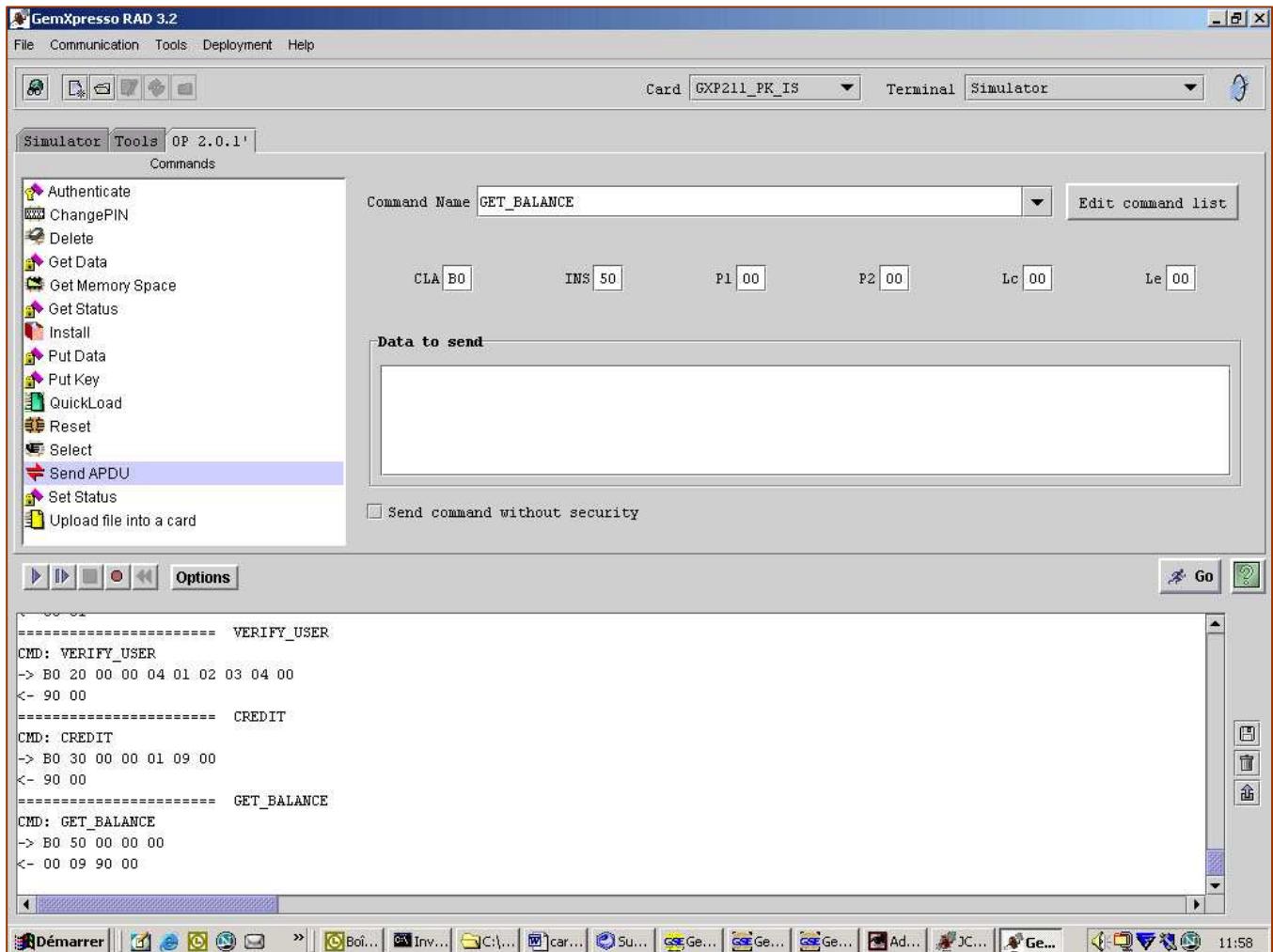
```
=====
Select
CMD: Select
-> 00 A4 04 00 06 00 00 00 00 01 01
<- 90 00
```

Le simulateur de carte reflète la sélection :

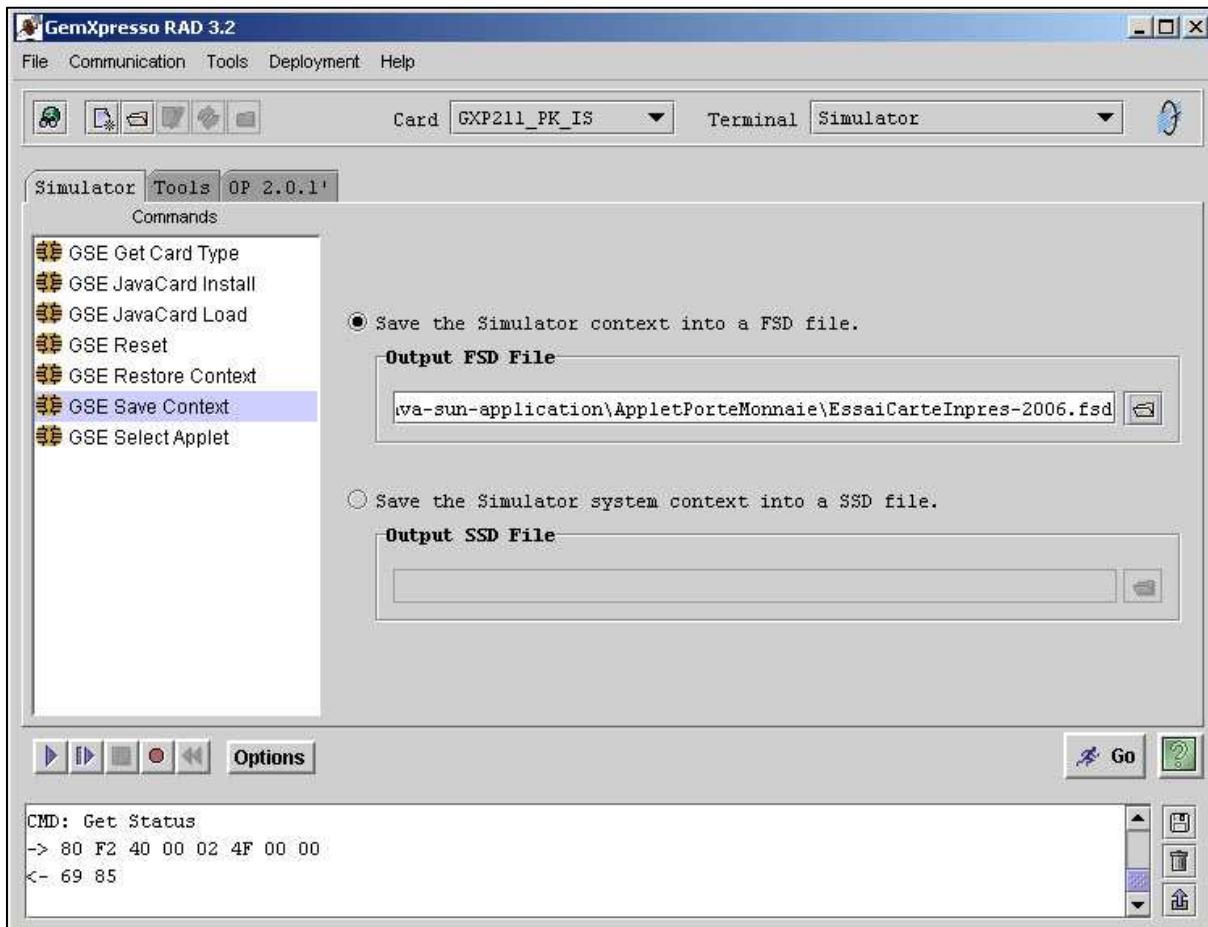


Nous pouvons donc à présent envoyer des commandes à notre applet ...

4) **Send APDU** : On exécute les commandes de l'applet : il suffit de choisir une commande et de lui donner ses éventuelles données :



5) **Sauvetage du contexte** : On peut sauver le contexte ainsi créé en passant dans l'onglet Simulator et en sélectionnant GSE Save Context :



Résultat →

===== Save Simulator Context
Data have been written in specified serialization file.

ce qui crée un fichier EssaiCarteInpres-2006.fsd (Full Serialized Data). Il convient de bien remarquer qu'en cas de chargement du contexte, il faut re-authentifier et re-sélectionner l'applet avant d'en invoquer les commandes.

17.7 Le déploiement d'un card service

Le simulateur de GemPlus va également nous servir à tester le CardService développé plus haut (appletPorteMonnaieCardService). Il faut cependant savoir que nous devrons y rectifier toutes les méthodes matérialisant une commande APDU en supprimant la ligne concernant la longueur de la réponse (et ceci également dans appletPorteMonnaieCardServiceFactory) – par exemple :

- ♦ dans **appletPorteMonnaieCardService** :

```
public void Select( ) throws CardTerminalException,
CardServiceUnexpectedResponseException
{
    ...
}
```

```

        cmd.setLength(0);                      // Taille de la commande APDU
        cmd.append((byte) 0x00);                // Classe d'instruction
        cmd.append((byte) 0xA4);                // Instruction
        cmd.append((byte) 0x04);                // Parametre 1
        cmd.append((byte) 0x00);                // Parametre 2
        cmd.append((byte) 0x06);                // Lc
        cmd.append((byte) 0x00);                // Données : AID de l'applet
        cmd.append((byte) 0x00);
        cmd.append((byte) 0x00);
        cmd.append((byte) 0x00);
        cmd.append((byte) 0x01);
        cmd.append((byte) 0x01);
//cmd.append((byte) 0x00); // Longueur attendue en réponse
...
}
    
```

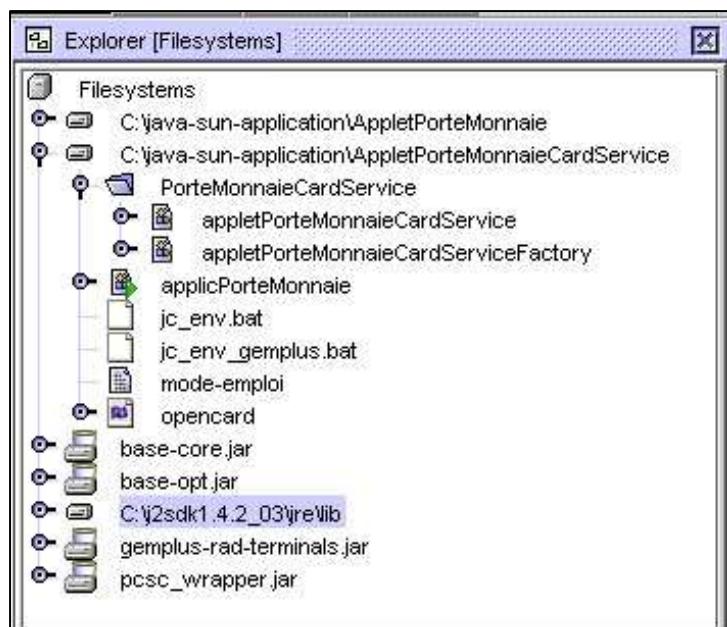
♦ dans **appletPorteMonnaieCardServiceFactory**:

```

protected CardType getCardType(CardID cid, CardServiceScheduler sched)
    throws CardTerminalException
{
    byte data[] = { (byte) 0x00,(byte) 0x00,(byte) 0x00,(byte) 0x00,
                    (byte) 0x01,(byte) 0x01 };

    ISOCommandAPDU cmd = new ISOCommandAPDU
        (//12,                      // Taille de la commande APDU
         (byte) 0x00,                // Classe d'instruction
         (byte) 0xA4,                // Instruction
         (byte) 0x04,                // P1
         (byte) 0x00,                // P2 - (byte) 0x06 // Lc est inutile car data est un tableau
         data);                     // Données
//(byte) 0x00);                  // Longueur attendue en réponse
...
}
    
```

Il faudra évidemment monter dans le projet les librairies du simulateur de GemPlus qui sont dans gemplus-rad-terminals.jar :



Enfin, il faudra modifier le fichier opencard.properties :

opencard.properties
OpenCard.services = PorteMonnaieCardService. <i>appletPorteMonnaieCardServiceFactory</i>
OpenCard.terminals = com.gemplus.opencard.terminal.GemplusRadCardTerminalFactory
Simulator SOCKETJC21SIMULATOR 127.0.0.1: 50009

L'exécution de notre application de test (après réinitialisation du simulateur) donnerait par exemple :

Début de l'application
TPDU Not Used
TPDU Not Used
Smart Card démarrée
Carte request intanciée - attente d'une carte
Envoi de la commande SELECT au JCRE :
APDU_Buffer = 00A4040006000000000101 (hex) | lc = 6 | le = -1
Response to 'SELECT' command: 9000
ATR : opencard.core.terminal.CardID@578ceb ATR: 3B 0D 80 31 80 65 B0 05 01 02 F0 83
00 90 00
Tentative de select
Envoi de la commande SELECT au JCRE :
opencard.core.terminal.CommandAPDU@1e0bc08
0000: 00 A4 04 00 06 00 00 00 01 01 00 00 00 00 00
Response to 'SELECT' command: 9000

select OK - tentative de verify
Envoi de la commande VERIFY_USER au JCRE :
opencard.core.terminal.CommandAPDU@1df073d
0000: B0 20 00 00 04 01 02 03 04 00 00 00 00 00 00
Response to 'VERIFY_USER' command: 9000

verify OK - tentative de credit
Envoi de la commande GET BALANCE au JCRE :
opencard.core.terminal.CommandAPDU@8a0d5d
0000: B0 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .P.....
Response to 'GET BALANCE' command: 9000
Montant : 1.0
Res : 00 01

Montant a traiter : 10.0
Envoi de la commande CREDIT au JCRE :
opencard.core.terminal.CommandAPDU@b1c5fa
0000: B0 30 00 00 01 0A 00 00 00 00 00 00 00 00 00 00 .0.....
Response to 'CREDIT' command: 9000

credit OK - tentative de getbalance
Envoi de la commande GET BALANCE au JCRE :
opencard.core.terminal.CommandAPDU@efd552

0000: B0 50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .P.....

Response to 'GET BALANCE' command: 9000

Montant : 11.0

Res : 00 0B

getbalance OK - tentative de debit

Montant a traiter : 6.0

Envoi de la commande DEDIT au JCRE :

opencard.core.terminal.CommandAPDU@efd552

0000: B0 40 00 00 01 06 00 00 00 00 00 00 00 00 00 00 .@.....

Response to 'DEDIT' command: 9000

...

18. La simplification (bis) : l'API smartcardio et GlobalPlatform

A l'heure actuelle, développer une application capable de dialoguer avec une applet pour Java card est devenu relativement simple. En effet, l'API javax.smartcardio et l'implémentation GPj de GlobalPlatform viennent à notre secours.

18.1 L'API javax.smartcardio

Le package **javax.smartcardio** est l'API du JDK 1.6 (JSR 268) permettant d'échanger des commandes et des réponses APDU avec des cartes à puce ISO 7816-4 via les lecteurs de carte PC/SC branchés sur l'hôte. Il définit donc en fait une API de communication entre des applications Java classiques et des applets exécutées sur une carte à puce.

Avec nos connaissances de l'OpenCard, on conçoit assez aisément le rôle des classes de ce package – le Javadoc le montre bien :

Class Summary	
<u>ATR</u>	A Smart Card's answer-to-reset bytes.
<u>Card</u>	A Smart Card with which a connection has been established.
<u>CardChannel</u>	A logical channel connection to a Smart Card.
<u>CardPermission</u>	A permission for Smart Card operations.
<u>CardTerminal</u>	A Smart Card terminal, sometimes referred to as a Smart Card Reader.
<u>CardTerminals</u>	The set of terminals supported by a TerminalFactory.
<u>CommandAPDU</u>	A command APDU following the structure defined in ISO/IEC 7816-4.
<u>ResponseAPDU</u>	A response APDU as defined in ISO/IEC 7816-4.
<u>TerminalFactory</u>	A factory for CardTerminal objects.
<u>TerminalFactorySpi</u>	The TerminalFactorySpi class defines the service provider interface.

A l'image de ce que l'on programme dans le contexte de l'OpenCard, on peut imaginer une application dialoguant avec une applet installée sur une Java Card réelle (pour nous, ce sera une carte GemAlto – l'ex GemPlus) :

ClientCard.java

```
package smartcardio;

import java.util.*;
import java.util.logging.*;
import javax.smartcardio.*;

public class ClientCard
{
    public static void main(String[] args)
    {
        byte [] Select_DF_GSM ={(byte)0xA0,(byte)0xA4,(byte)0x00,(byte)0x00,
                               (byte)0x02,(byte)0x7f,(byte)0x20};

        Card card =null;
```

```

TerminalFactory factory = null;
CardTerminal terminal = null;
ResponseAPDU r=null;
ATR atr=null;
CardTerminals cardterminals=null;
CardChannel channel =null;

// show the list of available terminals
factory = TerminalFactory.getDefault();
List<CardTerminal> terminals = null;
try
{
    terminals = factory.terminals().list();
}
catch (CardException ex)
{
    Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
}
System.out.println("Terminals: " + terminals);

// get the first terminal
terminal = terminals.get(0);

// is there a card ?
try
{
    terminal.waitForCardPresent(1000);
    if(terminal.isCardPresent())System.out.println("Card Inserted!!!");
    else System.out.println("TimeOut reached!!!");
}
catch (CardException e) {System.out.println(e.getMessage()); }

// connecting to card
try
{
    card = terminal.connect("T=0");
}
catch (CardException ex)
{
    Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
}
atr = card.getATR();
System.out.println("ATR: " + byteToString(atr.getBytes()));

channel = card.getBasicChannel();
try
{
    r = channel.transmit(new CommandAPDU(Select_DF_GSM));
}
catch (CardException ex)
{
    Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
}
System.out.println("response: " + byteToString(r.getBytes()));

```

```

try
{
    // disconnect
    card.disconnect(false);
}
catch (CardException ex)
{
    Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
}

final static char[] cnv = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A','B','C','D','E','F'};

static public String byteToString(byte[] value)
{
    String cline="";
    int i;
    char[] c= new char [3*value.length];

    for(i=0;i<value.length;i++)
    {
        c[3*i] = cnv[((int)value[i]) >> 4)&0xF] ;
        c[3*i+1] = cnv[((int)value[i]) & 0xF] ;
        c[3*i+2] = ' ';
    }
    cline= String.copyValueOf(c,0,3*value.length) ;
    return(cline);
}
}

```

18.2 Le déploiement avec GlobalPlatform/GPi

Il existe une implémentation Java de Global Platform nommée GPj : elle permet de communiquer avec toutes les cartes respectant la norme ISO 7816 (<http://gpj.sourceforge.net>). Cette implémentation est présente sous forme d'un simple jar qui utilise le package javax.smartcardio ainsi que la librairie cryptographique Bouncy Castle.

Le déploiement s'effectue en passant les paramètres nécessaires au déploiement d'une applet à méthode main de la classe GlobalPlatformService. Ces paramètres sont :

-sdaid <AID>	spécifie le security domain AID
-enc <key>, -mac <key> et -kek <key>	spécifie une clé pour l'encryptions des données, pour la création d'un MAC et pour le chiffrement d'autres clés
-visa2, -emv	spécifie la diversification des clés c'est-à-dire le fait de créer plusieurs clés à partir d'une clé de base et d'autres paramètres (comme la date, l'heure, etc.)
-delete <AID>,	supprime une applet ou un package
-deletedeps	supprime les dépendances de l'applet que l'on souhaite supprimer
-load <CAP>	télécharge l'applet sur la carte grâce au fichier CAP

-install	installe l'applet - il existe des options supplémentaires : -applet <AID> : par défaut, l'AID sera celui spécifier dans le fichier CAP; -package <AID> : par défaut, l'AID sera celui spécifier dans le fichier CAP; -param <bytes> : pour passer un paramètre à l'applet lors de l'installation
-list	liste tout ce qui se trouve sur la carte.

L'application cliente peut alors installer sur la carte l'applet avec laquelle elle va dialoguer en utilisant les services e GPj – la méthode suivante fait cela :

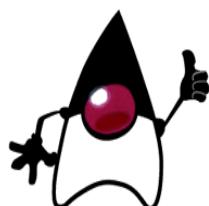
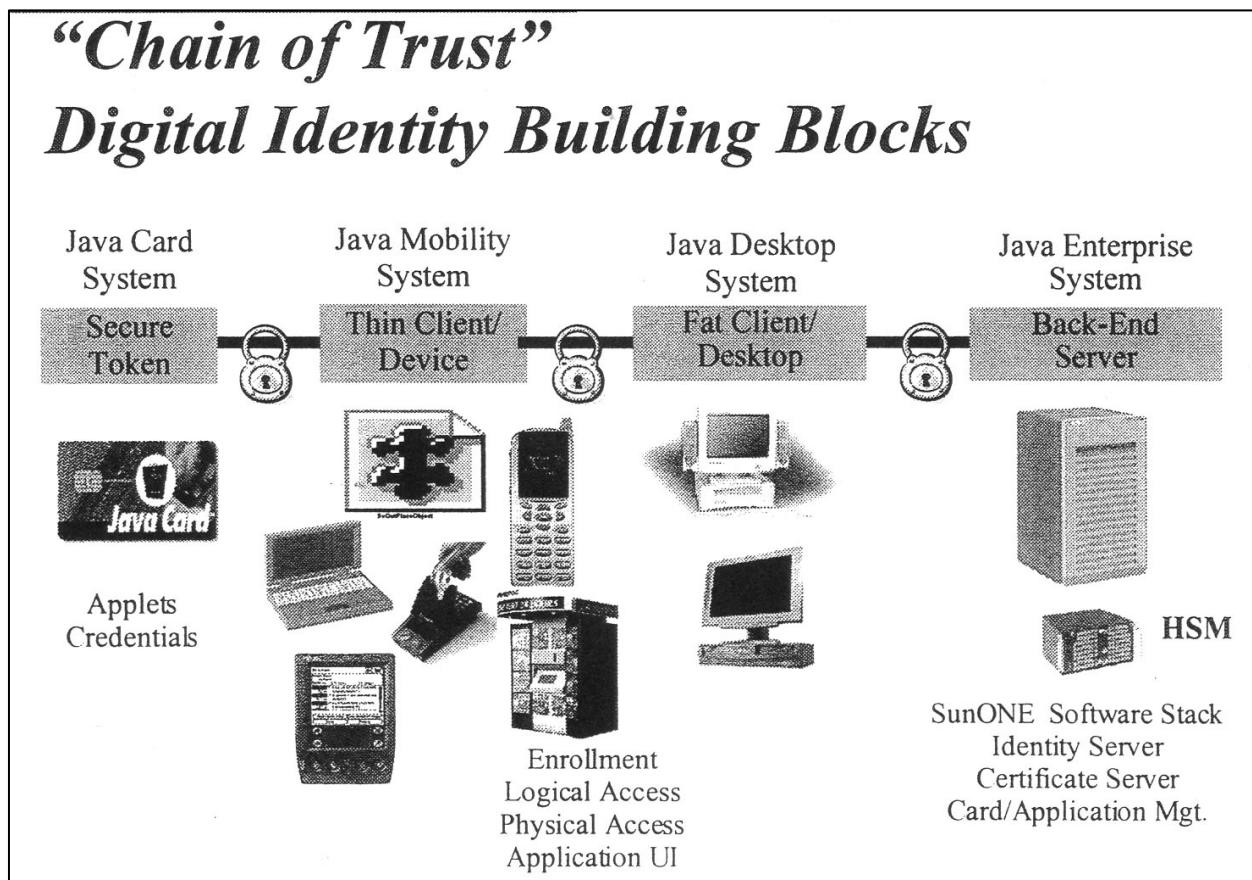
ClientCard.java (complement)

```
...
public class Main
{
    static CardTerminal terminal = null;
    ...
    private void installApplet(int cardType)
    {
        try
        {
            if (terminal.isCardPresent())
            {
                String[] param;
                if (cardType == GEMALTO_CARD)
                {
                    param = new String[]{"-sdaid", "a0 00 00 00 18 43 4d 00",
                                         "-enc", "47454d5850524553534f53414d504c45",
                                         "-kek", "47454d5850524553534f53414d504c45",
                                         "-mac", "47454d5850524553534f53414d504c45",
                                         "-visa2",
                                         "-delete", "000000000101",
                                         "-delete", "0000000001",
                                         "-load", "Applet.cap",
                                         "-install"};
                }
                else {
                    param = new String[]{"-delete", "000000000101",
                                         "-delete", "0000000001",
                                         "-load", "Applet.cap",
                                         "-install"};
                }
                GlobalPlatformService.main(param);
            }
        }
    }
}
```

```
        catch (CardException ex)
            {Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex); }
        catch (IOException ex)
            {Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex); }
    }
...
}
```

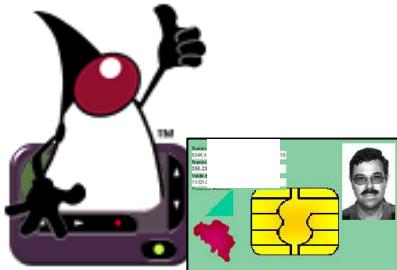
19. En conclusion : la chaîne de sécurité des plates-formes Java

La carte à puce est aussi un media d'identification – il suffit de penser aux nouvelles cartes d'identité électroniques. On a ainsi coutume de présenter la chaîne d'identification selon le schéma global suivant :



Quittons les cartes à puce tout en y restant :-o : passons à une fierté nationale (non, pas les frites, ni les moules, ni le jambon d'Ardenne) ...

XXX. La carte d'identité électronique belge



Je me révolte, donc je suis.

(A. Camus, L'Eté)

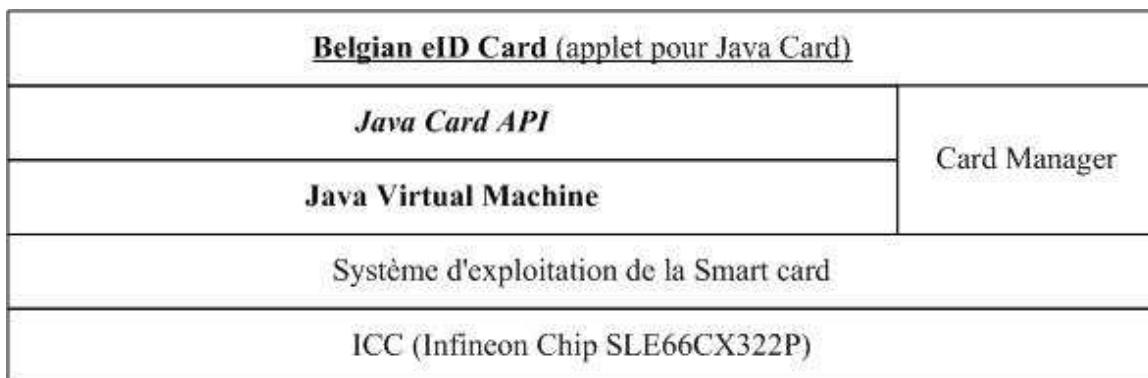
1. Une carte à puce pour carte d'identité

La Belgique a récemment décidé de doter tous ses citoyens âgés de plus de 12 ans d'une carte d'identité "électronique", c'est-à-dire se présentant sous la forme d'une carte à puce au format analogue à celui des cartes bancaires déjà bien connues. En fait, d'autres pays ont effectué la même démarche (ou envisagent de le faire), mais avec des cartes différentes ... la standardisation devra donc encore faire son œuvre.

Du point de vue technique, cette carte à puce d'identité comporte :

- ◆ un processeur 16 bits;
- ◆ un coprocesseur cryptographique avec une composante pour RSA et une pour DES;
- ◆ une ROM de 136 kB;
- ◆ une EEPROM de 32 kB;
- ◆ une RAM de 5 kB.

Il s'agit en fait d'une Java Card qui comporte une JVM GEOS et fait tourner une applet "Belgian eID Card" :



2. Les informations contenues dans une carte d'identité

On trouve imprimés sur la carte d'identité belge :

1) au recto : le nom de famille, les prénoms (les 2 premiers prénoms et la première lettre du troisième), les lieu et date de naissance, le sexe, la nationalité (!), le numéro de la carte, les dates de début et de fin de validité, la signature manuscrite et la photo du titulaire;

2) au verso : le numéro du registre national, le lieu d'émission de la carte et la signature manuscrite du fonctionnaire communal qui l'a délivrée.

On remarquera que l'adresse n'est pas imprimée, ce qui permet de conserver sa carte lorsque l'on déménage. Cette adresse est donc mémorisée dans la mémoire de la carte à puce (rue, numéro, code postal et localité) en plus de tous les renseignements visibles sur les deux faces de la carte et d'informations complémentaires comme le rang (prince, ...), un éventuel statut spécial (aveugle, ...), le fait qu'il s'agit d'une carte belge, européenne, diplomatique, etc.

3. Les informations cryptographiques

Mais l'intérêt majeur de la carte d'identité électronique est de contenir des "**certificats**" cryptographiques (au format X509v3) qui vont permettre au propriétaire de la carte de se faire reconnaître de manière sûre.

3.1 Les trois niveaux de certificats de l'identité électronique belge

On se souviendra que la techniques des certificats repose sur le principe des chaînes de certificats : chaque certificat est signé par une autorité supérieure, celle-ci possédant un certificat signé par une autorité encore plus élevée, jusqu'à parvenir ainsi à une autorité supérieure de référence connue mondialement.

Dans le cas de l'identité électronique belge, il existe trois niveaux :

- 1) au niveau le plus bas, il s'agit des **certificats des citoyens**, déjà évoqués ci-dessus; pour rappel, les clés sont des clés RSA de 1024 bits;
- 2) ces certificats personnels sont signés par une autorité intermédiaire appelée **Citizen CA**, qui constitue donc le 2^{ème} niveau avec d'autres autorités qui signent des certificats particuliers permettant aux agents communaux de réaliser certaines opérations sur les cartes; les clés sont des clés RSA de 2048 bits;
- 3) le **Belgium Root CA** est le 3^{ème} niveau, celui qui signe les certificats du niveau intermédiaire; les clés sont des clés RSA de 2048 bits; le propre certificat du Belgium Root CA est un certificat **Global Sign** – on atteint ainsi le niveau suprême avec une autorité dont les certificats sont en possession d'une large frange des logiciels courants et qui peuvent être obtenus facilement de manière sûre.

3.2 Les ressources cryptographiques disponibles

Plus précisément, la carte se présente un peu comme un keystore puisqu'elle comporte :

- 1) **trois clés privées RSA** (de 1024 bits) qui ne peuvent quitter la carte :
 - 1.1) une clé privée de signature;
 - 1.2) une clé privée d'authentification du propriétaire;
 - 1.3) une clé privée qui sera utilisée quand il s'agira de communiquer avec le Registre National (RRN), ce qui impliquera une authentification mutuelle;
- 2) les **certificats correspondants**, contenant les **clés publiques**, associés aux deux premières clés privées citées ci-dessus; la clé publique permettant le dialogue avec le Registre National n'est en effet pas placée dans un certificat, mais est conservée dans une base de donnée du Registre; plus précisément, on distingue :
 - 2.1) un "**certificat de signature**", c'est-à-dire un certificat dont la clé privée correspondante est utilisée pour signer les données transmises; autrement dit, il contient la clé publique correspondante qui va permettre d'authentifier ces données (c'est-à-dire de vérifier l'identité du signataire); ce certificat constitue donc une espèce d'équivalent de d'une "key entry" et "trusted certificate entry" de keystore, avec une clé publique dans un certificat attesté par une

autorité de certification; signalons encore qu'un mineur ne dispose pas de la faculté de signer électroniquement : son certificat de signature est révoqué jusqu'à sa majorité;

2.2) un "**certificat d'authentification**", c'est-à-dire un certificat permettant au propriétaire de la carte de prouver son identité à un système demandeur, qui en vérifiera la validité, le plus souvent auprès de l'autorité de certification qui l'a émis (référentiel CSP - **Certification Status Protocol**) et qui possède notamment des listes de révocation (CRL - **Certificate Revocation List**); ce certificat est donc encore une fois l'équivalent d'une "trusted certificate entry" de keystore;

3) un **certificat de Citizen CA** - il s'agit encore une fois de l'équivalent d'une "trusted certificate entry" pour une certificat de niveau intermédiaire; il permet de valider les deux certificats précédents;

4) un **certificat de Belgium Root CA** – il s'agit donc de l'équivalent d'une "trusted certificate entry" pour une certificat de niveau supérieur.

Les trois clés et les 4 certificats sont stockés dans l'EEPROM de la carte, dans un répertoire BelPIC. Tout comme le keystore, et en particulier ses key entries protégés par des mots de passe, l'accès à un certificat de signature réclame l'introduction du code PIN de la carte (seulement 3 essais sont admis avant le blocage de la carte).

Précisons encore que ces certificats contiennent un DN (*distinguished name*) qui comporte les attributs suivants :

- ◆ C (country) = BE
- ◆ CN (common name) = *Prénom Nom* (Authentification) ou (Signature)
- ◆ SN (surname) = *Nom*
- ◆ G (given name) = *Prénom1 Prénom2*
- ◆ OID.2.5.4.5 = *NuméroRegistreNational*

4. Les deux services fondamentaux

Avec de telles ressources, la carte d'identité électronique peut rendre plusieurs services, outre celui de fournir les renseignements généraux : l'authentification et la signature.

4.1 Le service d'authentification

Le possesseur de la carte peut **prouver son identité** à un système informatique au moyen du **certificat d'authentification** contenu sur la carte. Ce dernier ne peut être accédé que si l'utilisateur introduit son code PIN : le microprocesseur de la carte vérifiera si ce code est bien celui qui est mémorisé. L'authentification la plus simple (*légère*) réalisée par un serveur consiste à vérifier le certificat d'authentification qui lui est envoyé. Il pourra notamment voir notamment voir si il n'a pas été révoqué (par exemple suite à un vol). Cette vérification de statut du certificat (valide, suspendu, révoqué) se fait par une recherche dans les bases de données de l'autorité de certification (le CA) : on peut donc facilement modifier ce statut (par exemple, encore une fois, en cas de vol). L'authentification peut cependant être plus poussée (*lourde*) en utilisant un mécanisme de signature de challenge envoyé par le serveur (voir 9.1 plus loin).

4.2 La signature électronique

Le propriétaire de la carte peut utiliser la clé privée associée à son **certificat de signature** pour **signer un message électronique** ou un document électronique quelconque (fichier Word et Excel de MS-Office, Writer et Calc d'Open Office, pdf). Plus précisément, le logiciel qui veut obtenir une signature va envoyer le hachage du document à signer à la carte (c'est l'algorithme SHA-1 qui est utilisé). A nouveau, *l'utilisateur devra introduire son code PIN pour permettre l'accès à sa clé privée*. En cas de succès, la carte peut construire la signature correspondante (en cryptant le hachage avec la clé privée) et la renvoyer au logiciel.

5. La vérification des certificats

A priori, les certificats peuvent être vérifiés de deux manières :

- ◆ utiliser localement des fichiers **CRL** (Certificate Revocation List), c'est-à-dire une liste de certificats révoqués; cette liste qui doit évidemment être tenue constamment à jour (typiquement, toutes les 3 heures), via HTTP ou HTTPS; l'inconvénient est évidemment la charge de trafic réseau (ces listes sont le plus souvent de grandes tailles)
- ◆ s'adresse à un serveur chargé de répondre aux demandes de vérification de validité; ce serveur utilise le protocole **OCSP** (Online Certification Status Protocol); la charge réseau est largement moindre qu'avec la méthode CRL et, de plus, la vérification se fait en temps réel.

En fait, les certificats contiennent des champs donnant des informations pour la vérification :

- ◆ un champ désigne le site Web de référence consacré à la validité des certificats – ici, il s'agit de <http://repository.eid.belgium.be> :

The screenshot shows the eid services website (<http://repository.eid.belgium.be>) with the following details:

- Header:** eid services and Certipost Belgacom & De Post / La Poste.
- Language:** nl | fr | de | en
- Main Navigation:** Home, Statut certificats, Téléchargez certificats (Root CA, Citizen CA, Foreigner CA, Government), Documents, Achat cartes de test, Contact & Info.
- Left Sidebar:** Carte perdue? with a red "eid CARD STOP" sign containing the phone number 02 518 21 17.
- Section:** **Statut des certificats**
Consulter et vérifier le statut d'un certificat en utilisant l'un des outils suivants:
 - Certificate Status Web Service**: Represented by an icon of a computer mouse.
 - Certificate Revocation List Lookup Service**: Represented by an icon of a magnifying glass over a document labeled "CRL".
- Text:** Utiliser le "Certificate Status Web Service" pour vérifier en ligne le statut d'un certificat. Télécharger la "Certificate Revocation List (CRL) lookup Service" pour vérifier le statut d'un certificat.
L'état actif, suspendu ou révoqué d'un certificat peut également être vérifier via un logiciel client OCSP (Online Certificate Status protocol) en se connectant au serveur OCSP à l'adresse Internet suivante : <http://ocsp.eid.belgium.be>.
- Footer:** © COPYRIGHT CERTIPOST nv. ALL RIGHTS RESERVED.

qui donne accès d'une part au service de recherche par OCSP :

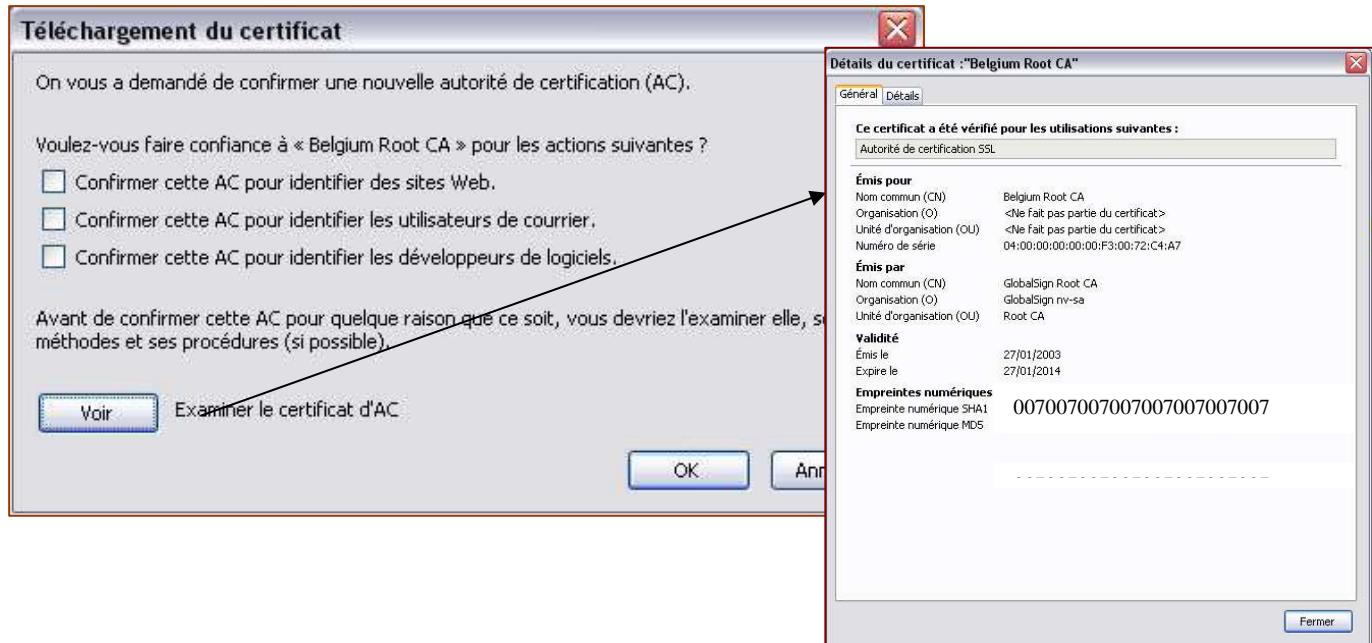
The screenshot shows the homepage of the ".be BELGIUM FEDERAL PORTAL". The title "SERVICE WEB DU STATUT DES CERTIFICATS" is at the top. Below it, a message says "Bienvenue Sur Les Pages Web Du Statut Des Certificats.". A text input field asks for the certificate serial number, with an example provided: "Pour retrouver le statut d'un certificat, veuillez introduire son numéro de série (exemple '10 00 00 00 00 06 85 04 fe 03 72 c1 f7 ff 80')..". A button labeled "Rechercher" is below the input field. The URL is http://ocsp.eid.belgium.be/

The screenshot shows the search results for a certificate. It displays the message "Le statut du certificat du CitizenCA200514 comportant le numéro de série 1 ... est: Valide (le certificat est valide)". The URL is http://ocsp.eid.belgium.be/

et d'autre part au service de téléchargement des CRL : <http://crl.eid.belgium.be/eidc200508.crl> – on peut alors télécharger le fichier en question :

The screenshot shows the homepage of the ".be BELGIUM FEDERAL PORTAL" for revocation lists. The title "SERVICE WEB DE RECHERCHE DE LA LISTE DES CERTIFICATS EID REVOQUÉS" is at the top. Below it, a message says "Bienvenue Sur La Page De Téléchargement De La Liste Des Certificats Révoqués (CRL) Concernant L'eID". A text input field asks for the date and time to search for the CRL. A modal dialog box titled "Statut d'import de la LCR" is open, stating "La liste de certificats révoqués (LCR) a été importée avec succès." It also shows the last update date: "Prochaine mise à jour le : 3/04/2007". A question is asked: "La mise à jour automatique n'est pas activée pour cette LRC. Voulez-vous activer la mise à jour automatique ?" with "Oui" and "Non" buttons. The URL is http://crl.eid.belgium.be/

- ◆ un autre champ désigne le fichier contenant le statut du certificat, par exemple
- ◆ un troisième champ fournit l'URL du certificat Belgian Root CA
<http://certs.eid.belgium.be/belgiumrs.crt> afin de télécharger celui-ci :



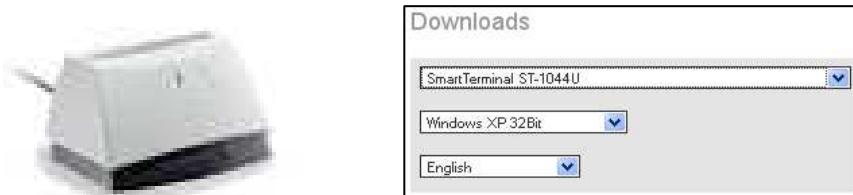
6. Les lecteurs de cartes

Tous les lecteurs de cartes à puce ne conviennent pas pour lire la carte d'identité électronique. Le site <http://www.cardreaders.be> permet d'obtenir une liste de lecteurs compatibles. Nous avons utilisé ici trois types de lecteurs :

1) le lecteur de cartes **ACR30** (USB 1.1) ou **ACR38U** (USB 2.0) de **ACS** (Advanced Card Systems Ltd) : il suffit d'aller sur <http://www.belgeid.be/> et download "programme de commande du lecteur de cartes" puis "MSI Installer for PC/SC driver – ..., Win XP, ...". On se retrouve avec un ACR38U_MSI_Winx86_1161_P.zip qui faut décompresser puis lancer l'exécutable setup.exe. Le driver sera mis en place (par exemple sous le nom de C:\notes-java-2006\carte identite\ACR38U_inst_111001_P\ACR38U_inst_11101_P\Drivers\USB\XP2K\A38usb.sys) et il n'y a plus qu'à brancher le lecteur ...



2) le lecteur **SmartTerminal 1044** (USB 2.0) de **Cherry** dont l'installation est instantanée sous certains Windows-XP ou Seven (le driver est disponible sur <http://www.cherry.de/english/service/download.php>); son pied lesté n'est pas seulement une coquetterie ...



3) le lecteur de cartes **GemPC USB** (USB 2.0) ou **GemPC Twin de GemPlus** (USB 2.0 et RS-232): l'installation du premier type se limite à un CD en autorun et une connexion USB à la machine, tandis que pour la version "twin" (USB et série), il suffit d'aller sur <http://store.gemplus.com/is-bin/INTERSHOP.enfinity/eCS/Store/en/-/-/Storefront-Start> pour aller chercher une "database" d'installation; il n'y a plus qu'à pluguer le lecteur et suivre le wizzard d'installation de Windows.



7. Le middleware d'utilisation et l'enregistrement des certificats

Le FEDICT, le Service public fédéral Technologie de l'Information et de la Communication, distribue un logiciel permettant de communiquer avec la carte d'identité placée dans un lecteur agréé. Ce software est basé sur le projet libre **OpenSC**. Il existe actuellement dans la version 2.5.9 et est disponible sur <http://eid.belgium.be> : on y obtient un exécutable extractable Belgian_Identity_Card_Run-time_2.5.9.exe.

L'installation se réalise sans problème. Mais il faut remarquer qu'elle peut s'installer en même temps comme un service (Belgian Identity Card Service – "service vie privée") qui empêche toute application non autorisée d'accéder au lecteur (les applications autorisées étant Internet Explorer et Outlook). Quand on utilise Firefox ou Thunderbird, on a donc tout intérêt à éviter non pas l'installation mais bien l'activation de ce service : il suffit de répondre non à la question correspondante dans le processus d'installation.

Une fois celle-ci terminée, on trouve dans la liste des programmes :

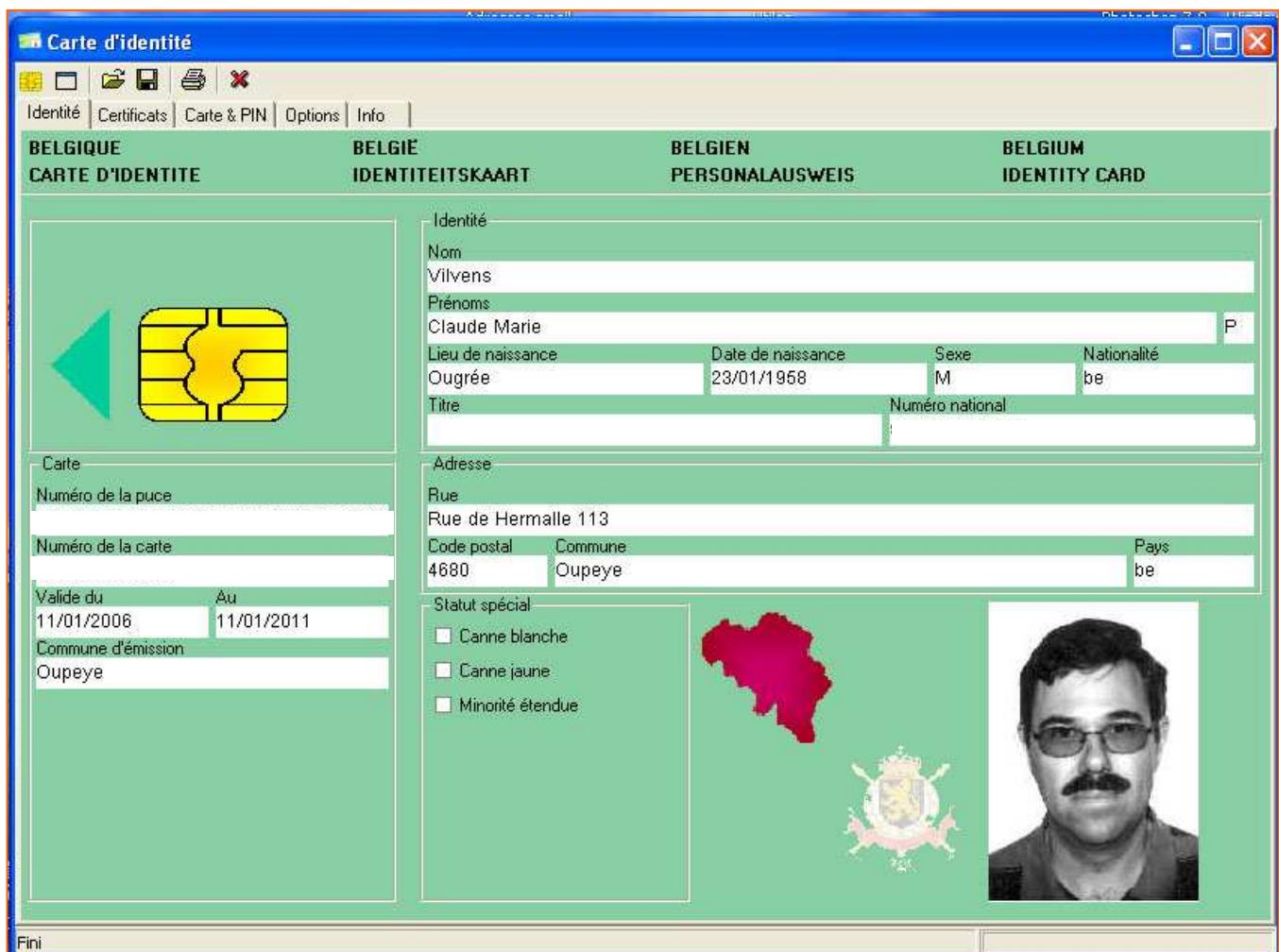


Le fait de lancer l'application ne provoquera aucune réaction immédiate. Mais le simple fait de placer une carte d'identité dans le lecteur provoquera le chargement des certificats des deux des autorités de certification dans le magasin à certificats de Windows, ainsi que nous le confirme l'apparition des messages :

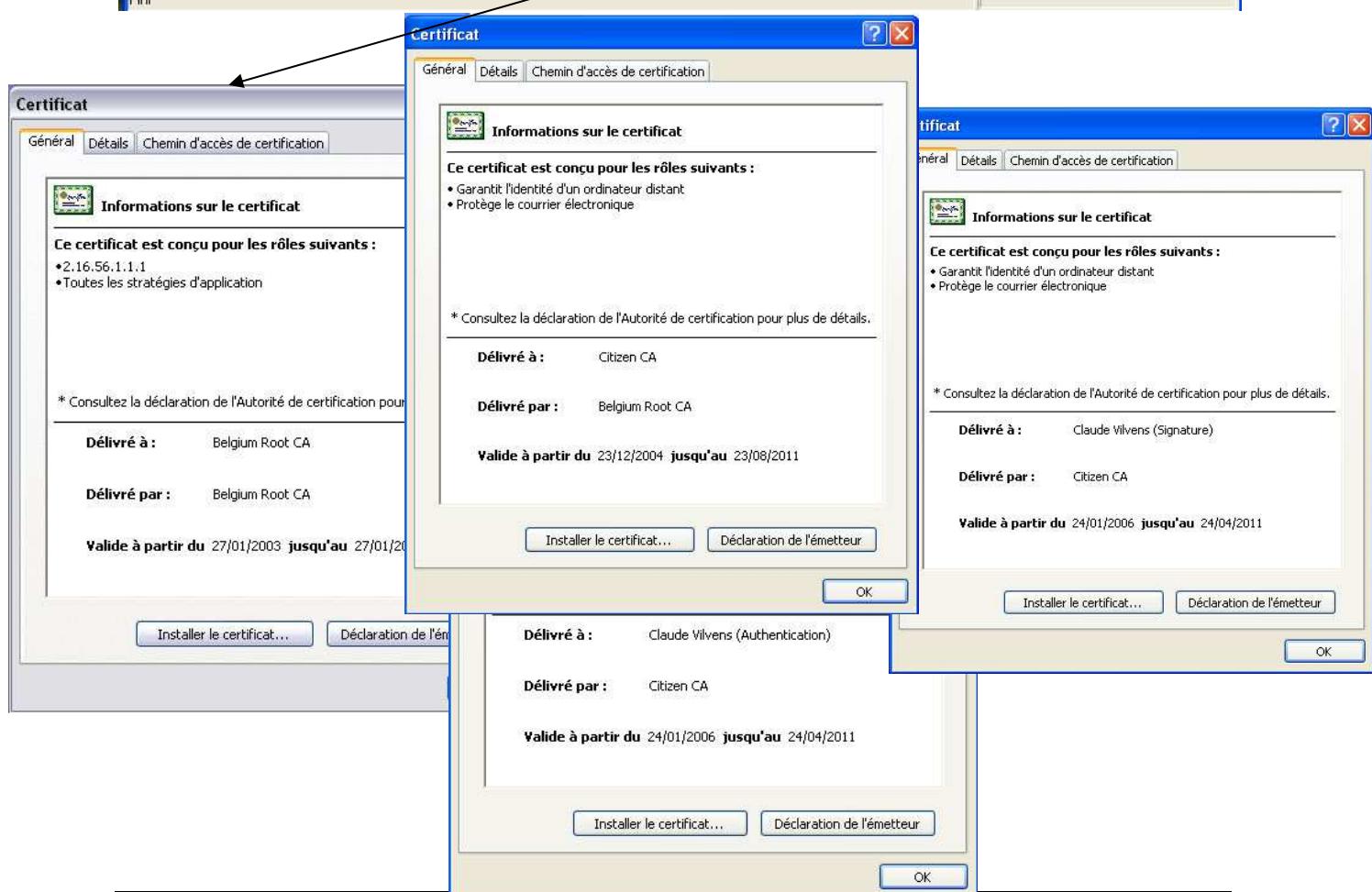
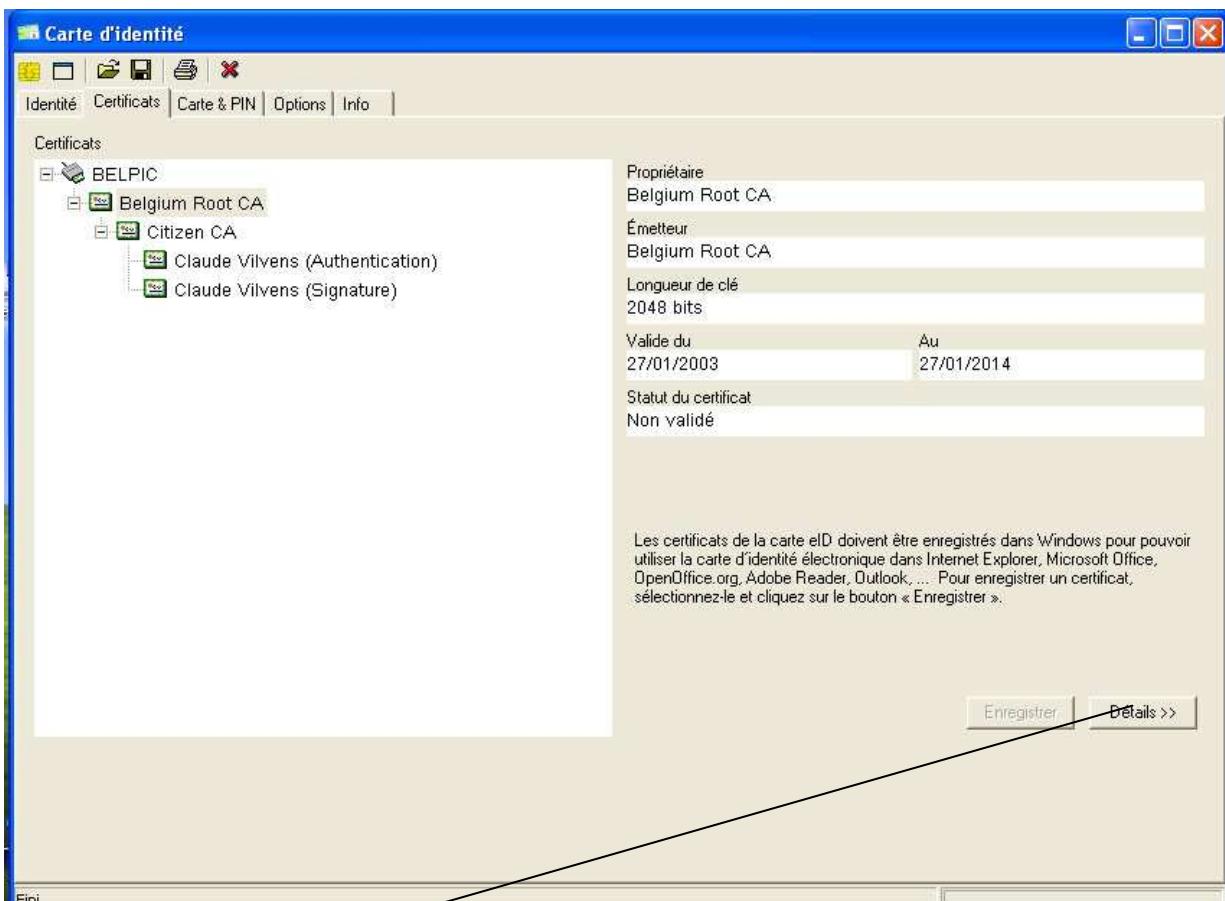




Le programme lui-même donne (après un clic dans l'icône représentant une puce si il avait été lancé avant l'introduction de la carte – et avec un voile pudique sur certaines informations ;-)) :



Les différents onglets permettent évidemment d'obtenir des informations complémentaires, portant notamment sur les certificats :



On peut donc enregistrer ses certificats dans le magasin à certificats de Windows (bouton "Installer le certificat"). Il est également possible de modifier son code PIN (si on connaît l'ancien !).

8. Le magasin à certificats de Windows

On peut vérifier que l'opération a bien été réalisée en lançant la Microsoft Management Console :

| C:\Documents and Settings\Claude>**mmc**

Il suffit de suivre le cheminement :

Fichier → Ajouter/Supprimer ... → bouton "Ajouter" → choisir "Certificats" dans la liste → bouton "Ajouter" → choisir dans la liste à puces : "Mon compte d'utilisateur" → boutons "Terminer"/"Fermer"/"Ok"

pour découvrir que les 4 certificats sont bien en place – par exemple :

Racine de la console\Certificats - Utilisateur actuel\Autorités de certification racines de confiance\Certificats	Délivré à	Délivré par	Date d'expiration	Rôles
Racine de la console				
Certificats - Utilisateur actuel				
Personnel				
Certificats				
Autorités de certification racines de confiance				
Certificats				
Confiance de l'entreprise				
Autorités intermédiaires				
Liste de révocation des certificats				
Certificats				
Objet utilisateur Active Directory				
Éditeurs approuvés				
Certificats non autorisés				
Certificats				
Le magasin Autorités de certification racines de confiance contient 109 certificats.				

(on peut parvenir au magasin à certificats directement par Exécuter... → certmgr.msc).

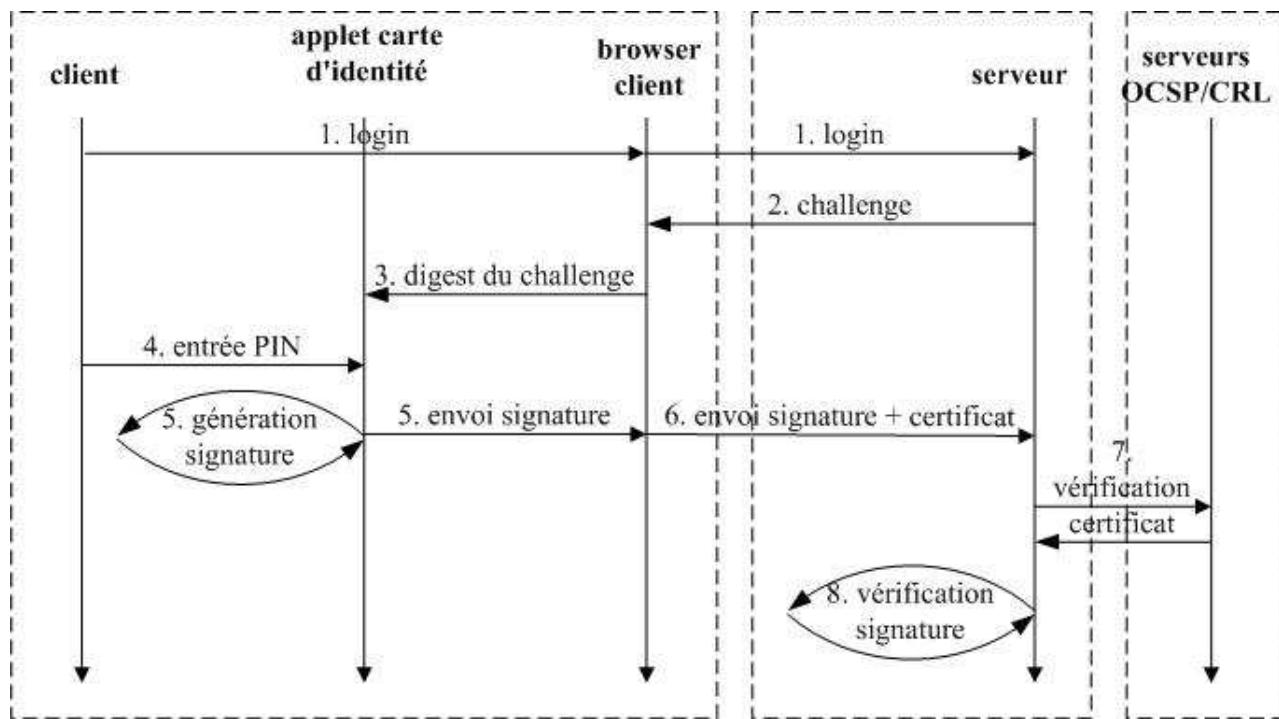
Pour que ces certificats soient utilisables par une application, il faut les configurer. Nous donnerons ici deux exemples de telles configurations pour le logiciel de messagerie électronique Thunderbird et le browser Firefox. Mais dans quel monde ces applications travaillent-elles ?

9. Le principe des fonctionnalités cryptographiques de la carte d'identité

9.1 L'authentification

Le mécanisme d'authentification ne doit pas être confondu avec celui de la signature : il s'agit simplement ici, sur base de son certificat d'authentification, de prouver qui l'on est à un correspondant réseau, typiquement un serveur. Le mécanisme est nettement plus sûr qu'une simple identification par login-mot de passe, comme nous allons le constater. Les étapes successives sont les suivantes, si Alice veut accéder à un serveur Server :

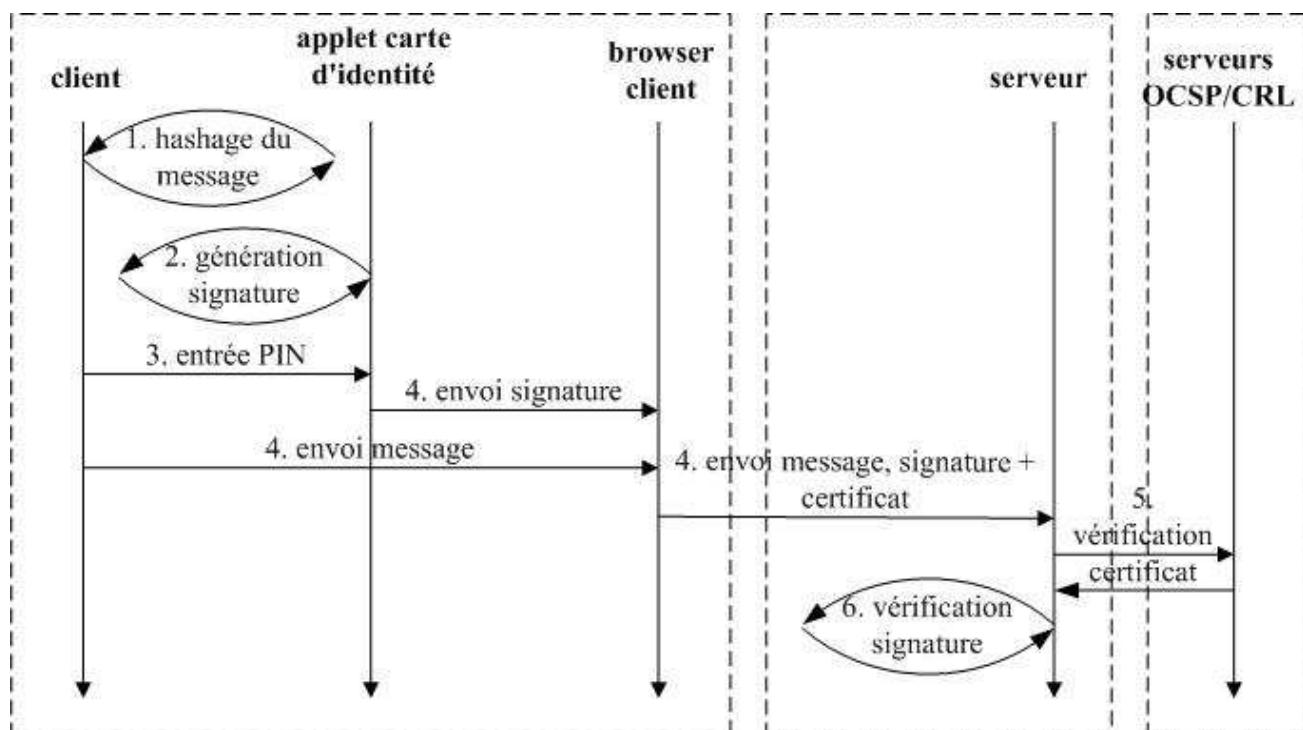
1. Alice, par l'intermédiaire de son browser, envoie une requête de login à Server;
2. en réponse, Server envoie un challenge (c'est-à-dire un message aléatoire) au browser d'Alice,
3. le browser génère un digest haché et l'envoie à l'applet de la carte d'identité pour signature;
4. Alice entre son code PIN, confirmant ainsi qu'elle autorise la signature;
5. l'applet de la carte construit la signature en cryptant le digest au moyen de la clé privée d'Alice qui se trouve sur la carte (et qui, pour rappel, ne peut en sortir);
6. le browser d'Alice envoie la signature et le certificat d'authentification d'Alice (trouvé à partir de la carte) à Server;
7. Server vérifie ce certificat par un mécanisme CRL ou OCSP;
8. en cas de succès, il vérifie la signature en décryptant celle-ci avec la clé publique d'Alice : le succès dépend évidemment du fait de retrouver ou pas le digest haché challenge d'origine.



9.2 La signature

Ici, il ne s'agit plus de se faire reconnaître, mais de signer un document électronique qui n'est plus un challenge, mais un document dont le contenu a une importance pour le client. Les étapes successives sont les suivantes, si Alice veut envoyer un message authentifié au serveur Server :

1. Alice commence par signer son message; pour rappel, cela revient d'abord à calculer un digest haché de celui-ci, ...
2. puis à crypter ce digest avec sa clé privée :pour ce faire, elle fera appel à l'applet de la carte d'identité qui utilisera la clé privée de signature contenue dans la carte d'identité;
3. Alice entre son code PIN, confirmant ainsi qu'elle autorise la signature;
4. le browser d'Alice envoie le message, la signature et le certificat d'authentification d'Alice (trouvé à partir de la carte) à Server;
5. Server vérifie ce certificat par un mécanisme CRL ou OCSP;
6. en cas de succès, il vérifie la signature – pour rappel, en décryptant celle-ci avec la clé publique d'Alice et en comparant le digest obtenu avec un digest local calculé sur le message reçu.

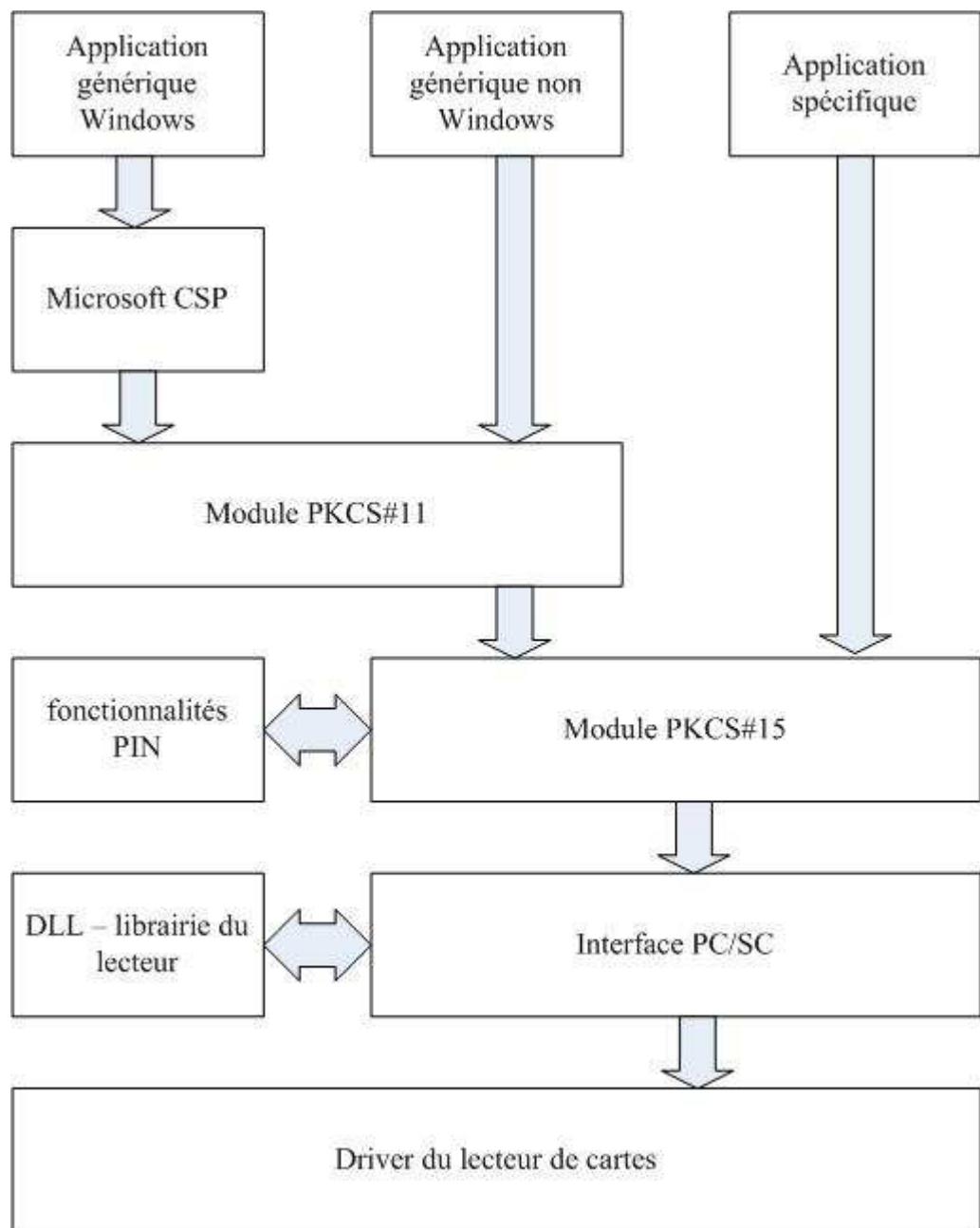


Voilà pour les principes. Reste à formaliser l'accès des applications aux informations cryptographiques qui leur permettront d'appliquer ces principes d'authentification et de signature.

10. L'accès des applications aux informations cryptographiques

10.1 L'architecture générale

Les applications qui vont accéder aux informations de la carte d'identité vont s'inscrire dans l'architecture suivante :



Les applications spécifiquement Windows utiliseront les fonctions **CSP** (Cryptographic Service Provider) de Microsoft. Celles-ci utiliseront ensuite des interfaces génériques, que les applications génériques utiliseront directement : **PS/SC** nous est bien connu (voir chapitre précédent), mais que sont les autres ?

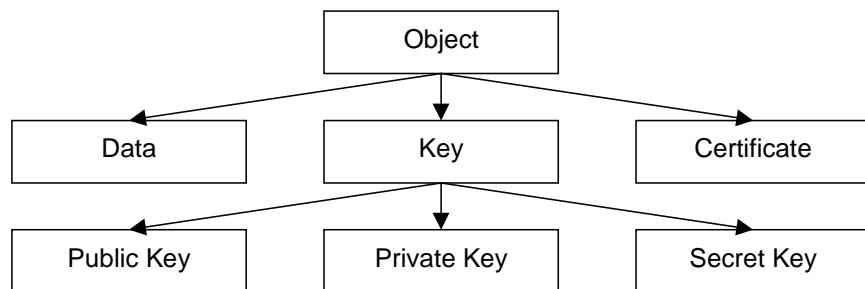
10.2 Le module PKCS#11

Pour rappel, les **PKCS** (Public Key Cryptography Standards) sont des normes de fait publiées par RSA Data Security Inc. La norme n°11 définit, sous forme d'APIs, un interface standard d'accès aux données cryptographiques mémorisées dans un dispositif hardware.

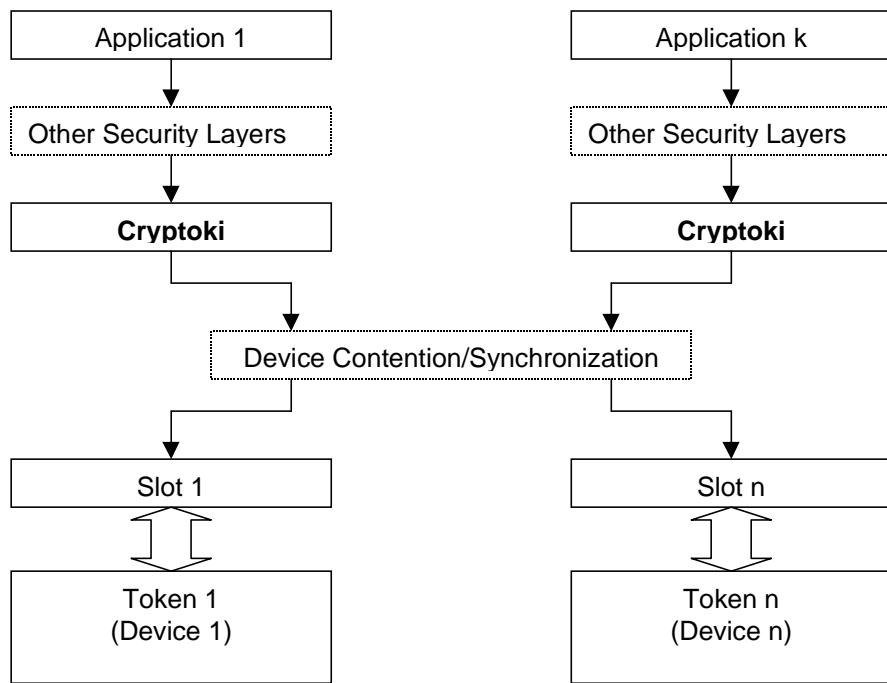


Plus précisément, **PKCS#11**, familièrement appelé **Cryptoki**, définit un interface de programmation en C qui donne à tout dispositif cryptographique une vue logique (c'est-à-dire sans spécification de format de stockage) appelée un **token** (jeton). Un token est caractérisé par diverses propriétés, également accessible par l'API, comme un nom, une description, un numéro de série, un nombre maximum de sessions, les tailles minimum et maximum des PINs

utilisés. L'accès aux divers tokens se fait par un **slot**, caractérisé par une description, le fait d'être hardware ou software, d'être amovible ou pas, d'être effectivement raccordé à un token ou pas. On constate donc que la vérité physique du stockage des informations cryptographiques peut être extrêmement variable, puisque tout est dissimulé derrière l'API. Celui-ci manipule essentiellement les objets suivants :

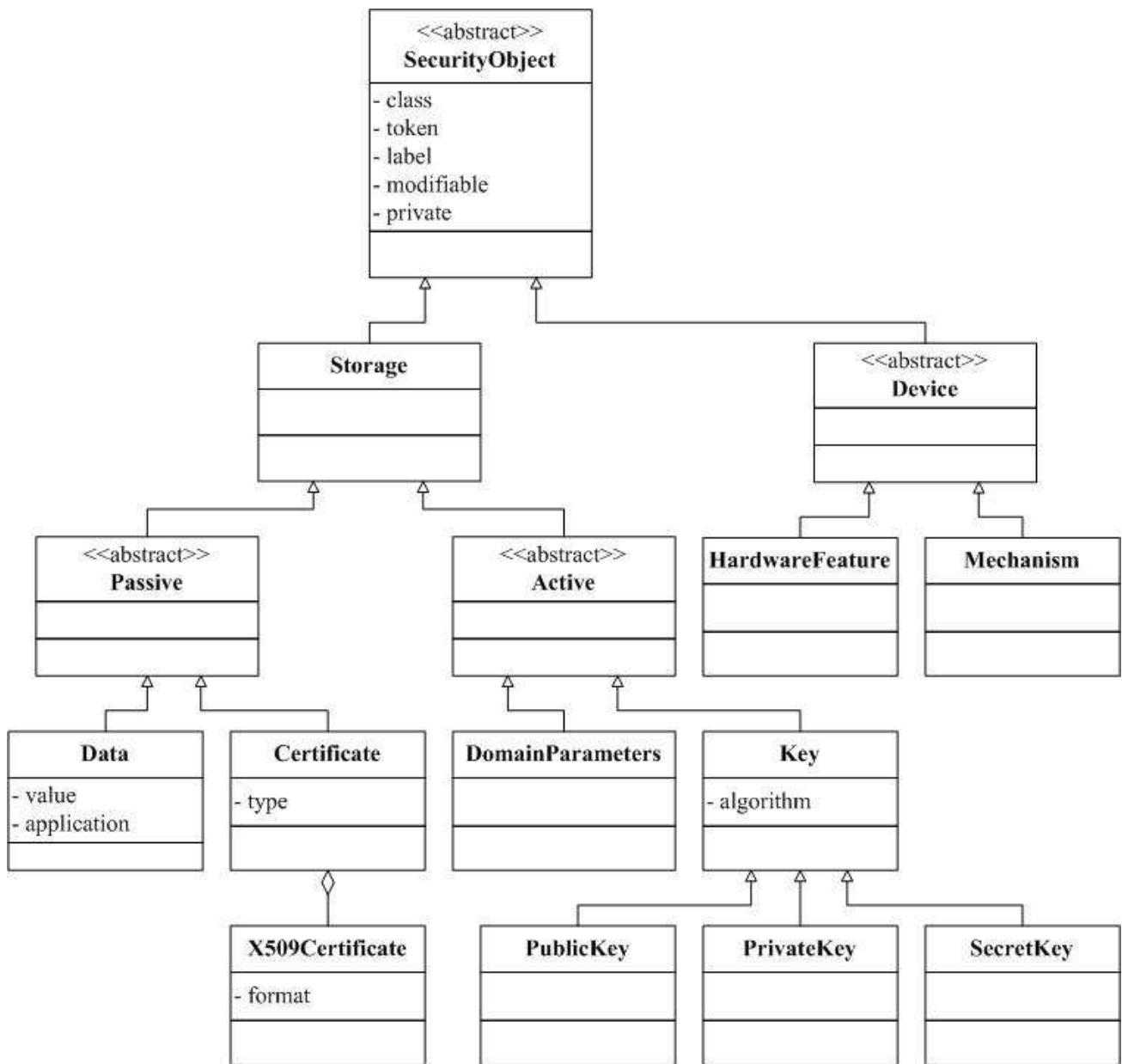


et les applications vont y accéder au travers de l'API :



Les fonctionnalités interfacées par l'API sont, outre les manipulations des différents objets (création, destruction, exportation et importation, configuration d'attributs), les chiffrements et déchiffrements, les digestes, les signatures et leurs vérifications, la génération de clés et leur encryptage/décryptage éventuel pour un transport sûr (wrap/unwrap key) et la génération de nombres aléatoires sécurisés.

De manière abstraite, on peut considérer que les objets manipulés ("objets" au sens large – ce sont des structures pour des langages comme le C) se placent dans une hiérarchie faisant intervenir 3 classes abstraites (Passive, Active et Device) qui ne font pas partie de la spécification mais qui répondent à un besoin de clarification :



Plus concrètement, l'API ressemble à ceci :

function API	description
C_Initialize	initializes Cryptoki
C_GetInfo	obtains general information about Cryptoki
C_GetSlotList	obtains a list of slots in the system
C_GetSlotInfo	obtains information about a particular slot
C_GetTokenInfo	obtains information about a particular token
C_GetMechanismList	obtains a list of mechanisms supported by a token
C_GetMechanismInfo	obtains information about a particular mechanism supported by a token
C_InitToken	initializes a token
C_InitPIN	initializes the normal user's PIN
C_SetPIN	modifies the PIN of the current user

C_OpenSession	opens a connection or “session” between an application and a particular token
C_CloseSession	closes a session
C_CloseAllSessions	closes all sessions with a token
C_GetSessionInfo	obtains information about the session
C_Login	logs into a token
C_Logout	logs out from a token
C_CreateObject	creates an object
C_CopyObject	creates a copy of an object
C_DestroyObject	destroys an object
C_GetObjectSize	obtains the size of an object in bytes
C_GetAttributeValue	obtains an attribute value of an object
C_SetAttributeValue	modifies an attribute value of an object
C_FindObjectsInit	initializes an object search operation
C_FindObjects	continues an object search operation
C_EncryptInit	initializes an encryption operation
C_Encrypt	encrypts single-part data
C_EncryptUpdate	continues a multiple-part encryption operation
C_EncryptFinal	finishes a multiple-part encryption operation
C_DecryptInit	initializes a decryption operation
C_Decrypt	decrypts single-part encrypted data
C_DecryptUpdate	continues a multiple-part decryption operation
C_DecryptFinal	finishes a multiple-part decryption operation
C_DigestInit	initializes a message-digesting operation
C_Digest	digests single-part data
C_DigestUpdate	continues a multiple-part digesting operation
C_DigestFinal	finishes a multiple-part digesting operation
C_SignInit	initializes a signature operation
C_Sign	signs single-part data
C_SignUpdate	continues a multiple-part signature operation
C_SignFinal	finishes a multiple-part signature operation
C_SignRecoverInit	initializes a signature operation, where the data can be recovered from the signature
C_SignRecover	signs single-part data, where the data can be recovered from the signature
C_VerifyInit	initializes a verification operation
C_Verify	verifies a signature on single-part data
C_VerifyUpdate	continues a multiple-part verification operation
C_VerifyFinal	finishes a multiple-part verification operation
C_VerifyRecoverInit	initializes a verification operation where the data is recovered from the signature
C_VerifyRecover	verifies a signature on single-part data, where the data is recovered from the signature
C_GenerateKey	generates a secret key
C_GenerateKeyPair	generates a public-key/private-key pair
C_WrapKey	wraps (encrypts) a key
C_UnwrapKey	unwraps (decrypts) a key
C_DeriveKey	derives a key from a base key

C_SeedRandom	mixes in additional seed material to the random number generator
C_GenerateRandom	generates random data
C_GetFunctionStatus	obtains updated status of a function running in parallel with the application
C_CancelFunction	cancels a function running in parallel with the application
Notify	callback function to process notifications from Cryptoki

Par exemple, pour signer un message, on procède de la manière suivante en termes de l'API :

1. Trouver un token : **C_GetSlotList** ➔ itérer **C_GetTokenInfo**
2. Ouvrir une session et se loguer : **C_OpenSession**, **C_Login**
3. Trouver une clé pour signer : **C_FindObjects**
4. Lancer le mécanisme de signature : **C_SignInit**, **C_Sign**
5. Fermer la session : **C_Logout**, **C_CloseSession**

La spécification définit en outre un certain nombre de symboles (constantes manifestes des langages de programmation) dont nous ferons usage plus loin et qui représentent des propriétés booléennes (les symboles CK_TRUE et CK_FALSE sont bien sûr les valeurs de vérité que l'API utilise pour la valeur de ces symboles) ou non :

Permissions de base sur les objets	
CKA_TOKEN	token ou objet session ?
CKA_MODIFIABLE	attribut read-write ou read-only
CKA_SENSITIVE	attribut sensible non accessible
CKA_PRIVATE	une authentification préalable est requise pour que l'objet soit visible
CKA_TRUSTED	clé : elle peut servir pour un transport sécurisé certificat : peut être reconnu comme de confiance
CKA_LOCAL	objet créé sur le device lui-même (il n'a pas été importé)
Opérations autorisées sur les objets	
CKA_ENCRYPT	l'objet peut être chiffré
CKA_DECRYPT	l'objet peut être déchiffré
CKA_SIGN	l'objet peut être signé
CKA_VERIFY	la signature jointe à l'objet peut être vérifiée
CKA_VERIFY_RECOVER	la signature encapsulant l'objet peut être vérifiée
CKA_DERIVE	on peut créer de nouvelles clés à partir de la clé qui protège l'objet
CKA_ALLOWED_MECHANISMS	liste des mécanismes autorisés à accéder à l'objet
Importations et exportations autorisées sur les objets	
CKA_WRAP	l'objet peut être utilisé pour l'exportation sécurisé d'un autre objet

CKA_WRAP_WITH_TRUSTED	comme le précédent mais seulement pour un objet dont le CKA_TRUSTED est à CK_TRUE
CKA_UNWRAP	l'objet peut être utilisé pour l'importation sécurisé d'un autre objet
CKA_EXTRACTABLE	l'objet peut être extrait et exporté
CKA_WRAP_TEMPLATE	l'exportation n'est autorisée que pour les objets cités
CKA_UNWRAP_TEMPLATE	l'importation n'est autorisée que pour les objets cités
Historique des objets	
CKA_ALWAYS_SENSITIVE	l'attribut est CKA_SENSITIVE est toujours à CK_TRUE
CKA_NEVER_EXTRACTABLE	l'attribut CKA_EXTRACTABLE n'est jamais à CK_TRUE
CKA_START_DATE	début de la date de validité de l'objet
CKA_END_DATE	fin de la date de validité de l'objet

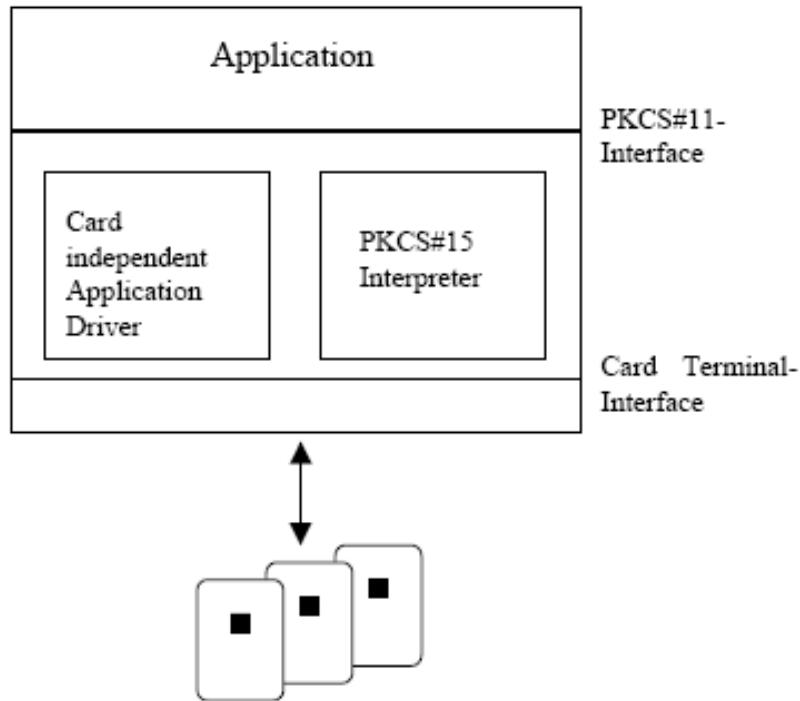
C'est évidemment de ces APIs dont nous avons besoin ici dans le contexte de la signature du courrier électronique, mais pas seulement comme nous le verrons plus loin. Sous Windows, l'API se présente sous la forme d'une dll nommée **beidpkcs11.dll** (sous UNIX, d'une bibliothèque partagée **libbeidpkcs11.so**).

10.3 Le module PKCS#15

Dans le cas particulier où les informations cryptographiques se trouvent dans une carte à puce, on sait qu'il existe une spécification PC/SC visant à permettre l'interopérabilité entre les systèmes à cartes à puce et les systèmes informatiques classiques. Cependant, cette spécification ne donne aucune règle quant au format du contenu de ces cartes, notamment les clés et certificats. La portabilité de ces informations cryptographiques n'est donc pas du tout assurée par PC/SC puisqu'il est supposé que l'utilisateur accèdera aux informations de sa carte en utilisant un service provider, celui-ci étant choisi par le fabricant de la carte..

On peut remarquer que l'Open Card framework n'apporte pas de solution à ce problème (il ne préoccupe tout simplement pas de la question) tandis que la spécification Java Card travaille à un niveau plus abstrait qui occulte les problèmes de formats internes.

C'est ici qu'intervient une autre spécification PKCS: la spécification **PKCS#15**. Elle définit la structure de stockage des clés, certificats, etc qui permettra aux APIs PKCS#11 (par exemple) de les manipuler comme si il s'agissait d'un accès à un système de fichiers. Autrement dit, le module PKCS#15 interface le module PKCS#11 sur les APIs standards d'accès aux ressources d'une carte à puce, typiquement PC/SC :



Les seuls prérequis à l'utilisation des APIs PKCS#15 sont que la carte utilisée soit conforme à l'ISO 7816. De plus, il reste possible de stocker des informations cryptées ou signées extérieurement à la carte. Enfin, le seul type de fichier supposé supporté est celui de la forme d'un bloc contigu de bytes. On dit encore pour résumer que la spécification est "*token neutral*".

L'objet de la spécification est donc de définir un format de fichier. L'idée de base est que tout fichier est obligatoirement décrit par un EF (Elementary File) nommé l'**ODF** (**O**bject **D**irectory **F**ile), qui rassemble en fait des pointeurs vers d'autres DF (Directory File) : PrKDFs, PuKDFs, SKDFs, CDFs, DODFs and AODFs, chacun étant destiné à contenir des objets PKCS#15 particuliers : clés, certificats, PINs, etc. Un répertoire au moins de chaque type doit exister. Plus précisément :

- ◆ les fichiers-répertoires **KDFs** décrivent les clés (Cryptographic Key Directory Files), soit les **PrKDFs**, **SKDFs** and **PuKDFs** (inutile sans doute de définir ces acronymes ...); ces fichiers contiennent diverses informations (attributs) sur les clés se trouvant sur la carte comme leur identifiant, l'algorithme utilisé, la taille de la clé, les restrictions éventuelles d'utilisation et éventuellement des pointeurs vers les clés elles-mêmes.
 - ◆ les fichiers-répertoires **CDFs** décrivent les certificats (Certificate Directory Files) : ces fichiers contiennent diverses informations (attributs) sur les certificats se trouvant sur la carte comme leur identifiant, le type de certificat et des pointeurs vers les certificats eux-mêmes; à remarquer que si le certificat contient une clé publique associée à une clé privée également connue du système de fichiers PKCS#15 considéré, l'identifiant doit être le même.
 - ◆ les fichiers-répertoires **TCDFs** décrivent les certificats à validité vérifiée (Trusted Certificate Directory Files) que l'utilisateur ne peut pas remplacer.
 - ◆ les fichiers-répertoires **AODFs** décrivent les objets d'authentification (Authentication Object Directory Files), essentiellement les codes PINs; ces fichiers contiennent diverses informations (attributs) sur les PINs se trouvant sur la carte comme leur numéro de référence,

leur longueur, les caractères autorisés, leurs caractères de padding et des pointeurs vers les PINs eux-mêmes; à remarquer que les PrKDFs utilisent ces numéros de références pour lier les clés privées au code PIN qui devra être fourni pour leur utilisation.

- ◆ les fichiers-répertoires **DODFs** décrivent les objets qui ne sont ni des clés, ni des certificats, ni des objets d'identification (**Data Object Directory Files**); ces fichiers contiennent diverses informations (attributs) sur les objets en question, comme l'identifiant de l'application à laquelle ils appartiennent, le fait d'être public ou privé, etc et bien sûr des pointeurs sur ces objets.
- ◆ le fichier-répertoire **TIF** (**TokenInfo File**) est obligatoire : il contient les informations sur les tokens du système comme leur numéro de série ou les algorithmes supportés.

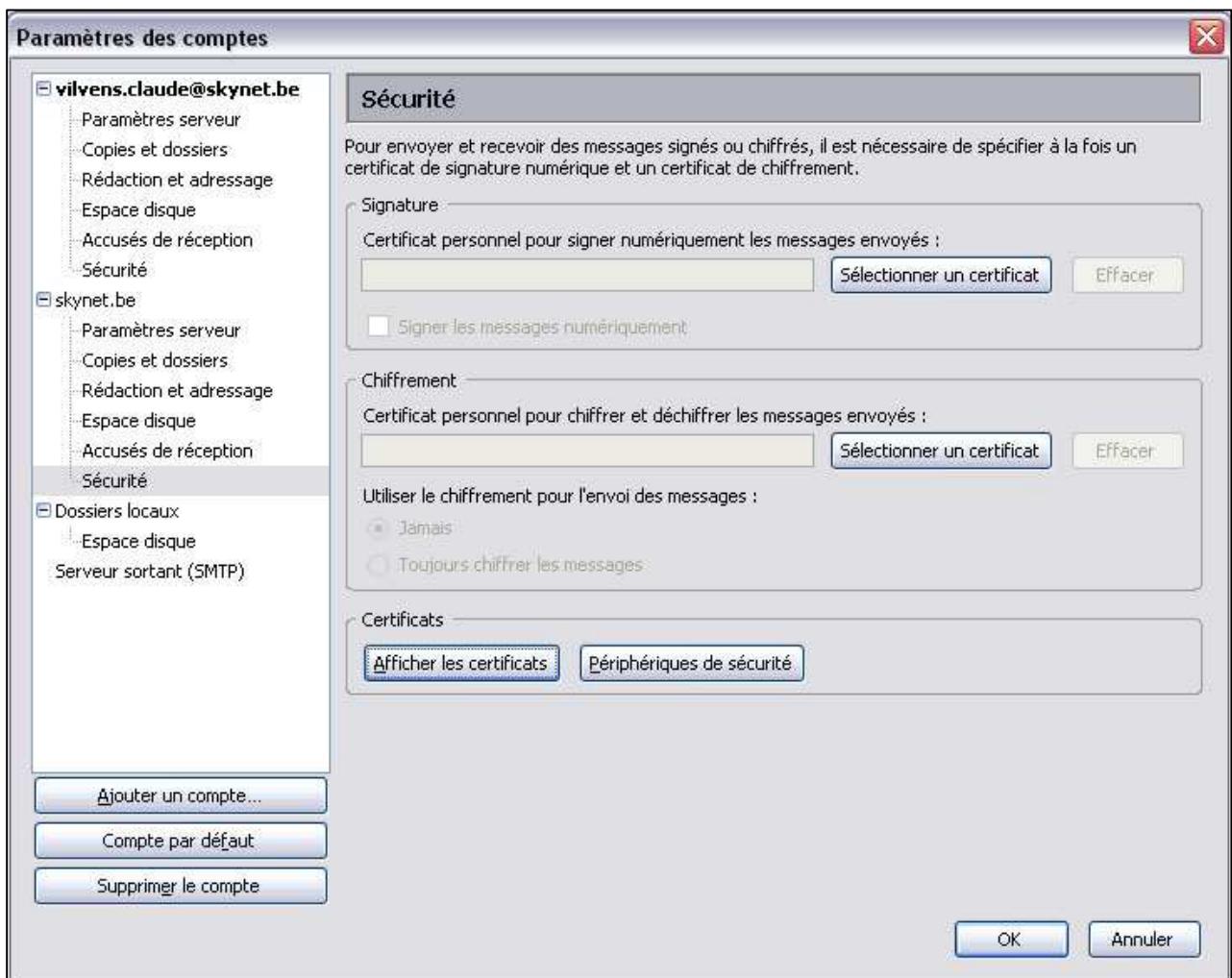
Le décor étant ainsi campé, la procédure de configuration des applications consommant des fonctionnalités cryptographiques peut à présent être bien comprise.

11. L'utilisation des certificats d'identité avec les e-mails (Thunderbird)

11.1 Paramétriser son compte de messagerie

On s'en doute, il faut configurer le compte de messagerie pour qu'il puisse utiliser les informations de la carte d'identité. Plusieurs opérations doivent en fait être réalisées. On peut s'en rendre compte en sélectionnant le compte de messagerie retenu et en visualisant les paramètres du compte :





11.2 L'enregistrement du module PKCS#11

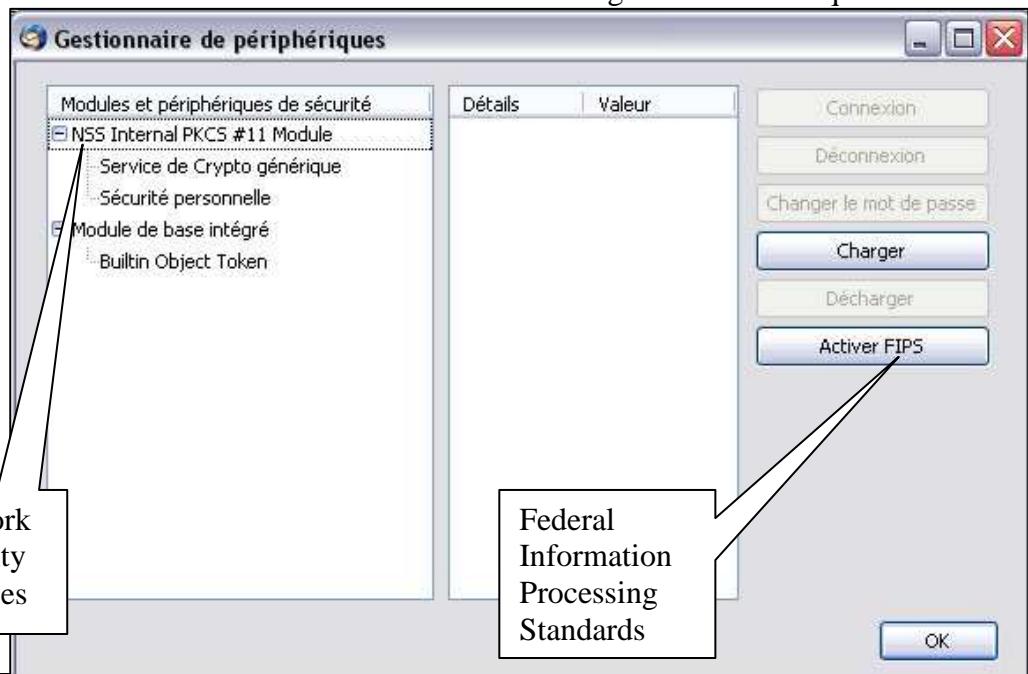
En premier lieu, il faut que la carte d'identité soit reconnue par le logiciel de messagerie comme un "**périphérique de sécurité**", c'est-à-dire un support pouvant contenir les informations nécessaires à la confection et à la vérification d'une signature électronique.

Pour cela, il faut enregistrer un module PKCS#11.

Au préalable, le lecteur de carte d'identité doit être connecté, mais sans contenir de carte. L'appui sur le bouton "Périphériques de sécurité" fait apparaître la boîte de dialogue :

Network
Security
Services

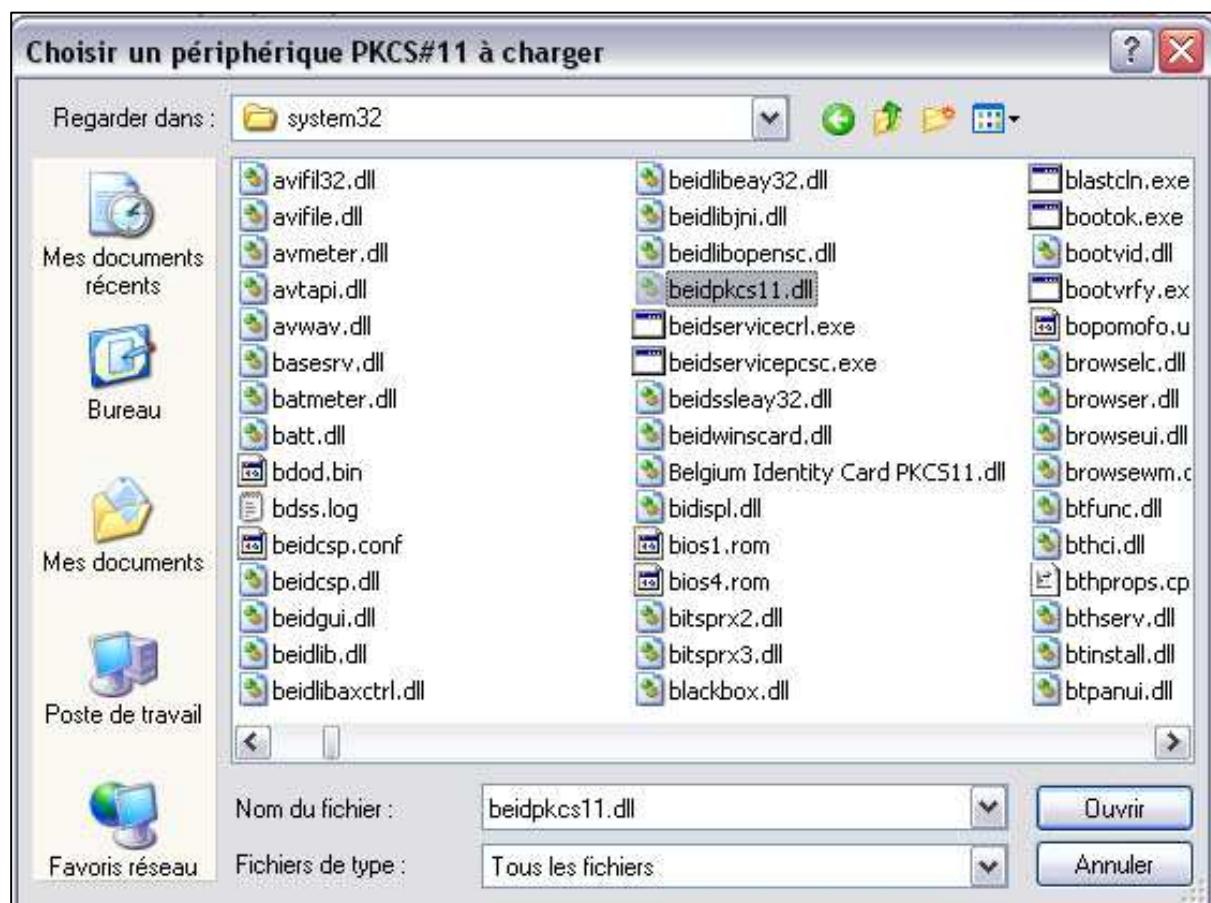
Federal
Information
Processing
Standards



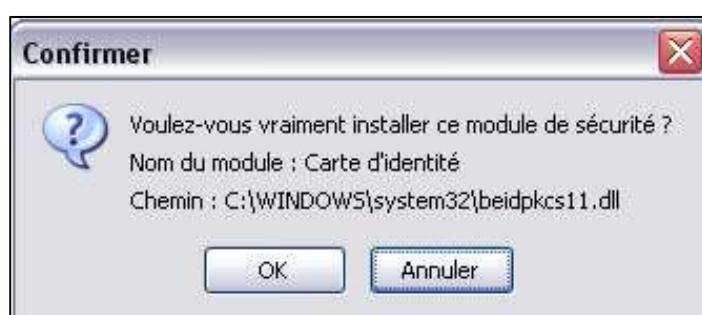
Bien sûr, il suffit de sélectionner le module PKCS#11 et d'appuyer sur le bouton charger. Il sera demandé au préalable de donner un nom au module qui va être chargé (peu importe quoi : par exemple, "Carte d'identité") :



puis, après appui sur le bouton "Parcourir", de sélectionner la dll nécessaire :



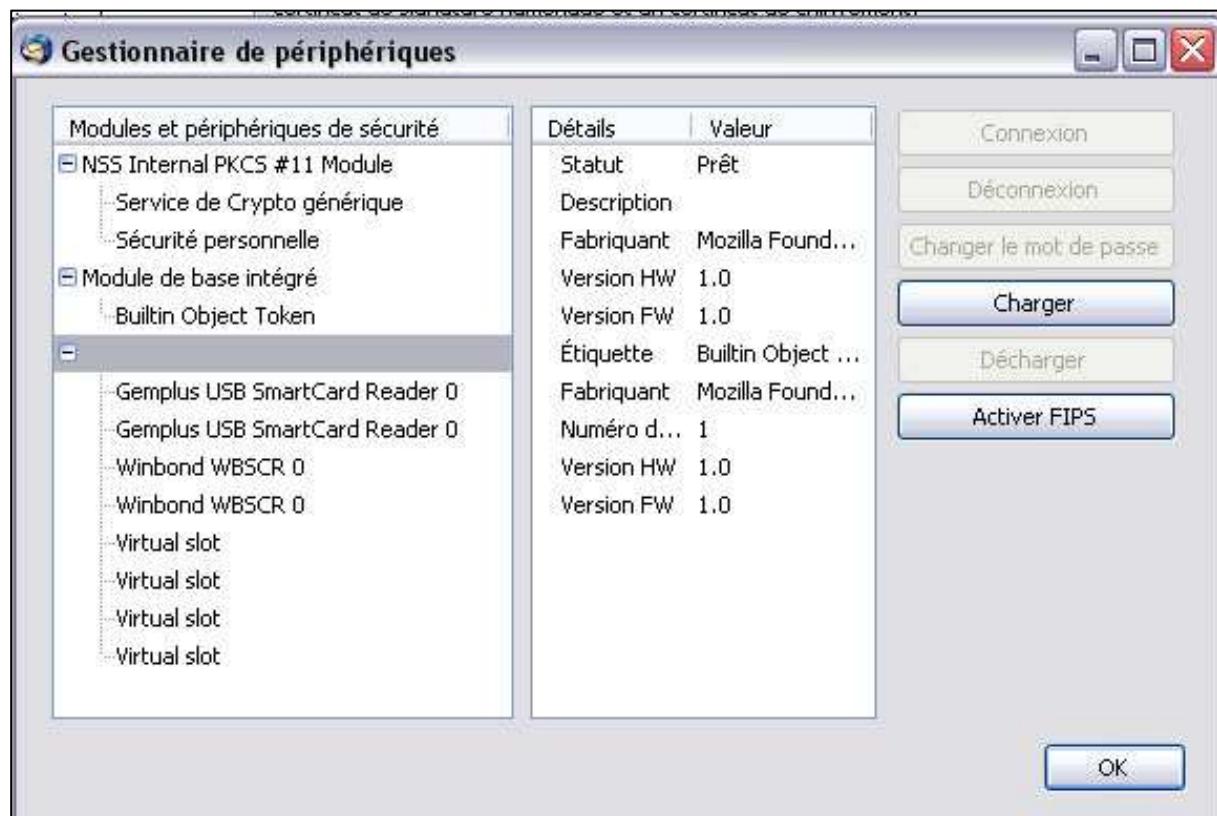
Il ne reste plus qu'à confirmer :



Si tout se passe bien :

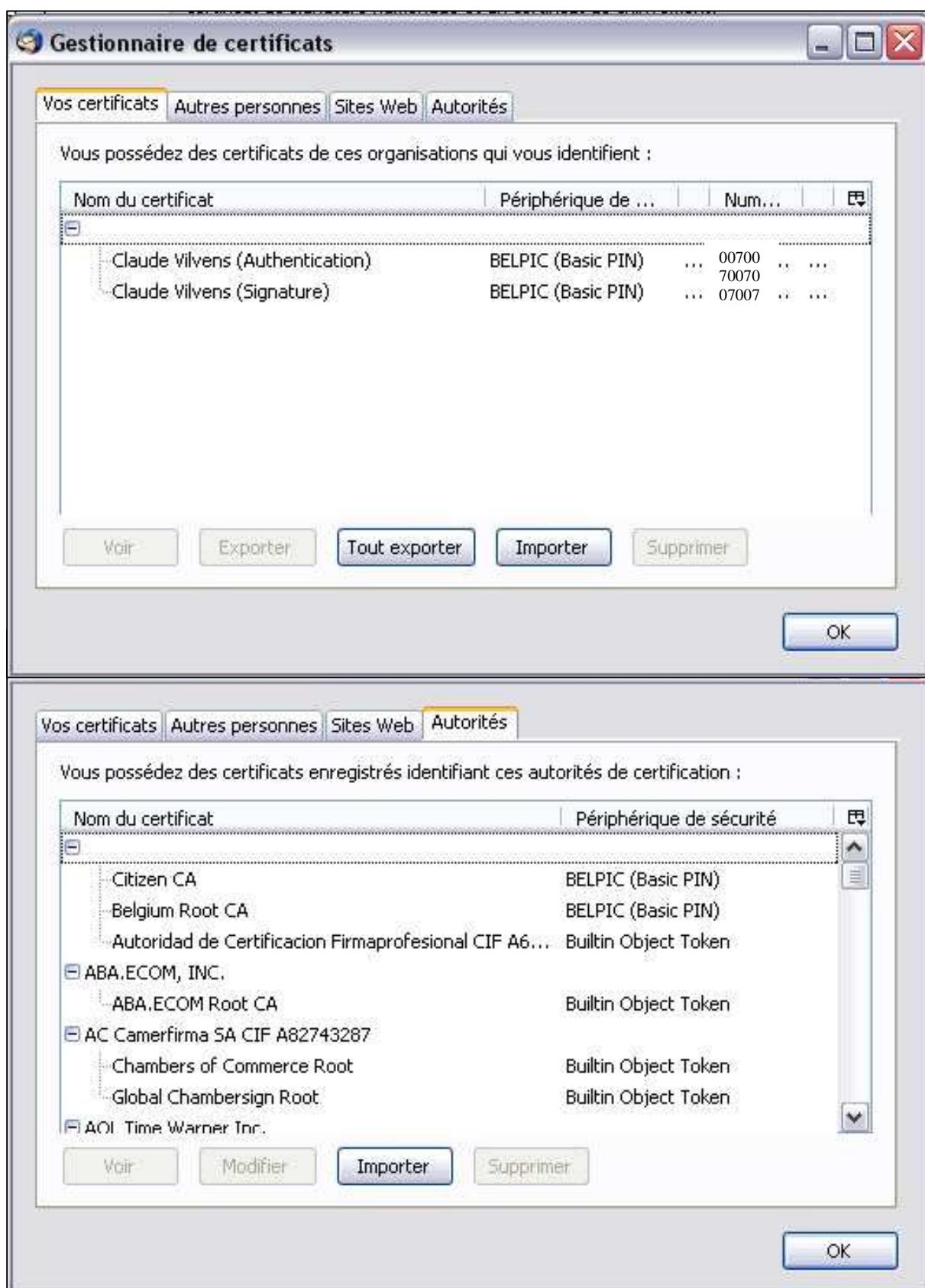


On peut vérifier ensuite que le lecteur de carte (ici, un Gemplus USB) a été ajouté à la liste :



11.3 L'enregistrement des certificats

Une fois la carte d'identité insérée dans le lecteur, on peut demander au logiciel Thunderbird de copier les certificats qu'il trouve sur la carte dans son Gestionnaire de certificats. Pour cela, à partir de "Paramètres des comptes" – "Sécurité", on peut appuyer sur le bouton "Afficher les certificats" : les onglets "Vos certificats" et "Autorités" font bien apparaître les 4 certificats bien connus d'une carte d'identité :



Pour indiquer que l'autorité Belgium Root CA est valable pour attester une signature de courrier électronique, il suffit de sélectionner le certificat correspondant et d'appuyer sur le bouton "Modifier" :



11.4 La configuration du mécanisme de signature

Maintenant que tous les certificats nécessaires sont engrangés, il nous reste à configurer le logiciel pour qu'il puisse les utiliser. Pour cela, toujours à partir de "Paramètres des comptes" – "Sécurité", on va appuyer sur le bouton "Sélectionner un certificat" puis sélectionner le certificat de signature à partir de la carte d'identité :

11.5 La signature d'un message

Lors de la rédaction d'une message, on peut choisir de signer celui-ci :



On peut alors remarquer dans la ligne de statut une icône en forme de stylo pour indiquer que le message va être signé. Lors de l'envoi du premier message signé, il est demandé d'introduire le code PIN pour activer permettre l'accès à la carte d'identité :



11.6 La réception d'un message signé

Lorsqu'un message signé parvient à un logiciel de messagerie, celui-ci va tenter de vérifier la signature.

1) En cas de succès, le symbole du style (avec point d'interrogation) apparaîtra à droite dans le titre :

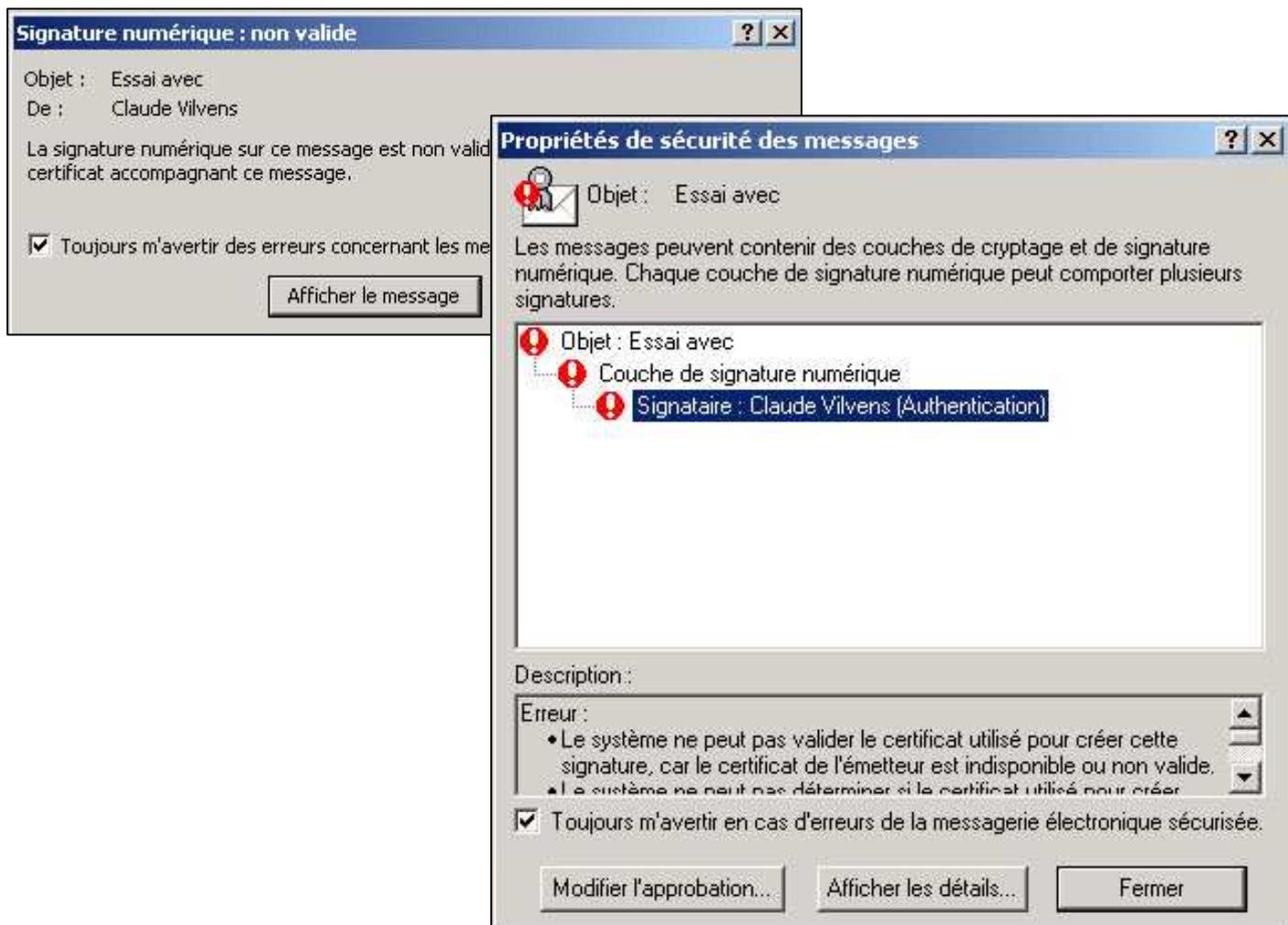


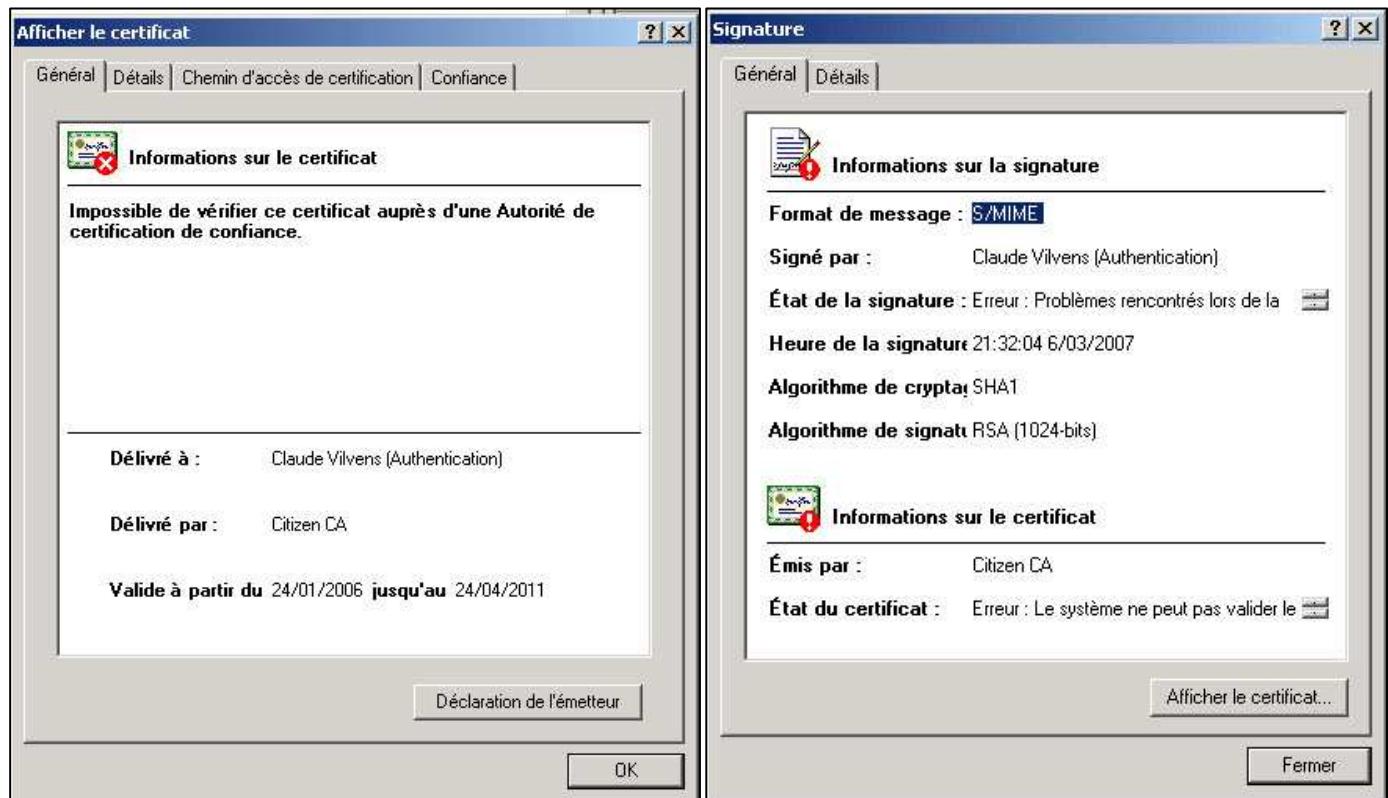
Le point d'interrogation signifie simplement qu'une information supplémentaire est disponible, information que l'on peut obtenir en cliquant sur l'icône :



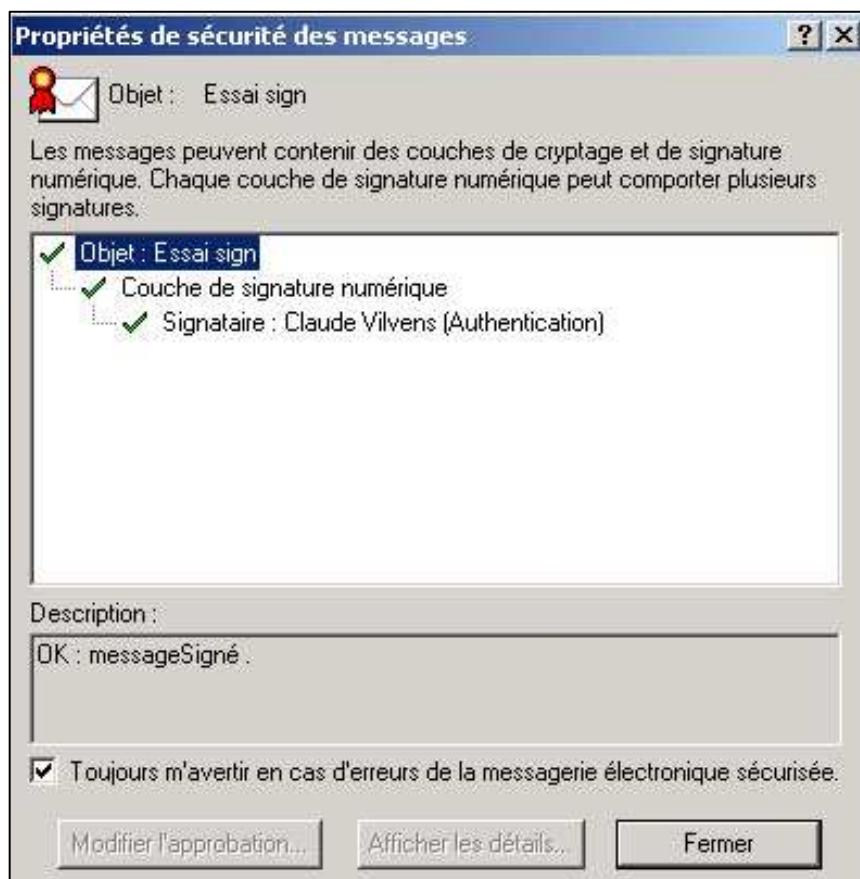
On remarquera encore les réserves exprimées : la signature est valide, mais le logiciel ne sait pas vérifier la corrélation entre le signature et l'adresse e-mail, ce qui est normal puisque ce renseignement ne figure pas dans le certificat.

2) Supposons à présent que le message soit réceptionné par un utilisateur utilisant Outlook. Il obtiendra à la réception, si il ne possède pas les certificats nécessaires :





Par contre, si les certificats ont été installés sur la machine réceptrice :

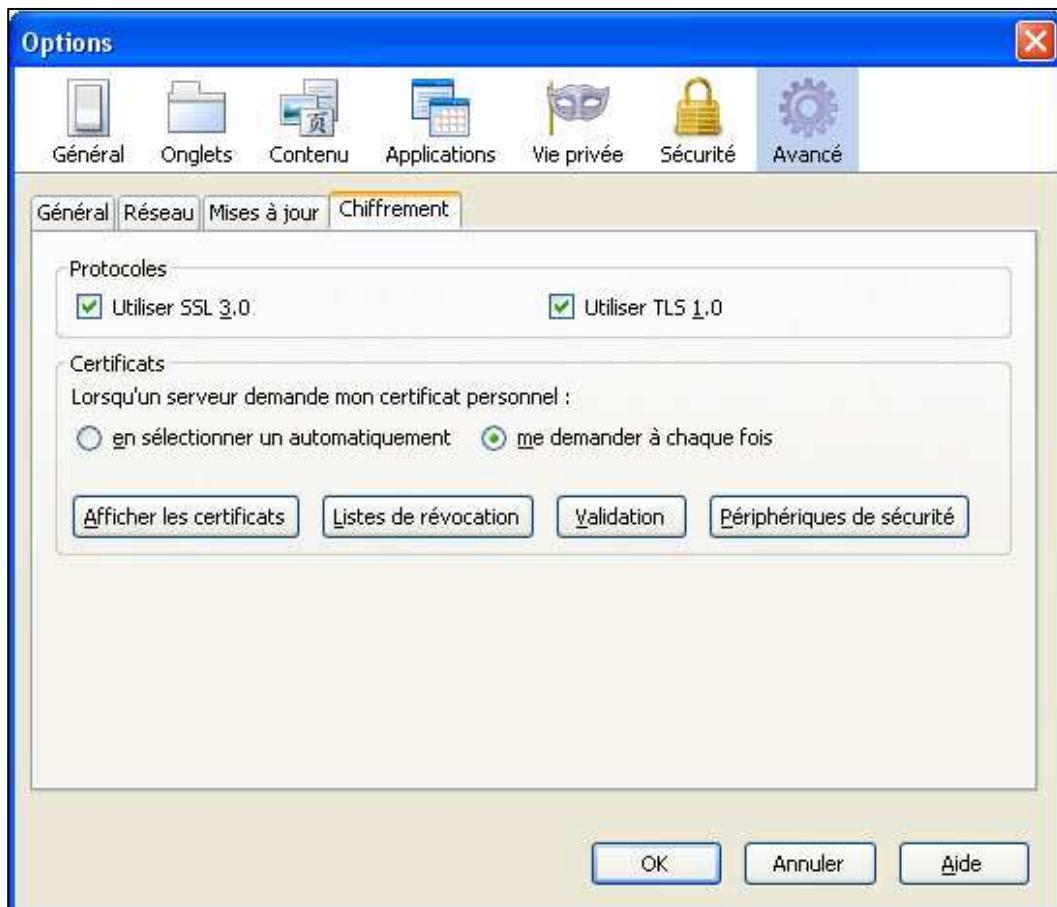


12. L'utilisation des certificats d'identité avec un browser (Firefox)

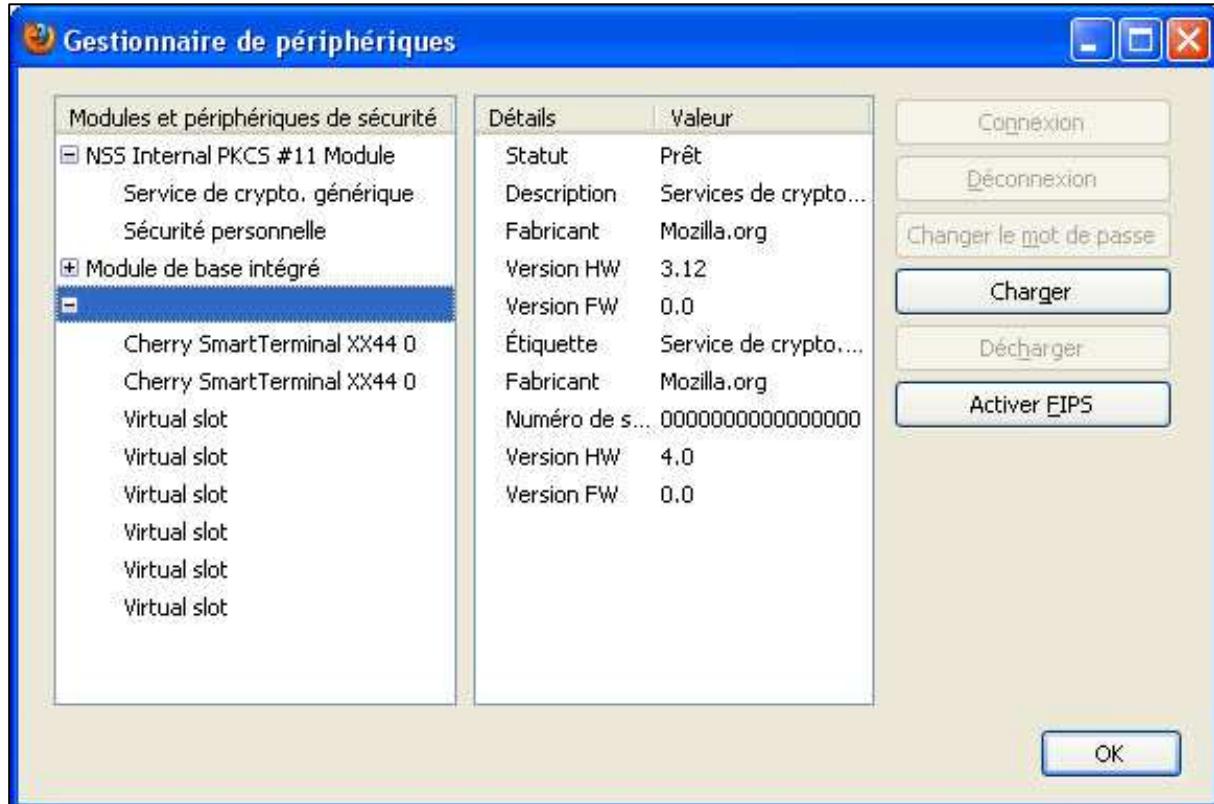
Certains sites Web réclament de la part du client un certificat. Comme la carte d'identité en contient, nous allons pouvoir les utiliser à cet effet.

12.1 L'enregistrement du module PKCS#11

En premier lieu, comme pour le logiciel de messagerie, il faut que la carte d'identité soit reconnue par le browser comme un "**périphérique de sécurité**", c'est-à-dire un support pouvant contenir les informations nécessaires à la manipulation de certificats et de signatures électroniques. Pour cela, il faudra donc enregistrer le **module PKCS#11** (pour rappel, un interface standard d'accès aux données cryptographiques mémorisées dans un dispositif hardware). On peut procéder comme pour Thunderbird, à partir de Outils → Options puis onglet Avancé :



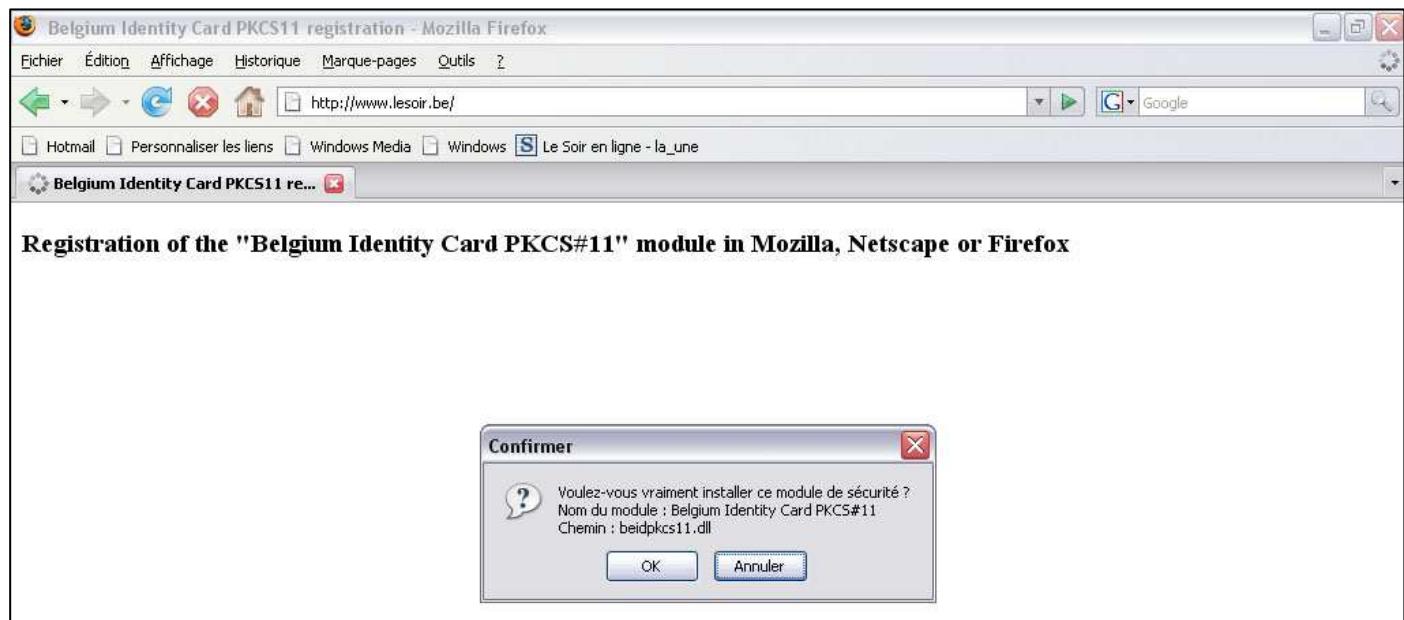
On procède alors de la même manière après appui sur le bouton "Périphériques de sécurité" et on arrive finalement à constater que le lecteur de carte (ici, un SmartTerminal 1044 USB) a été ajouté à la liste :

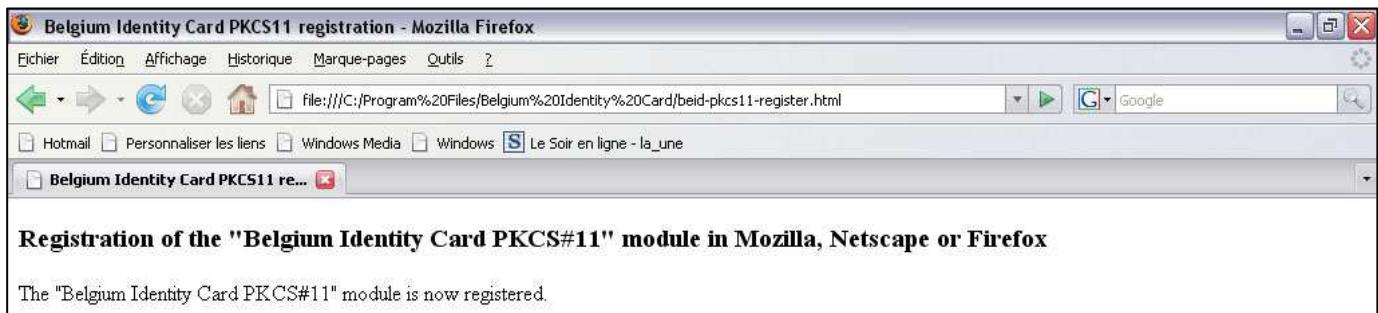


Une autre manière de procéder est le suivant. Au préalable, le lecteur de carte d'identité doit être connecté, mais sans contenir de carte. L'enregistrement proprement dit est alors simple : on ouvre dans le browser le fichier

C:\Program Files\Belgium Identity Card\beid-pkcs11-register.html

(sous UNIX, /usr/local/share/beid/beid-pkcs11-register.html). On obtient :

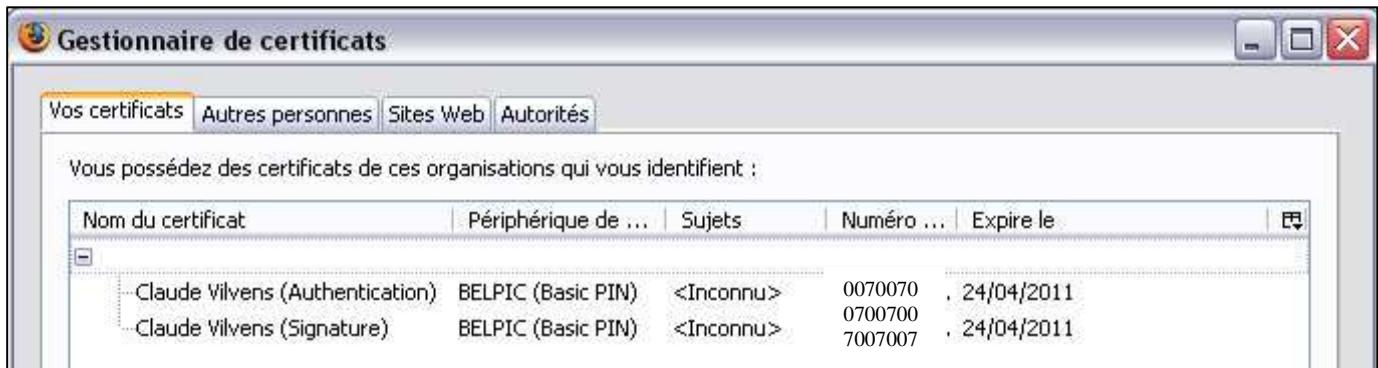




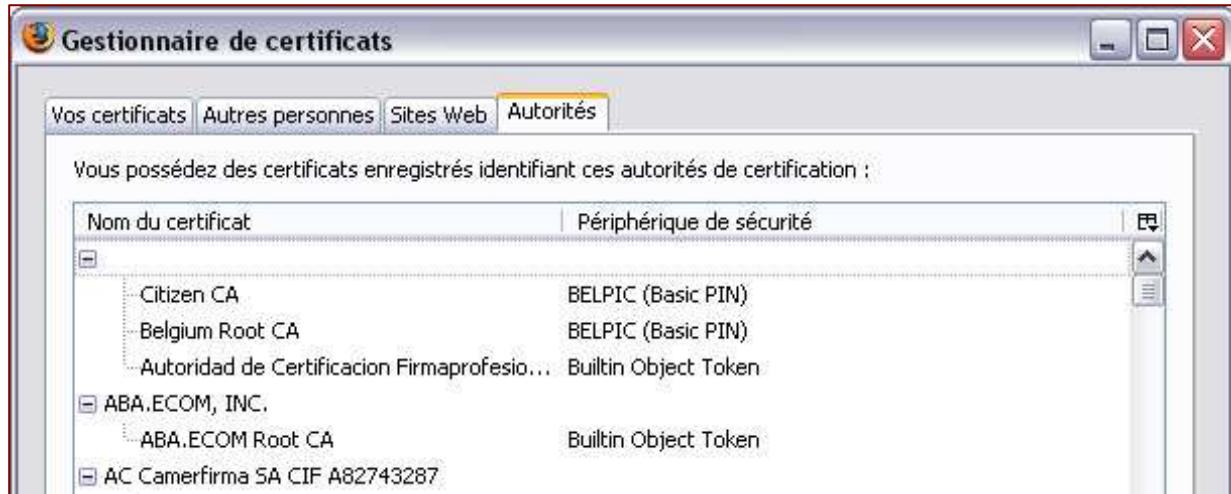
Attention : cet enregistrement se fait avec du code JavaScript et il conviendra donc de vérifier au préalable que celui-ci est activé (dans le browser avec l'onglet Contenu obtenu à partir du menu par Outils → Options).

12.2 L'enregistrement des certificats

Une fois la carte d'identité insérée dans le lecteur, on peut demander au browser (ici, Firefox) de copier les certificats qu'il trouve sur la carte dans son Gestionnaire de certificats. Pour cela, le choix Outils→Options puis l'onglet Avancé mais cette fois pour appuyer sur le bouton "Afficher les certificats" : le browser va aller lire les 4 certificats sur la carte d'identité et les ranger dans deux catégories : "Vos certificats" et "Autorités". Plus précisément, l'onglet des certificats personnels montre tout d'abord :



- en fait, seul le premier certificat sera utile pour l'accès à des sites Web protégés, car le second est voué à la confection d'une signature. L'onglet "Autorités" quant à lui montre les certificats du niveau "Autorité" :



Evidemment, pour la chaîne de certificats fonctionne, il nous faut indiquer que le "Belgium Root CA" est fiable.

13. L'API général de la carte d'identité belge

13.1 La librairie de développement

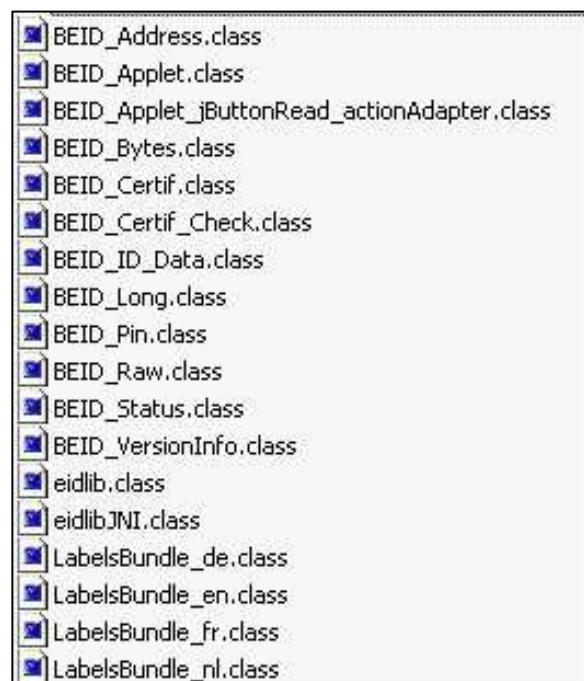
Outre le middleware, il est possible d'obtenir les APIs permettant d'utiliser les fonctionnalités de la carte d'identité : il s'agit du BElgian eID toolkit (**BEID**). L'API est tout d'abord défini sans référence à un langage précis, en terme de fonctions décrites de manière formelle. Cette description sera ensuite déclinée dans différents langages de développements actuels, comme C, VBA, ActiveX, Perl et Java.

La librairie proprement dite a été écrite en C et il va donc falloir faire appel au mécanisme du **Java Native Interface (JNI)** qui, pour rappel, permet aux applications Java d'utiliser du code natif qui est le résultat de compilations et éditions de liens de code source écrit en C, C++ ou Assembleur (voir volume II, chapitre XV, paragraphe 6). Il nous faudra donc prévoir un bloc statique pour charger la librairie qui se nomme *beidlibjni.dll* et qui se trouve dans le répertoire C:\windows\system32 :

CIIinfosIdentite.java (chargement de la librairie)

```
...
static
{
    try
    {
        System.loadLibrary("beidlibjni");
    }
    catch (UnsatisfiedLinkError e)
    {
        System.err.println("Impossible de charger la librairie native" + e); System.exit(1);
    }
}
...
...
```

Bien entendu, il s'agit ici du chargement dynamique d'une librairie, qui sera trouvée sans problème car windows\system32 se trouve habituellement dans le PATH. Mais le compilateur Java doit être capable de vérifier l'utilisation des fonctions de la librairie (qu'il va voir comme des méthodes statiques) : il devra trouver les prototypes dans un jar. Deux jars, d'ailleurs signés, nommés **beidlib.jar** et **eidlib.jar** (situés dans le répertoire C:\Program Files\Belgium Identity Card) sont fournis lors de l'installation du framework : ils contiennent la déclinaison Java de l'API :



Il s'agit bien d'un mécanisme de JNI comme on peut le voir dans la classe de base :

eidlbinJNI.java

```
// Source File Name: eidlibJNI.java

package be.belgium.eid;

class eidlibJNI
{
    eidlibJNI() { }

    public static final native int get_BEID_Status_general(long l);
    public static final native int get_BEID_Status_system(long l);
    public static final native int get_BEID_Status_pcsc(long l);
    public static final native byte[] get_BEID_Status_cardSW(long l);
    public static final native long new_BEID_Status();
    public static final native void delete_BEID_Status(long l);

    public static final native byte[] get_BEID_Certif_certif(long l);
    public static final native String get_BEID_Certif_certifLabel(long l);
    public static final native int get_BEID_Certif_certifStatus(long l);
    public static final native long new_BEID_Certif();
    public static final native void delete_BEID_Certif(long l);
    public static final native int get_BEID_Certif_Check_usedPolicy(long l);
    public static final native long get_BEID_Certif_Check_certificate(long l, int i);
    public static final native int get_BEID_Certif_Check_certificatesLength(long l);
    public static final native int get_BEID_Certif_Check_signatureCheck(long l);
    public static final native long new_BEID_Certif_Check();
    public static final native void delete_BEID_Certif_Check(long l);

    ...
    public static final native short get_BEID_ID_Data_version(long l);
    public static final native String get_BEID_ID_Data_cardNumber(long l);
    public static final native String get_BEID_ID_Data_chipNumber(long l);
    public static final native String get_BEID_ID_Data_validityDateBegin(long l);
    public static final native String get_BEID_ID_Data_validityDateEnd(long l);
    public static final native String get_BEID_ID_Data_municipality(long l);
    public static final native String get_BEID_ID_Data_nationalNumber(long l);
    public static final native String get_BEID_ID_Data_name(long l);
    public static final native String get_BEID_ID_Data(firstName1(long l));

    ...
    public static final native byte[] get_BEID_Bytes_data(long l);
    public static final native long new_BEID_Bytes();

    ...
    public static final native long BEID_VerifyPIN(long l, String s, long l1);
    public static final native long BEID_ChangePIN(long l, String s, String s1, long l1);
    public static final native long BEID_GetPINStatus(long l, long l1, int i, long l2);

    ...
    public static final native long BEID_ReadFile(byte abyte0[], long l, long l1);
    public static final native long BEID_WriteFile(byte abyte0[], byte abyte1[], long l);
    public static final native long BEID_SendAPDU(byte abyte0[], long l, long l1);
}
}
```

On trouve dans cet API quatre groupes de fonctions :

- ◆ les fonctions d'initialisation et de terminaison, dont le rôle est en fait de faire démarrer le framework ou de l'arrêter;
- ◆ les fonctions d'identité, de résidence, de photographie, qui sont évidemment celles qui nous intéressent directement;
- ◆ des fonctions "haut-niveau" permettant l'accès à toute information stockée sur la carte (le code PIN, des fichiers, etc);
- ◆ des fonctions "bas-niveau", d'un intérêt limité pour le développeur d'application.

Voyons à présent les méthodes (fonctions) les plus intéressantes.

13.2 Les fonctions générales d'initialisation et de terminaison

Tout commence toujours, obligatoirement, avec l'appel de la fonction qui initialise le framework :

BEID_Init (<nom PC/SC du lecteur – ou NULL pour une détection automatique>,
<indicateur d'utilisation de l'OCSP>,
<indicateur d'utilisation d'un CRL>,
<adresse d'accès au lecteur – peut être ignoré au haut niveau>)

Pour rappel, **PC/SC** (pour Personal Computer/Smart Card) est une librairie gérant l'accès d'un PC aux cartes à puce sous Windows par l'intermédiaire d'un lecteur de cartes. Les spécifications de cette librairie sont établies par le PC/SC Workgroup, composé de grands fabricants de cartes à puce ou d'ordinateurs, dans un but de standardisation des commandes.

Les deux flags peuvent prendre les valeurs 0 (pas utilisé), 1 (optionnel) ou 2 (obligatoire). En cas d'utilisation simultanée des deux politiques OCSP et CRL, c'est l'OCSP qui est tentée la première. Bien sûr, l'application devra se terminer par l'appel de la fonction

BEID_Exit()

qui libère les ressources prises par BEID_Init().

13.3 Les fonctions d'initialisation en Java

A ces définitions formelles correspondent les méthodes Java statiques de la classe **eidlib** du package **be.belgium.eid** :

- ◆ **public static BEID_Status BEID_Init (String ReaderName, int OCSP, int CRL, BEID_Long CardHandle)**
- ◆ **public static BEID_Status BEID_Exit()**

Comme toutes leurs consœurs, ces fonctions retournent une information sur le succès ou l'échec de l'opération - il s'agit d'une structure définie formellement par :

BEID_Status

```
{
    <code general>
    <errno>
    <erreur PC/SC>
    <les deux status word APDU>
}
```

La valeur 0 pour le premier champ signifie toujours un succès – quant aux autres :

Value	C constant	Explanation
0	BEID_OK	Function succeeded
1	BEID_E_SYSTEM	Unknown system error (see system error code)
2	BEID_E_PCSC	Unknown PC/SC error (see PC/SC error code)
3	BEID_E_CARD	Unknown card error (see card status word)
4	BEID_E_BAD_PARAM	Invalid parameter (NULL pointer, out of bound, etc.)
5	BEID_E_INTERNAL	An internal consistency check failed
6	BEID_E_INVALID_HANDLE	The supplied handle was invalid
7	BEID_E_INSUFFICIENT_BUFFER	The data buffer to receive returned data is too small for the returned data
8	BEID_E_COMM_ERROR	An internal communications error has been detected
9	BEID_E_TIMEOUT	A specified timeout value has expired
10	BEID_E_UNKNOWN_CARD	The smart card is not recognized
11	BEID_E_KEYPAD_CANCELLED	Input on pinpad cancelled
12	BEID_E_KEYPAD_TIMEOUT	Timeout returned from pinpad
13	BEID_E_KEYPAD_PIN_MISMATCH	The two PINs did not match
14	BEID_E_KEYPAD_MSG_TOO_LONG	Message too long on pinpad
15	BEID_E_INVALID_PIN_LENGTH	Invalid PIN length
16	BEID_E_VERIFICATION	Error in a signature verification (see 2.13)
17	BEID_E_NOT_INITIALIZED	Toolkit not initialized
18	BEID_E_UNKNOWN	An internal error has been detected, but the source is unknown
19	BEID_E_UNSUPPORTED_FUNCTION	Function is not supported
20	BEID_E_INCORRECT_VERSION	The Toolkit version is incompatible with the calling interface. The program needs to be re-compiled.
21	BEID_E_INVALID_ROOT_CERT	Wrong Root certificate
22	BEID_E_VALIDATION	Error certificate validation (OSCPICRL) (see 2.13)

En Java, cela nous donne la classe **BEID_Status** avec comme méthodes utiles :

- ◆ public int getGeneral()
- ◆ public int getSystem()
- ◆ public int getPcsc()
- ◆ public byte[] getCardSW()

On peut donc imaginer comme point de départ d'une application lisant une carte d'identité électronique quelque chose du genre (la classe BEID_Long n'a que le grand mérite de gérer un entier long – méthode getLong()) :

CIIinfosIdentite.java (initialisation du framewrok)

```
...
BEID_Status oStatus;
BEID_Long CardHandle = new BEID_Long();

oStatus = eidlib.BEID_Init(null, 0, 0, CardHandle);
...
```

14. La fonction d'identité

L'acquisition des données non cryptographiques se fait très simplement au moyen de fonctions assez similaires. L'entrée du code PIN n'est pas nécessaire et l'appel peut se faire même si un autre DF est sélectionné. Nous illustrerons ici la fonction **BEID_getId()** qui permet de récupérer toutes les données liées, de près ou de loin, à l'identité du propriétaire de la carte – donc, elle ne permet pas de récupérer les données concernant le domicile ou la photo (ce sont les fonctions **BEID_GetAddress()** et **BEID_GetPicture()** qui s'en chargent, selon un mécanisme analogue). Cette fonction a pour syntaxe :

BEID_GetID (<adresse d'une structure BEID_ID_Data à remplir avec les renseignements d'identité>,
 <adresse d'une structure BEID_Certif_Check à remplir avec les informations concernant les certificats>)

1) La première structure **BEID_ID_Data** contient évidemment tous les champs prévisibles dans ce contexte, comme les nom et prénoms, le lieu et la date de naissance, etc (la liste exhaustive sera donnée avec la liste des méthodes Java).

2) La deuxième structure **BEID_Certif_Check** est un peu plus délicate puisqu'elle correspond à la vérification de la signature des données – nous allons y revenir ...

En Java, ces deux structures se déclinent en deux classes munies d'un constructeur par défaut et l'accès aux informations d'identité se fait au moyen de la méthode statique de la classe **eidlib** :

```
public static BEID_Status BEID_GetID (BEID_ID_Data IDData, BEID_Certif_Check CertifCheck)
```

Elle peut donc s'utiliser ainsi :

CIIinfosIdentite.java (acquisition des données d'identité)

```
...
BEID_ID_Data IDData = new BEID_ID_Data();
BEID_Certif_Check CertCheck = new BEID_Certif_Check();

oStatus = eidlib.BEID_GetID(IDData, CertCheck);
...
```

L'objet BEID_ID_Data peut nous fournir tous les renseignements nécessaires avec les méthodes suivantes :

information	méthode
version	public short getVersion()
numéro de carte	public String getCardNumber()
numéro de microprocesseur	public String getChipNumber()
date de validité : début et fin	public String getValidityDateBegin(), public String getValidityDateEnd()
commune d'émission	public String getMunicipality()
numéro national	public String getNationalNumber()
nom, prénoms	public String getName(), public String getFirstName1(), public String getFirstName2(), public String getFirstName3()
nationalité	public String getNationality()
lieu et date de naissance	public String getBirthLocation() public String getBirthDate()
sexé	public String getSex()
titre de noblesse éventuel	public String getNobleCondition()
type de document	public int getDocumentType()
statuts spéciaux éventuels (canne blanche, canne jaune, minorité)	public boolean getWhiteCane(), public boolean getYellowCane(), public boolean getExtendedMinority()

On peut donc imaginer la première application simple suivante :

CIIinfosIdentite.java

```
/*
 * CIIinfosIdentite.java
 * Created on 12 septembre 2007, 16:19
 */
```

```
package carteidentite;
```

```
/**
 * @author Vilvens
 */
```

```
import be.belgium.eid.*;
```

```

public class CIInfosIdentite
{
    static
    {
        try
        {
            System.loadLibrary("beidlibjni");
        }
        catch (UnsatisfiedLinkError e)
        {
            System.err.println("Native code library eidlib failed to load.\n" + e);
            System.exit(1);
        }
    }

    public CIInfosIdentite() { }

    public static void main(String[] args)
    {
        BEID_Status oStatus;
        BEID_Long CardHandle = new BEID_Long();

        oStatus = eidlib.BEID_Init(null, 0, 0, CardHandle);
        System.out.println("BEID_Init : Code de statut = " + oStatus.getGeneral());

        BEID_ID_Data IDData = new BEID_ID_Data();
        BEID_Certif_Check CertCheck = new BEID_Certif_Check();

        oStatus = eidlib.BEID_GetID(IDData, CertCheck);
        System.out.println("BEID_GetID : Code de statut = " + oStatus.getGeneral());

        System.out.println("Carte valide du " + IDData.getValidityDateBegin() + " au " +
                           IDData.getValidityDateEnd());
        System.out.println("Commune qui a délivré la carte : " + IDData.getMunicipality());
        System.out.println("Nom : " + IDData.getName());
        System.out.println("Prénom : " + IDData.getFirstName1());
        System.out.println("Lieu de naissance : " + IDData.getBirthLocation());
        System.out.println("Date de naissance : " + IDData.getBirthDate());
        System.out.println("Sexe : " + IDData.getSex());
    }
}

```

Ce qui donne comme résultat (pour la petite histoire, vu que tout le monde s'en fiche ;-)) :



BEID_Init : Code de statut = 0
BEID_GetID : Code de statut = 0
Carte valide du 20060111 au 30110111
Commune qui a délivré la carte : Oupeye
Nom : Vilvens
Prénom : Claude Marie
Lieu de naissance : Ougrée
Date de naissance : 19680123
Sexe : M

Bien sûr, si la carte d'identité n'est pas dans le lecteur :

BEID_Init : Code de statut = 2
BEID_GetID : Code de statut = 17

Mais il nous faut tout de même mieux appréhender les mécanismes de sécurité ...

15. La sécurité

15.1 Les principes généraux

En fait, toute fonction chargée de fournir des données signées va, avant de s'exécuter, vérifier la signature et l'intégrité de la chaîne de certificats associés. On obtiendra donc dans la structure **BEID_Certif_Check** le résultat de ces vérifications et les données ne seront délivrées qu'en cas de succès. Cette structure BEID_Certif_Check comporte les informations suivantes :

1) le code de la stratégie utilisée pour la vérification des certificats; la valeur de ces codes peut être :

Value	C constant	Explanation
0	BEID_POLICY_NONE	No policy used
1	BEID_POLICY_OCSP	OCSP policy used
2	BEID_POLICY_CRL	CRL policy used
3	BEID_POLICY_BOTH	OCSP and CRL policy used

Essentiellement, on peut donc vérifier par consultation du statut du certificat chez l'autorité de référence (CSP) ou par consultation d'une liste de révocation.

2) le résultat de la vérification de la signature (il en sera également ainsi pour l'accès aux données du domicile mais ce sera plutôt un digest qui sera vérifié dans le cas de l'accès à la photo); les codes de résultats sont :

Value	C constant	Explanation
-1	BEID_SIGNATURE_PROCESSING_ERROR	Error verifying the signature.
0	BEID_SIGNATURE_VALID	The signature is valid.
1	BEID_SIGNATURE_INVALID	The signature is not valid.
2	BEID_SIGNATURE_VALID_WRONG_RRN_CERT	The signature is valid but wrong RRN certificate.
3	BEID_SIGNATURE_INVALID_WRONG_RRN_CERT	The signature is not valid and wrong RRN certificate.

3) une liste des certificats (un tableau en C) avec un champ indiquant leur nombre; un certificat est mémorisé sous la forme d'une structure BEID_Certif qui comporte :

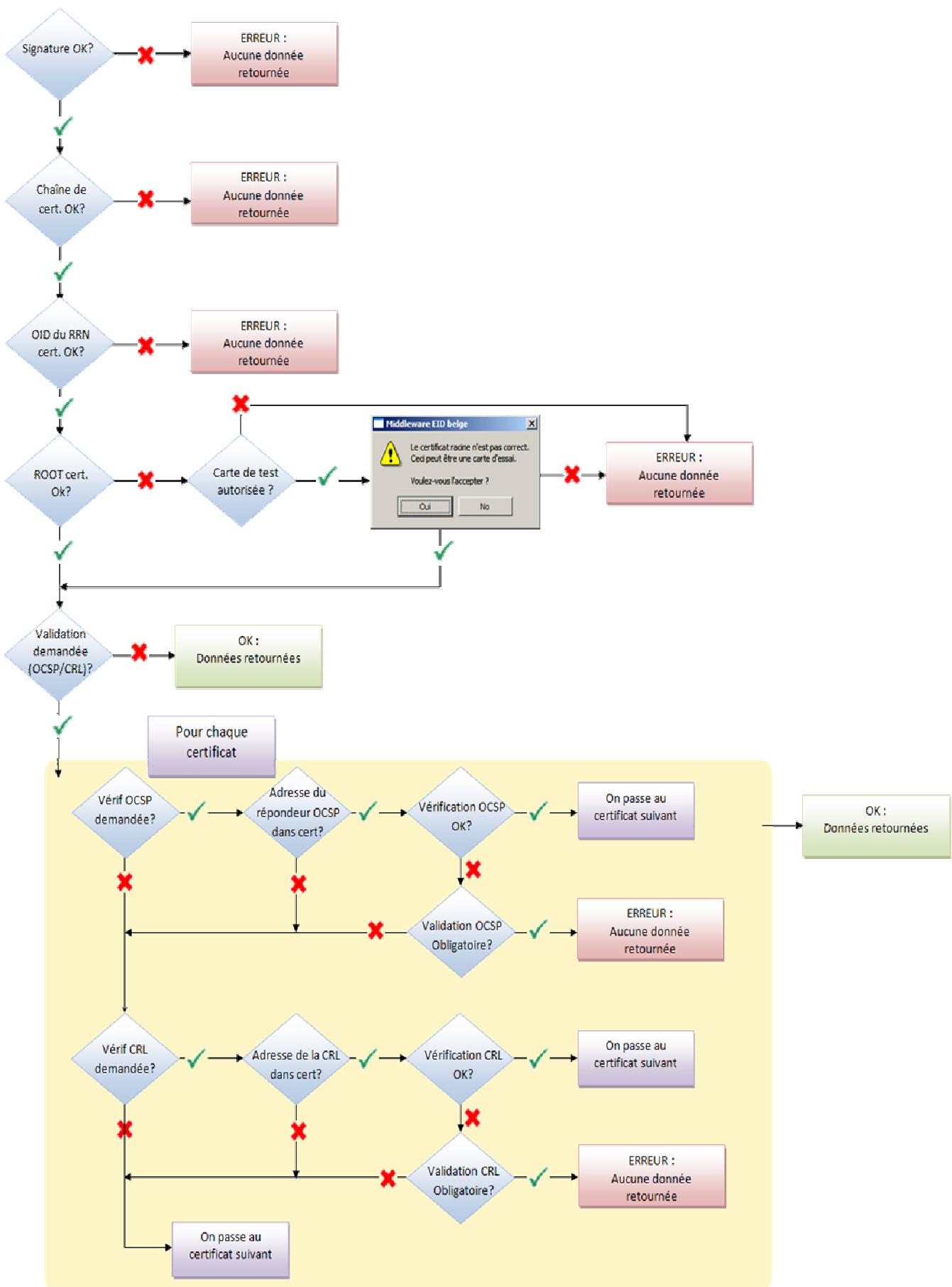
- ◆ le tableau contenant les bytes du certificat;
- ◆ sa longueur;
- ◆ son type (signature, authentification, racine, autorité de certification) sous forme d'une chaîne de caractères;
- ◆ son statut de validité, dont les codes possibles sont :

Value	C-constant	Explanation
0	SEID_CERTSTATUS_CERT_VALIDATED_OK	Validation has occurred successfully.
1	SEID_CERTSTATUS_CERT_NOT_VALIDATED	No validation has been done.
2	SEID_CERTSTATUS_UNABLE_TO_GET_ISSUER_CERT	Unable to get issuer certificate
3	SEID_CERTSTATUS_UNABLE_TO_GET_CRL	Unable to get certificate CRL
4	SEID_CERTSTATUS_UNABLE_TO_DECRYPT_CERT_SIGNATURE	Unable to decrypt certificate's signature
5	SEID_CERTSTATUS_UNABLE_TO_DECRYPT_CRL_SIGNATURE	Unable to decrypt CRL's signature
6	SEID_CERTSTATUS_UNABLE_TO_DECODE_ISSUER_PUBLIC_KEY	Unable to decode issuer public key
7	SEID_CERTSTATUS_CERT_SIGNATURE_FAILURE	Certificate signature failure
8	SEID_CERTSTATUS_CRL_SIGNATURE_FAILURE	CRL signature failure
9	SEID_CERTSTATUS_CERT_NOT_YET_VALID	Certificate is not yet valid
10	SEID_CERTSTATUS_CERT_HAS_EXPIRED	Certificate has expired
11	SEID_CERTSTATUS_CRL_NOT_YET_VALID	CRL is not yet valid
12	SEID_CERTSTATUS_CRL_HAS_EXPIRED	CRL has expired
13	SEID_CERTSTATUS_ERR_IN_CERT_NOT_BEFORE_FIELD	Format error in certificate's notBefore field
14	SEID_CERTSTATUS_ERR_IN_CERT_NOT_AFTER_FIELD	Format error in certificate's notAfter field
15	SEID_CERTSTATUS_ERR_IN_CRL_LAST_UPDATE_FIELD	Format error in CRL's lastUpdate field
16	SEID_CERTSTATUS_ERR_IN_CRL_NEXT_UPDATE_FIELD	Format error in CRL's nextUpdate field
17	SEID_CERTSTATUS_OUT_OF_MEM	Out of memory
18	SEID_CERTSTATUS_DEPTH_ZERO_SELF_SIGNED_CERT	Self signed certificate
19	SEID_CERTSTATUS_SELF_SIGNED_CERT_IN_CHAIN	Self signed certificate in certificate chain
20	SEID_CERTSTATUS_UNABLE_TO_GET_ISSUER_CERT_LOCALLY	Unable to get local issuer certificate
21	SEID_CERTSTATUS_UNABLE_TO_VERIFY_LEAF_SIGNATURE	Unable to verify the first certificate
22	SEID_CERTSTATUS_CERT_CHAIN_TOO_LONG	Certificate chain too long
23	SEID_CERTSTATUS_CERT_REVOKED	Certificate revoked
24	SEID_CERTSTATUS_INVALID_CA	Invalid CA certificate
25	SEID_CERTSTATUS_PATH_LENGTH_EXCEEDED	Path length constraint exceeded
26	SEID_CERTSTATUS_INVALID_PURPOSE	Unsupported certificate purpose
27	SEID_CERTSTATUS_CERT_UNTRUSTED	Certificate not trusted
28	SEID_CERTSTATUS_CERT_REJECTED	Certificate rejected
29	SEID_CERTSTATUS_SUBJECT_ISSUER_MISMATCH	Subject issuer mismatch
30	SEID_CERTSTATUS_AKID_SKID_MISMATCH	Authority and subject key identifier mismatch
31	SEID_CERTSTATUS_AKID_ISSUER_SERIAL_MISMATCH	Authority and issuer serial number mismatch
32	SEID_CERTSTATUS_KEYUSAGE_NO_CERTSIGN	Key usage does not include certificate signing
33	SEID_CERTSTATUS_UNABLE_TO_GET_CRL_ISSUER	Unable to get CRL issuer certificate
34	SEID_CERTSTATUS_UNHANDLED_CRITICAL_EXTENSION	Unhandled critical extension

Plus précisément, la demande d'"informations signées met en route l'algorithme suivant :

```
si signature non valide
    alors données refusées - FIN
    sinon
        si chaîne de certificats vide
            alors données refusées - FIN
            sinon
                si OID du certificat du Registre National non valide
                    alors données refusées - FIN
                    sinon
                        si certificat Root non valide
                            alors si pas de carte de test
                                alors données refusées - FIN
                                sinon si l'utilisateur n'est pas d'accord
                                    alors données refusées - FIN
                                sinon
                            si la validation OCSP ou CRL n'est pas demandée
                                alors données fournies - FIN
                                sinon passage en revue des certificats de la chaîne et contrôle OCPSP ou CRL selon le cas
                                    si échec d'une validation
                                        alors données refusées - FIN
                                        sinon données fournies - FIN
```

Schématiquement, la documentation officielle présente les choses ainsi :



15.2 Les informations de sécurité en Java

Sans surprise, la classe Java **BEID_Certif_Check** correspondante aligne, à côté d'un constructeur par défaut, les méthodes équivalentes :

- ◆ public BEID_Certif_Check()
- ◆ public int getUsedPolicy()
- ◆ public BEID_Certif getCertificate(int Index)
- ◆ public int getCertificatesLength()
- ◆ public int getSignatureCheck()

avec une classe **BEID_Certif** tout aussi prévisible :

- ◆ public BEID_Certif()
- ◆ public byte[] getCertif()
- ◆ public String getCertifLabel()
- ◆ public int getCertifStatus()

Le kit de développement fourni comporte des fonctions d'analyse de ces informations de sécurité et il serait sans doute de peu d'intérêt de les réécrire, si ce n'est que nous en avons fait des méthodes statiques :

CIIinfosIdentite.java (avec informations de sécurité)

```
/*
 * CIIinfosIdentite.java
 */

package carteidentite;

/**
 * @author Vilvens
 */

import be.belgium.eid.*;

public class CIIinfosIdentite
{
    static
    {
        try
        {
            System.loadLibrary("beidlibjni");
        }
        catch (UnsatisfiedLinkError e)
        {
            System.err.println("Impossible de charger la librairie" + e);
            System.exit(1);
        }
    }
}
```

```

public CIInfosIdentite() { }
public static void main(String[] args)
{
    BEID_Status oStatus;
    BEID_Long CardHandle = new BEID_Long();
    oStatus = eidlib.BEID_Init(null, 0, 0, CardHandle);
    System.out.println("BEID_Init : Code de statut = " + oStatus.getGeneral());
    BEID_ID_Data IDData = new BEID_ID_Data();
    BEID_Certif_Check CertCheck = new BEID_Certif_Check();
    oStatus = eidlib.BEID_GetID(IDData, CertCheck);
    System.out.println("BEID_GetID : Code de statut = " + oStatus.getGeneral());

    System.out.println("Informations de sécurité : ");
    System.out.println("Nombre de certificats = " + CertCheck.getCertificatesLength());
PrintSignCheck(CertCheck);

    System.out.println("Carte valide du " + IDData.getValidityDateBegin() + " au " +
        IDData.getValidityDateEnd());
    System.out.println("Commune qui a délivré la carte : " + IDData.getMunicipality());
    System.out.println("Nom : " + IDData.getName());
    System.out.println("Prénom : " + IDData.getFirstName1());
    System.out.println("Lieu de naissance : " + IDData.getBirthLocation());
    System.out.println("Date de naissance : " + IDData.getBirthDate());
    System.out.println("Sexe : " + IDData.getSex());
}

public static void PrintSignCheck (BEID_Certif_Check CertifCheck)
{
    String resultStr[] =
        { "System error", "Valid", "Invalid", "Valid & wrong RRN certificate",
        "Invalid & wrong RRN certificate" };
    String resultStrPol[] = { "None", "OCSP", "CRL", "Both" };

    System.out.println(" *** Signature result : " + CertifCheck.getSignatureCheck() + " " +
        resultStr[CertifCheck.getSignatureCheck() + 1] );

    System.out.println(" *** Certificate checking used policy : " +
        CertifCheck.getUsedPolicy() + " " +
        resultStrPol[CertifCheck.getUsedPolicy()]);
    for ( int i = 0; i < CertifCheck.getCertificatesLength(); i++ )
        PrintCertifCheck( CertifCheck.getCertificate(i) );
    System.out.println("");
}

public static void PrintCertifCheck ( BEID_Certif certif )
{
    System.out.println(" *** Certificate " + certif.getCertifLabel() +
        " Status : " + certif.getCertifStatus() + "(" +
        VerifyCertErrorString(certif.getCertifStatus()) + ")");
}

```

```
public static String VerifyCertErrorString(long n)
{
    switch ((int)n)
    {
        case 0: return("Valid");
        case 1: return("Not validated");
        case 2: return("Unable to get issuer certificate");
        case 3: return("Unable to get certificate CRL");
        case 4: return("Unable to decrypt certificate's signature");
        case 5: return("Unable to decrypt CRL's signature");
        case 6: return("Unable to decode issuer public key");
        case 7: return("Certificate signature failure");
        case 8: return("CRL signature failure");
        case 9: return("Certificate is not yet valid");
        case 10: return("CRL is not yet valid");
        case 11: return("Certificate has expired");
        case 12: return("CRL has expired");
        case 13: return("Format error in certificate's notBefore field");
        case 14: return("Format error in certificate's notAfter field");
        case 15: return("Format error in CRL's lastUpdate field");
        case 16: return("Format error in CRL's nextUpdate field");
        case 17: return("Out of memory");
        case 18: return("Self signed certificate");
        case 19: return("Self signed certificate in certificate chain");
        case 20: return("Unable to get local issuer certificate");
        case 21: return("Unable to verify the first certificate");
        case 22: return("Certificate chain too long");
        case 23: return("Certificate revoked");
        case 24: return ("Invalid CA certificate");
        case 25: return ("Path length constraint exceeded");
        case 26: return ("Unsupported certificate purpose");
        case 27: return ("Certificate not trusted");
        case 28: return ("Certificate rejected");
        case 29: return("Subject issuer mismatch");
        case 30: return("Authority and subject key identifier mismatch");
        case 31: return("Authority and issuer serial number mismatch");
        case 32: return("Key usage does not include certificate signing");
        case 33: return("Unable to get CRL issuer certificate");
        case 34: return("Unhandled critical extension");
        default: return "Unknown status";
    }
}
```

Si les deux certificats ont bien été enregistrés avec la carte dans le lecteur, la boîte de dialogue de confirmation apparaît



et on obtient :

BEID_Init : Code de statut = 0

BEID_GetID : Code de statut = 0

Informations de sécurité :

*** Signature result : 0 Valid

*** Certificate checking used policy : 0 None

*** Certificate RN Status : 1(Not validated)

Carte valide du 20060111 au 30110111

Commune qui a délivré la carte : Oupeye

Nom : Vilvens

Prénom : Claude Marie

Lieu de naissance : Ougrée

Date de naissance : 19680123

Sexe : M

15.3 La récupération d'un objet certificat

Bien sûr, nous récupérons bien ainsi les informations de sécurité, mais la suite de nos développements, basés sur la notion de certificats (comme par exemple SSL), réclamera bien vite que nous disposions d'objets certificats Java. Il suffit pour cela d'utiliser des classes et méthodes déjà décrites dans le chapitre consacré à la cryptographie de vase (Java II, chapitre XIV, paragraphe 12). Pour rappel, on trouve dans le package `java.security.cert` :

- ◆ la classe basique **Certificate**
- ◆ la classe **X509Certificate** dérivée de la précédente;
- ◆ la classe factory **CertificateFactory**, dont on obtient une instance avec la méthode `factory` :

```
public static final CertificateFactory getInstance(String type) throws CertificateException
```

et qui possède la méthode permettant d'obtenir une représentation mémoire d'un certificat lu sur un flux quelconque :

```
public final Certificate generateCertificate(InputStream inStream)
    throws CertificateException
```

Ici, ce flux sera un flux de bytes obtenu à partir des méthodes getCertificate() et getCertif() :

CIIinfosIdentite.java (avec construction d'un objet certificat)

```
import be.belgium.eid.*;
import java.io.ByteArrayInputStream;

import java.security.cert.*;

public class CIIinfosIdentite
{
    static
    {
        try
        {
            System.loadLibrary("beidlibjni");
        }
        catch (UnsatisfiedLinkError e)
        {
            System.err.println("Impossible de charger la librairie" + e); System.exit(1);
        }
    }
    ...
    public static void main(String[] args)
    {
        ...
        try
        {
            CertificateFactory cf = CertificateFactory.getInstance("X509");

            for (int i=0; i<CertCheck.getCertificatesLength(); i++)
            {
                byte[] bc = ((BEID_Certif)CertCheck.getCertificate(i)).getCertif();
                X509Certificate x509c =
                    (X509Certificate)cf.generateCertificate(new ByteArrayInputStream(bc));

                System.out.println("*** Certificat trouvé (" + i+1 + ")");
                System.out.println(x509c);
            }
        }
        catch (CertificateException e)
        {
            System.out.println("Exception de certificat : "+e.getMessage());
        }
        finally
        {
            eidlib.BEID_Exit();
        }
    }
}
```

Le résultat :

BEID_Init : Code de statut = 0

BEID_GetID : Code de statut = 0

Informations de sécurité :

Nombre de certificats = 1

...

*** Certificat trouvé (01)

[

[

Version: V3

Subject: CN=RRN, O=RRN, C=BE

Signature Algorithm: SHA1withRSA, OID = 1.2.854.113696.1.1.5

Key: Sun RSA public key, 1024 bits

modulus: 1587350587756117594...64861894926969697656057997126069653

public exponent: 65537

Validity: [From: Thu Dec 23 12:00:00 CET 2004,

To: Tue Aug 23 12:00:00 CEST 2011]

Issuer: CN=Belgium Root CA, C=BE

SerialNumber: [04111....]

Certificate Extensions: 6

[1]: ObjectId: 2.5.29.14 Criticality=false

SubjectKeyIdentifier [...]

[2]: ObjectId: 2.5.29.35 Criticality=false

AuthorityKeyIdentifier [...]

[3]: ObjectId: 2.5.29.31 Criticality=false

CRLDistributionPoints [

[DistributionPoint:

[URIName: http://crl.eid.belgium.be/belgium.crl]

]

[4]: ObjectId: 2.5.29.32 Criticality=false

CertificatePolicies [...]

[5]: ObjectId: 2.5.29.15 Criticality=true

KeyUsage [

DigitalSignature

Non_repudiation

]

[6]: ObjectId: 2.5.29.19 Criticality=false

BasicConstraints:[...]

]

Algorithm: [SHA1withRSA]

Signature:

```
0000: BF 07 55 73 41 DF 34 38 F3 9C 24 0F 90 BA 4C 9B ..UsA.48..$...L.  
0010: 93 A1 9D 7F 9A D2 BA 21 06 8D AB 4F D2 35 BD 61 .....!...O.5.a  
0020: CB B0 0E 4E A9 A7 47 CB 7E 86 55 BE 14 D2 46 EF ...N..G...U...F.  
....  
00F0: 95 BD 60 B7 8D FF D7 C9 9C 8E AB DB 5F F1 90 88 ..` ....._...
```

]

16. Utilisation de la carte d'identité dans un système de keystores

Nous avons jusqu'à présent manipulé la carte d'identité électronique par l'intermédiaire de son API de base. Il est cependant possible d'utiliser cet outil cryptographique dans le contexte des keystores, plus classique pour une application Java.

16.1 Un keystore PKCS#11

Nous allons hisser la carte d'identité électronique au niveau d'un keystore en utilisant un keystore un peu particulier : il s'agit du Keystore PKCS#11. On se souviendra que PKCS#11 désigne un interface de programmation (encore appelé le **Cryptoki**) qui donne à tout dispositif cryptographique (hardware ou software) une même vue logique (le token ou jeton) à laquelle on accède par un slot qui contient les caractéristiques de ce token. Dans le cas de la carte d'identité, il est tentant de la voir comme une espèce de "keystore embarqué". Pour pouvoir pratiquer ainsi, on utilise un provider particulier, le **Sun PKCS#11 provider**, qui n'implémente pas des algorithmes (comme Cryptix ou Bouncy Castle) mais qui établit une passerelle-écran entre les APIs JCA/JCE et l'interface PKCS#11. En travaillant avec un tel provider, on se met à l'abri de modifications des APIs du PKCS#11, puisque leur usage est encapsulé dans les APIs fournies par le provider.

Pour que nous puissions programmer :

```
CartIdKeystore = KeyStore.getInstance("PKCS11");
```

il nous suffit de disposer d'un provider capable de fournir ce keystore particulier. Nous pourrions l'ajouter dans fichier java.security, en précisant le chemin de la dll (ou de la librairie partagée sous UNIX) fourniissant l'API PKCS#11 :

```
security.provider.7=sun.security.pkcs11.SunPKCS11 c:\WINDOWS\system32\beidpkcs11.dll
```

Mais nous allons plutôt l'ajouter dynamiquement à notre liste de providers au moyen de la méthode de la classe Security :

```
public static int addProvider(Provider provider)
```

De cette manière, nous allons pouvoir configurer notre provider en lui fournissant un fichier de configuration qui n'est rien d'autre qu'un fichier de type properties (donc fichier texte à lignes "clé-valeur"). Ces propriétés (on parle aussi d'"attributs") sont au minimum :

attribut	signification
name	nom du provider, donc ce qui sera renvoyé par la méthode getName() sous la forme "SunPKCS11-<name>"

library	chemin de la librairie d'implémentation de la librairie PKCS#11
---------	---

Mais il en existe d'autres :

attribut	signification
description	description du provider, donc ce qui sera renvoyé par la méthode getInfo() sous la forme "SunPKCS11-<name>"
slot	id du slot fourni (0 par défaut)
slotListIndex	valeur de base pour les sids de slot
enabledMechanisms / disabledMechanisms	permet de spécifier les mécanismes de l'interface PKCS#11 que le provider doit utiliser (ou au contraire ne pas utiliser) avec une valeur représentée par une suite de noms de mécanismes séparés par des espaces, le tout entouré d'accolades – donc, par exemple : enabledMechanisms = { CKM_RSA_PKCS CKM_RSA_PKCS_KEY_PAIR_GEN }
attributes	définition des attributs des clés utilisés par les différents algorithmes selon la syntaxe : attributes (operation, type_clé, algorithme_clé) = { nom_attribut= valeur_attribut [...] } avec ◆ operation : "generate", "import" ou "*"; ◆ type_clé : "CKO_PUBLIC_KEY", "CKO_PRIVATE_KEY", "CKO_SECRET_KEY" ou "*"; ◆ algorithme_clé : "CKK_RSA", "CKK_DSA", "CKK_DH", "CKK_AES", "CKK_DES", "CKK_DES3", "CKK_RC4", "CKK_BLOWFISH", "CKK_GENERIC" ou "*"; ◆ nom_attribut : "CKA_SIGN", "CKA_ENCRYPT", "CKA_DECRYPT", "CKA_VERIFY", "CKA_SENSITIVE", etc. Par exemple : attributes(*,CKO_PRIVATE_KEY,*) = { CKA_SIGN = true } ➔ toutes les clés privées peuvent servir à signer attributes(*,CKO_PRIVATE_KEY,CKK_DH) = { CKA_SIGN = null } ➔ les clés générées par Diffie-Hellman ne peuvent servir à signer.

Dans notre cas, nous allons aller à l'essentiel en définissant "à la dure" le fichier de configuration dans un ByteArrayInputStream :

```
ByteArrayInputStream bais = new ByteArrayInputStream
```

("name = beid\nlibrary = c:\\WINDOWS\\system32\\beidpkcs11.dll".getBytes());
Le provider sera instancié au moyen du constructeur :

```
public SunPKCS11(InputStream configStream)
```

à qui nous passerons ici notre ByteArrayInputStream. Un processus de signature avec la clé de signature de la carte d'identité utilisant le provider PKCS#11 sera finalement celui-ci:

CIKeystore.java

```
package carteidentite;

import java.io.*;
import java.security.*;
import java.security.cert.*;
import java.util.logging.*;

public class CIKeystore
{
    private static ByteArrayInputStream bais = new ByteArrayInputStream
        ("name = beid\nlibrary = c:\\WINDOWS\\system32\\beidpkcs11.dll".getBytes());
    private static final String aliasClesSignatureKeystore = "Signature";
        // L'alias s'appelle obligatoirement ainsi !
    public static void main(String[] args)
    {
        Provider pkcs11Provider = new sun.security.pkcs11.SunPKCS11(bais);
        Security.addProvider(pkcs11Provider);
        KeyStore CartIdKeystore = null;
        try
        {
            // 1. Création du Keystore
            CartIdKeystore = KeyStore.getInstance("PKCS11");
            CartIdKeystore.load(null, null);
            System.out.println("*** KeyStore PKCS11 créé ***");

            // 2. Signature
            PrivateKey clePrivee;
            clePrivee = (PrivateKey) CartIdKeystore.getKey(aliasClesSignatureKeystore,
                null);
            System.out.println("*** Private Key = " + clePrivee.toString());
            Signature s = Signature.getInstance("SHA1withRSA");
            s.initSign(clePrivee);
            String message = "Mot de passe du jour : KlixPataclix";
            s.update(message.getBytes());
            byte[] signature = s.sign();
            System.out.println("*** Signature générée pour " + message + " = " +
                new String(signature)+" ***");

            // 3. Certificat associé
            X509Certificate cert = (X509Certificate)CartIdKeystore.getCertificate(
```

```

aliasClesSignatureKeystore);
System.out.println("Classe instanciée : " + cert.getClass().getName());
System.out.println("Type de certificat : " + cert.getType());
System.out.println("Nom du propriétaire du certificat : " +
cert.getSubjectDN().getName());
PublicKey clePublique = cert.getPublicKey();
System.out.println("... sa clé publique : " + clePublique.toString());
System.out.println("... la classe instanciée par celle-ci : " +
clePublique.getClass().getName());
System.out.println("Dates limites de validité : [" + cert.getNotBefore() + " - " +
cert.getNotAfter() + "]");
System.out.println("Signataire du certificat : " + cert.getIssuerDN().getName());
System.out.println("Algo de signature : " + cert.getSigAlgName());
System.out.println("Signature : " + cert.getSignature());

// 4. Vérification de la signature
Signature sVerif = Signature.getInstance("SHA1withRSA");
sVerif.initVerify(clePublique);
sVerif.update(message.getBytes());
boolean signatureValide = sVerif.verify(signature);
if(signatureValide)System.out.println("*** La signature est valide ***");
else System.out.println("*** La signature n'est pas valide ***");
}

catch (KeyStoreException ex)
{ Logger.getLogger(CIKeystore.class.getName()).log(Level.SEVERE, null, ex); }
catch (UnrecoverableKeyException ex) { ... }
catch (InvalidKeyException ex) { ... }
catch (SignatureException ex) { ... }
catch (IOException ex) { ... }
catch (NoSuchAlgorithmException ex) { ... }
catch (CertificateException ex) { ... }
}
}

```

Résultat :



*** KeyStore PKCS11 créé ***

*** Private Key = SunPKCS11-beid RSA private key, 1024 bits (id 8, token object, sensitive, unextractable)

Classe instanciée : sun.security.x509.X509CertImpl

Type de certificat : X.509

Nom du propriétaire du certificat : SERIALNUMBER=4801230123, GIVENNAME=Claude
Marie, SURNAME=Vilvens, CN=Claude Vilvens (Signature), C=BE

... sa clé publique : Sun RSA public key, 1024 bits

modulus:

9902428351277...8463819593139244702254930510264043561250762119817430120327

public exponent: 65537

... la classe instanciée par celle-ci : sun.security.rsa.RSAPublicKeyImpl

Dates limites de validité : [Tue Jan 24 10:32:14 CET 2006 - Sun Apr 24 11:32:14 CEST 2011]

Signataire du certificat : SERIALNUMBER=999914, CN=Citizen CA, C=BE

Algo de signature : SHA1withRSA

RightSignature

*** La signature est valide ***

On pouvait évidemment extraire les certificats du Citizen CA et du Belgium Root CA – il suffit d'insérer dans le programme précédent :

CIKeystore.java [affichage de la chaîne de certificats]

• • •

```

String[] alias = {"Signature", "CA", "Root"};
X509Certificate[] cert = new X509Certificate[alias.length];
for (int i=0; i<alias.length; i++)
{
    System.out.println("--- Certificat de " + alias[i] + " ---");
    cert[i] = (X509Certificate)CarteIdKeystore.getCertificate(alias[i]);
    System.out.println("Classe instanciée : " + cert[i].getClass().getName());
    System.out.println("Type de certificat : " + cert[i].getType());
    System.out.println("Nom du propriétaire du certificat : " +
        cert[i].getSubjectDN().getName());
    PublicKey clePublique = cert[i].getPublicKey();
    System.out.println("... sa clé publique : " + clePublique.toString());
    System.out.println("... la classe instanciée par celle-ci : " +
        clePublique.getClass().getName());
    System.out.println("Dates limites de validité : [" + cert[i].getNotBefore() + " - " +
        cert[i].getNotAfter() + "]");
    System.out.println("Signataire du certificat : " + cert[i].getIssuerDN().getName());
    System.out.println("Algo de signature : " + cert[i].getSigAlgName());
    System.out.println("Signature : " + cert[i].getSignature());
}

```

Résultat :

--- Certificat de Signature ---

Classe instanciée : sun.security.x509.X509CertImpl

Type de certificat : X.509

Nom du propriétaire du certificat : SERIALNUMBER=4801230123, GIVENNAME=Claude Marie, SURNAME=Vilvens, **CN=Claude Vilvens (Signature)**, C=BE

... sa clé publique : Sun RSA public key, 1024 bits
modulus:

9902428351277...8463819593139244702254930510264043561250762119817430120327
public exponent: 65537

... la classe instanciée par celle-ci : sun.security.rsa.RSAPublicKeyImpl

Dates limites de validité : [Tue Jan 24 10:32:14 CET 2006 - Sun Apr 24 11:32:14 CEST 2011]

Signataire du certificat : SERIALNUMBER=999914, **CN=Citizen CA**, C=BE

Algo de signature : SHA1withRSA

Signature : [B@16f0472

--- Certificat de CA ---

Classe instanciée : sun.security.x509.X509CertImpl

Type de certificat : X.509

Nom du propriétaire du certificat : SERIALNUMBER=200514, **CN=Citizen CA**, C=BE

... sa clé publique : Sun RSA public key, 2048 bits
modulus:

199556247067....04092777457742839785496354823829944925675596374967611
public exponent: 65537

... la classe instanciée par celle-ci : sun.security.rsa.RSAPublicKeyImpl

Dates limites de validité : [Thu Dec 23 12:00:00 CET 2004 - Tue Aug 23 12:00:00 CEST 2011]

Signataire du certificat : **CN=Belgium Root CA**, C=BE

Algo de signature : SHA1withRSA

Signature : [B@18d107f

--- Certificat de Root ---

Classe instanciée : sun.security.x509.X509CertImpl

Type de certificat : X.509

Nom du propriétaire du certificat : **CN=Belgium Root CA**, C=BE

... sa clé publique : Sun RSA public key, 2048 bits
modulus:

2532727247174242475310876114....798430948172071644141969017225065301229219951
public exponent: 65537

... la classe instanciée par celle-ci : sun.security.rsa.RSAPublicKeyImpl

Dates limites de validité : [Mon Jan 27 00:00:00 CET 2003 - Mon Jan 27 00:00:00 CET 2014]

Signataire du certificat : **CN=Belgium Root CA**, C=BE

Algo de signature : SHA1withRSA

Signature : [B@360be0

Bien sûr, si le lecteur de cartes n'est pas connecté ou si la carte n'est pas insérée, on obtient :

GRAVE: null

java.security.KeyStoreException: **PKCS11 not found**

at java.security.KeyStore.getInstance(KeyStore.java:587)
at carteidentite.CIKeystore.main(CIKeystore.java:48)

Caused by: java.security.NoSuchAlgorithmException: **PKCS11 KeyStore not available**

at sun.security.jca.GetInstance.getInstance(GetInstance.java:142)
at java.security.Security.getImpl(Security.java:659)
at java.security.KeyStore.getInstance(KeyStore.java:584)
... 1 more

Evidemment, nous avons utilisé la même keystore pour signer et vérifier la signature. En pratique, le destinataire d'un message signé au moyen de la carte d'identité doit seulement disposer d'un keystore contenant les certificats nécessaires à la vérification.

16.2 L'exportation de certificats du magasin de Windows

On se souviendra que l'on peut parcourir le magasin à clé de Windows avec la Microsoft Management Console :

C:\Documents and Settings\Claude>**mmc**

ou encore plus directement avec

Exécuter ...

certmgr.msc

Nous avons ainsi installé un "snapin" de gestion des certificats. Un clic droit sur le certificat rootCA (par exemple) donne un menu contextuel permettant de réaliser l'exportation de ce certificat dans un fichier .cer :



Assistant Exportation de certificat

Bienvenue !

Cet Assistant vous aide à copier des certificats, des listes de certificats de confiance et des listes de révocation de certificats depuis le magasin de certificats vers votre disque dur.

Un certificat, émis par une Autorité de certification, est une confirmation de votre identité et contient des informations utilisées pour protéger vos données ou établir des connexions réseau sécurisées. Le magasin de certificats est la zone système où les certificats sont conservés.

Pour continuer, cliquez sur Suivant.

< Précédent Suivant > Annuler

Format de fichier d'exportation

Les certificats peuvent être exportés sous plusieurs formats de fichier.

Sélectionnez le format à utiliser :

- Binaire codé X.509 (.cer)
- Codé à base 64 X.509 (.cer)
- Standard de syntaxe de message cryptographique - Certificats PKCS #7 (.p7b)

Inclure tous les certificats dans le chemin d'accès de certification si possible

Échange d'informations personnelles - PKCS #12 (.pfx)

Inclure tous les certificats dans le chemin d'accès de certification si possible

Activer la protection renforcée (nécessite IE 5.0, NT 4.0 SP4 ou supérieur)

Supprimer la clé privée si l'exportation s'est terminée correctement

< Précédent Suivant > Annuler

Assistant Exportation de certificat

Fichier à exporter

Spécifiez le nom du fichier à exporter

Nom du fichier : C:\java-netbeans-application\CarteIdentite\BelgiumRootCA.cer

Parcourir...

Fin de l'Assistant Exportation de certificat

Vous avez terminé correctement l'Assistant Exportation de certificat.

Vous avez spécifié les paramètres suivants :

- Nom du fichier
- Exporter les clés
- Inclure tous les certificats dans le chemin d'accès de ce
- Format de fichier

< Précédent Terminer Annuler

Assistant Exportation de certificat

L'exportation s'est effectuée correctement.

OK

On peut faire de même pour le certificat Citizen et celui de signature, mais pour ce dernier on notera des propositions supplémentaires et une question additionnelle :

Racine de la console\Certificats - Utilisateur actuel\Personnel\Certificats

Délivré à	Délivré par	Date
Claude Vilvens (Authentication)	Citizen CA	24/04
Claude Vilvens (Signature)	en CA	24/04
Myriam Peruzzi (Signature)	en CA	9/02
willows		

Ouvrir Toutes les tâches Ouvrir

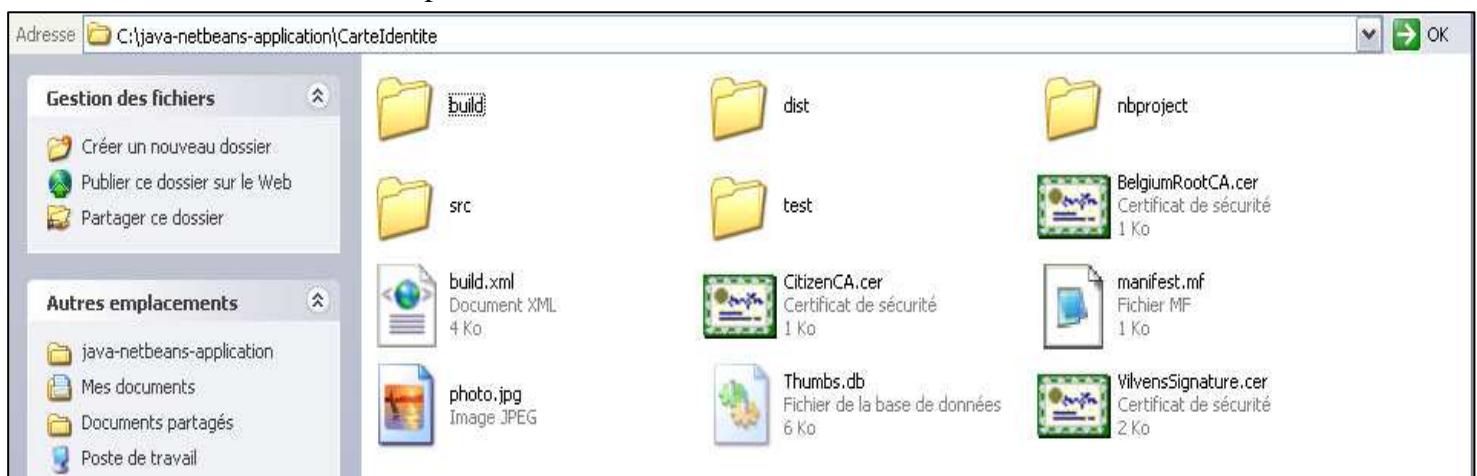
- Couper
- Copier
- Supprimer
- Propriétés
- Aide

Demander un certificat avec une nouvelle clé...
Demander un certificat avec la même clé...
Renouveler le certificat avec une nouvelle clé...
Renouveler le certificat avec la même clé...
Exporter...

Afficher un certificat



On en arrive ainsi à disposer des fichiers :



16.3 Un keystore PKCS#12

Il nous reste à présent à intégrer ces certificats dans un keystore PKCS12, ce qui ne présente pas de difficulté particulière si on sait que

- ♦ on sait créer un objet Certificate à partir d'un fichier DER au moyen de la méthode de **CertificateFactory** :

```
public final Certificate generateCertificate(InputStream inStream)
    throws CertificateException
```

- ♦ on fait entrer un certificat dans un objet Keystore avec la méthode de **Keystore** :

```
public final void setCertificateEntry(String alias, Certificate cert)
    throws KeyStoreException
```

- ♦ on sauve le contenu de l'objet Keystore dans un fichier keystore au moyen de la méthode :

```
public final void store(OutputStream stream, char[] password)
```

```
throws KeyStoreException, IOException, NoSuchAlgorithmException,
CertificateException
```

Le programme correspondant s'écrit alors très simplement selon les étapes suivantes :

- ◆ création d'un Keystore vide;
- ◆ création des trois objets certificats et insertion dans le Keystore;
- ◆ sauvegarde du Keystore.

CIKeystorePkcs12.java

```
package carteidentite;

import java.io.*;
import java.security.*;
import java.security.cert.*;
import java.util.logging.*;

/**
 * @author Vilvens
 */

public class CIKeystorePkcs12
{
    private static String codeProvider = "BC";
    public static void main(String[] args)
    {
        KeyStore ks = null;
        char[] pwdKeystore = "pwdpwd".toCharArray();
        try
        {
            ks = KeyStore.getInstance("PKCS12", codeProvider);
            ks.load(null, pwdKeystore);
        }
        catch (NoSuchProviderException ex)
        {
            Logger.getLogger(CIKeystorePkcs12.class.getName()).log(Level.SEVERE, null,
                ex);
        }
        catch (KeyStoreException ex)
        {
            Logger.getLogger(CIKeystorePkcs12.class.getName()).log(Level.SEVERE, null,
                ex);
        }
        catch (IOException ex)
        {
            Logger.getLogger(CIKeystorePkcs12.class.getName()).log(Level.SEVERE, null,
                ex);
        }
        catch (NoSuchAlgorithmException ex)
        {
            Logger.getLogger(CIKeystorePkcs12.class.getName()).log(Level.SEVERE, null,
                ex);
        }
        catch (CertificateException ex)
        {
            Logger.getLogger(CIKeystorePkcs12.class.getName()).log(Level.SEVERE, null,
                ex);
        }
        System.out.println("Keystore créé");
    }
}
```

```

System.out.println("Répertoire courant : " + System.getProperty("user.dir"));
FileInputStream[] inStream = new FileInputStream[3];
String[] nomsFichiersCertificats = { "BelgiumRootCA.cer", "CitizenCA.cer",
    "VilvensSignature.cer" };
String[] alias = { "BelgiumRootCert", "CitizenCert", "VilvensSignatureCert" };
String nomsFichierKeystore = "KeystoreCarteIdentite";
CertificateFactory cf = null;
try
{
    cf = CertificateFactory.getInstance("X.509");
    for (int i = 0; i < nomsFichiersCertificats.length; i++)
        try
        {
            inStream[i] = new FileInputStream(System.getProperty("user.dir") +
                System.getProperty("file.separator") + nomsFichiersCertificats[i]);
            X509Certificate cert =
                (X509Certificate) cf.generateCertificate(inStream[i]);
            System.out.println("Objet certificat " + nomsFichiersCertificats[i] + " créé !");
            ks.setCertificateEntry(alias[i], cert);
        }
        catch (FileNotFoundException e)
        {
            System.out.println("Fichier certificat : " + e.getMessage());
        }
        catch (KeyStoreException e)
        {
            System.out.println("Certificat dans keystore : " + e.getMessage());
        }

    ks.store(new FileOutputStream(System.getProperty("user.dir") +
        System.getProperty("file.separator") + nomsFichierKeystore), pwdKeystore);
    System.out.println("Fichier keystore " + nomsFichierKeystore + " créé !");
}
catch (KeyStoreException ex)
{
    Logger.getLogger(CIKeystorePkcs12.class.getName()).log(Level.SEVERE, null,
        ex);
}
catch (IOException ex) { ... }
catch (NoSuchAlgorithmException ex) { ... }
catch (CertificateException ex) { ... }
}
}

```

16.4 La vérification d'une signature avec un keystore PKCS#12

Dans de telles conditions, tout qui possède ce fichier keystore "KeystoreCarteIdentite" est capable de vérifier une signature réalisée au moyen de la carte d'identité, cela sans jamais avoir du lire la carte sur sa machine. Ainsi, si nous reprenons un programme précédent qui signe un message avec cette carte d'identité et qui sérialise ce message et sa signature :

CIKeystore.java (signature et sérialisation)

```
package carteidentite;
```

```
import java.io.*;
import java.security.*;
```

```

import java.security.cert.*;
import java.util.logging.*;

public class CIKeystore
{
    private static ByteArrayInputStream bais = new ByteArrayInputStream
        ("name = beid\nlibrary = c:\\\\WINDOWS\\\\system32\\\\beidpkcs11.dll".getBytes());
    private static final String aliasClesSignatureKeystore = "Signature";
        // L'alias s'appelle obligatoirement ainsi !
    public static void main(String[] args)
    {
        Provider pkcs11Provider = new sun.security.pkcs11.SunPKCS11(bais);
        Security.addProvider(pkcs11Provider);
        KeyStore CartIdKeystore = null;
        try
        {
            CartIdKeystore = KeyStore.getInstance("PKCS11");
            CartIdKeystore.load(null, null);
            System.out.println("*** KeyStore PKCS11 créé ***");

            PrivateKey clePrivee;
            clePrivee = (PrivateKey) CartIdKeystore.getKey(aliasClesSignatureKeystore,
null);
            System.out.println("*** Private Key = " + clePrivee.toString());
            Signature s = Signature.getInstance("SHA1withRSA");
            s.initSign(clePrivee);
            String message = "Mot de passe du jour : KlixPataclix";
            s.update(message.getBytes());

            byte[] signature = s.sign();
            System.out.println("*** Signature générée pour " + message + " = "+
                new String(signature)+" ***");

            // NEW : Sérialisations : pour vérification carte identité
            ObjectOutputStream oos = new ObjectOutputStream (
                new FileOutputStream(System.getProperty("user.dir") +
                    System.getProperty("file.separator") + "messagedujour"));
            oos.writeObject(message);oos.close();
            System.out.println("Fichier message créé !");
            oos = new ObjectOutputStream (
                new FileOutputStream(System.getProperty("user.dir") +
                    System.getProperty("file.separator") + "signaturemessagedujour"));
            oos.writeObject(signature);oos.close();
            System.out.println("Fichier signature créé !");
        }
        catch (KeyStoreException ex) { ... }
        ...
    }
}

```

alors tout qui possède le fichier KeystoreCarteIdentite est capable de vérifier cette signature :

VerifCIKeystore.java

```
package carteidentite;

import java.io.*;
import java.security.*;
import java.security.cert.*;
import java.util.logging.*;

/**
 *
 * @author Vilvens
 */
public class VerifCIKeystore
{
    private static String codeProvider = "BC";
    public static void main(String[] args)
    {
        try
        {
            KeyStore CartIdKeystore = null;
            String nomsFichierKeystore = "KeystoreCarteIdentite";
            char[] pwdKeystore = "pwdpwd".toCharArray();
            String alias = "VilvensSignatureCert";
            CartIdKeystore = KeyStore.getInstance("PKCS12", codeProvider);
            CartIdKeystore.load(new FileInputStream(System.getProperty("user.dir") +
                System.getProperty("file.separator")) + nomsFichierKeystore, pwdKeystore);
            X509Certificate certS = (X509Certificate)CartIdKeystore.getCertificate(alias);
            PublicKey clePublique = certS.getPublicKey();

            // Désérialisation du message et de sa signature
            // Sérialisations : pour vérification carte identité
            ObjectInputStream ois = new ObjectInputStream (
                new FileInputStream(System.getProperty("user.dir") +
                    System.getProperty("file.separator") + "messagedujour"));
            String message = null;
            byte[] signature= null;
            try
            {
                message = (String) ois.readObject();
                ois.close();
                System.out.println("Message lu : " + message);
                ois = new ObjectInputStream (
                    new FileInputStream(System.getProperty("user.dir") +
                        System.getProperty("file.separator") + "signaturemessagedujour"));
                signature = (byte[]) ois.readObject();
                ois.close();
            }
        }
    }
}
```

```

        catch (ClassNotFoundException ex)
        {Logger.getLogger(VerifCIKeystore.class.getName()).log(Level.SEVERE,null,ex); }
        System.out.println("Fichier signature créé !");

Signature sVerif = Signature.getInstance("SHA1withRSA");
sVerif.initVerify(clePublique);
sVerif.update(message.getBytes());
boolean signatureValide = sVerif.verify(signature);
if(signatureValide)System.out.println("*** La signature est valide ***");
else System.out.println("*** La signature n'est pas valide ***");
}
catch (KeyStoreException ex)
{ Logger.getLogger(VerifCIKeystore.class.getName()).log(Level.SEVERE,null,ex);}
catch (NoSuchProviderException ex) { ...}
catch (InvalidKeyException ex) { ...}
catch (SignatureException ex) { ...}
catch (IOException ex) { ...}
catch (NoSuchAlgorithmException ex) { ...}
catch (CertificateException ex) { ...}
}
}
}

```

Et, bien sûr, la signature est vérifiée avec succès ☺ ! Mais encore faut-il que le certificat utilisé soit toujours valide ...

17. Le principe de l'utilisation de la carte d'identité pour l'authentification d'une MIDlet

Il est à priori tentant d'imaginer d'authentifier une MIDlet au moyen des outils fournis par la carte d'identité électronique. Il n'est cependant pas possible de procéder comme nous l'avons fait dans le chapitre consacré à J2ME (paragraphe 20 : "Une MIDlet authentifiée") car *la clé privée utilisée pour la signature ne peut être extraite de la carte d'identité pour être placée dans un keystore*, keystore dans lequel Netbeans entend la trouver ☹ ...

L'idée sera donc plus artisanale. Nous allons signer le fichier jar de la MIDlet par programmation (avec signature au moyen de la carte d'identité), puis nous récupérerons, au format base64, la signature et le certificat, que nous n'aurons plus qu'à recopier dans le fichier jad associé à notre midlet. Nous utilisons pour cela la classe **Base64** de Bouncy Castle (package org.bouncycastle.util.encoders) dont le rôle est clairement de fournir les méthodes (statiques) de conversion d'une chaîne de caractères ou d'un tableau de bytes en une chaîne base64 et vice-versa. Parmi les versions polymorphes disponibles, nous utiliserons :

```
public static int encode(byte[] data, java.io.OutputStream out) throws java.io.IOException
```

La signature et le certificat seront ainsi transformés en chaînes base64 que nous écrirons dans deux fichiers. Cela donne :

CIJ2MESignatureMIDlet.java

```
package carteidentite;
```

```
import java.io.*;
```

```

import java.security.*;
import java.security.cert.*;
import java.util.logging.*;
import org.bouncycastle.util.encoders.Base64;

/**
 * @author Vilvens
 */

public class CIJ2MESignatureMIDlet
{
    private static ByteArrayInputStream bais = new ByteArrayInputStream
        ("name = beid\nlibrary = c:\\WINDOWS\\system32\\beidpkcs11.dll".getBytes());
    private static final String aliasClesSignatureKeystore = "Signature";
        // L'alias s'appelle obligatoirement ainsi !
    public static void main(String[] args)
    {
        Provider pkcs11Provider = new sun.security.pkcs11.SunPKCS11(bais);
        Security.addProvider(pkcs11Provider);
        KeyStore CartIdKeystore = null;
        try
        {
            CartIdKeystore = KeyStore.getInstance("PKCS11");
            CartIdKeystore.load(null, null);
            System.out.println("*** KeyStore PKCS11 créé ***");

            PrivateKey clePrivee;
            clePrivee = (PrivateKey) CartIdKeystore.getKey(aliasClesSignatureKeystore,
                null);
            System.out.println("*** Private Key = " + clePrivee.toString());
            Signature s = Signature.getInstance("SHA1withRSA");
            s.initSign(clePrivee);
            String nomDuJar = "C:\\java-netbeans-application\\MidletPersonalData\\\
                dist\\MidletPersonalData.jar";
            FileInputStream fis = new FileInputStream(nomDuJar);
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            int b = 0;
            do
            {
                b = fis.read();
                if(b != -1) baos.write(b);
            }
            while(b != -1);
            byte[] bytesASigner = baos.toByteArray();
            s.update(bytesASigner);
            byte[] signature = s.sign();
            ByteArrayOutputStream baos64 = new ByteArrayOutputStream();
            Base64.encode(signature, baos64);
            byte[] signatureBase64 = baos64.toByteArray();
            System.out.println("*** Signature générée pour le jar = "+signatureBase64 + "

```

```

***");
DataOutputStream dos = new DataOutputStream (new FileOutputStream(
    "C:\\java-netbeans-application\\signatureBase64"));
dos.write(signatureBase64, 0, signatureBase64.length);dos.close();
System.out.println("Fichier signature créé !");

String[] alias = { "Signature", "CA", "Root"};
X509Certificate[] cert = new X509Certificate[alias.length];
for (int i=0; i<alias.length; i++)
{
    System.out.println("--- Certificat de " + alias[i] + " ---");
    cert[i] = (X509Certificate)CartIdKeystore.getCertificate(alias[i]);
    System.out.println("Classe instanciée : " + cert[i].getClass().getName());
    System.out.println("Type de certificat : " + cert[i].getType());
    System.out.println("Nom du propriétaire du certificat : " +
        cert[i].getSubjectDN().getName());
    PublicKey clePublique = cert[i].getPublicKey();
    System.out.println("... sa clé publique : " + clePublique.toString());
    System.out.println("... la classe instanciée par celle-ci : " +
        clePublique.getClass().getName());
    System.out.println("Dates limites de validité : [" + cert[i].getNotBefore() +
        " - " + cert[i].getNotAfter() + "]");
    System.out.println("Signataire du certificat : " + cert[i].getIssuerDN().
        getName());
    System.out.println("Algo de signature : " + cert[i].getSigAlgName());
    System.out.println("Signature : " + cert[i].getSignature());
}
X509Certificate certS = (X509Certificate)CartIdKeystore.getCertificate(
    aliasClesSignatureKeystore);
PublicKey clePublique = certS.getPublicKey();
dos = new DataOutputStream (new FileOutputStream(
    "C:\\java-netbeans-application\\certificatBase64"));
baos64 = new ByteArrayOutputStream();
Base64.encode(certS.getEncoded(),baos64);
dos.write(baos64.toByteArray(), 0, baos64.toByteArray().length);dos.close();
System.out.println("Fichier certificat créé !");

Signature sVerif = Signature.getInstance("SHA1withRSA");
sVerif.initVerify(clePublique);
sVerif.update(bytesASigner);
boolean signatureValide = sVerif.verify(signature);
if(signatureValide)System.out.println("**** La signature est valide ****");
else System.out.println("**** La signature n'est pas valide ****");
}
catch (KeyStoreException ex)
{
    Logger.getLogger(CIKeystore.class.getName()).log(Level.SEVERE, null, ex);
}
catch (UnrecoverableKeyException ex) { ... }
catch (InvalidKeyException ex) { ... }

```

```

        catch (SignatureException ex) { ... }
        catch (IOException ex) { ... }
        catch (NoSuchAlgorithmException ex) { ... }
        catch (CertificateException ex) { ... }
    }
}

```

Après l'exécution de ce programme, nous disposons ainsi des deux fichiers contenant les lignes à introduire dans le fichier jad. Nous supposons aussi avoir extrait du magasin à clés les certificats EID qu'il nous faudra placer sur le mobile afin que la vérification de la signature soit possible :



MIID/jCCAuagAwIBAgIQEAAAAAAAudcSf8g73EFnpTANBgkqhkiG9w0BAQUFADAz
MQswCQYDVQQGEwJCRTETMBEGA1UEAxMKQ2l0aXplbiBDQTEPMA0GA1UEBR
MGMjAwNTE0MB4XDTA2MDEyNDA5MzIxNFoXDTExMDQyNDA5MzIxNFowcTEL
MAkGA1UEBhMCQkUxIzAhBgNVBAMTGjNsYXVkJSBWaWx2ZW5zIChTaWduYXR
1cmUpMRAwDgYDVQQEEwdWaWx2ZW5zMRUwEwYDVQQqEwxDbGF1ZGUgTWFy
aWUxFDASBgNVBAUTCzU4MDEyMzAzOTEzMIGfMA0GCSqGSIb3DQEBAQUAA4G
NADCBiQKBgQCNA+gddgbCA4xSyATKiflar8c16f0BvuC1BspzZRssSJxJsIPQpg6rR7dY
KB0JBULXMMHvRnHP4FRVzNt5qms9rXx4doKcCliSwHR9P6adxE+7yfDBBOvDaEZ17
+Ft51ObpD7OFcbjwV/1lK8F/I2fwpi9O6xgFGTCCWSwX03hwIDAQABo4IBUjCCAU4
wRAYDVR0gBD0wOzA5BgdgOAEBQIBMC4wLAYIKwYBBQUHAgEWIGh0dHA6Ly
9yZXBvc2l0b3J5LmVpZC5iZWxnaXVtLmJIMA4GA1UdDwEB/wQEAvIGQDAfBgNVHS
MEGDAwBR52pE0s0MnznaVXkcljZrZB1JymTA5BgNVHR8EMjAwMC6gLKAqhihodH
RwOi8vY3JsLmVpZC5iZWxnaXVtLmJIL2VpZGMyMDA1MTQuY3JsMBEGCWCGSAG
G+EIBAQQEAvIFIDBtBgggrBgfFBQcBAQRhMF8wNQYIKwYBBQUHMAKGKWh0dH
A6Ly9jZXJ0cy5laWQuYmVsZ2l1bS5iZS9iZWxnaXVtcnMuY3J0MCYGCCsGAQUFBzAB
hhpodHRwOi8vb2NzcC5laWQuYmVsZ2l1bS5iZTAYBgggrBgfFBQcBAwQMMIAowCAY
GBACORgEBMA0GCSqGSIb3DQEBBQUAA4IBAQBqvCv2zh0WcQ0lglO08LF1VptQUf
MTs5Bps3jh2S5PJFUaVeZCGgnXhdHKuFsDEwpKoAJuij5kztdnp9PALXR72QrcSXTusE
KHbiUfpzX8s3qof6ivjaCe06bAzcSmYIX8q9S2JiWazhy8xSjhCOLxWRmOFgrAViz3cwWZ
FqpjP6a8lJm+ndmr+NkguFURm+BxF9OtUk6sVBMX956bjm4IjFky+iXSvVTP/DJRM2fm9
FcBgdasu1ghu5zgQR3qxXCIB/ud2ChPVjoxDQI5fH298gPpz4xf16XPitS2rjaQ3SdRDCk6sz
Rui9OGnW6LhLmF7dBjHXfzc/4zbe1A6zY

X3AdHvF6JQHbmHCgvfqjyXyOV9/mq5bn/T90L4dD4fnpy77xy0pIKiV4TvE0id6bwB7W7
M7h9/2sGvH8XMPkFuUj7j4z2ytr1A7vBqlksQunIfwwvaM1Mbd1CUm1kIRGqjs9AXs+uU
DlsO7moIkNz2zskO4iKBt5KZ0W3T/tb3w=

Il ne reste plus qu'à rectifier le jad de notre MIDlet (initialement non signé) :

MidletPersonalData.jad

MIDlet-1: HelloMIDlet, , hello.HelloMIDlet

MIDlet-Certificate-1-1:

MIID/jCCAuagAwIBAgIQEAAAAAAudcSf8g73EFnpTANBgkqhkiG9w0BAQUFADAz
MQswCQYDVQQGEwJCRTETMBEGA1UEAxMKQ2l0aXplbiBDQTEPMA0GA1UEBR
MGMjAwNTE0MB4XDTA2MDEyNDA5MzIxNFoXDTExMDQyNDA5MzIxNFowcTEL
.....FcBgdasu1ghu5zgQR3qxXCIb/ud2ChPVjoxDQI5fH298gPpz4xf16XPitS2rjiaQ3Sd
DCk6szRui9OGnW6LhLmF7dBjHXfzc/4zbe1A6zY

MIDlet-Jar-RSA-SHA1:

X3AdHvF6JQHbmHCgvfqjyXyOV9/mq5bn/T90L4dD4fnpy77xy0pIKiV4TvE0id6bw7W7
M7h9/2sGvH8XMPkFuUj7j4z2ytr1A7vBqlksQunIfwwvaM1Mbd1CUm1kIRGqjs9AXs+uU
DlsO7moIkNz2zsK04iKBt5KZ0W3T/tb3w=

MIDlet-Jar-Size: 3213

MIDlet-Jar-URL: MidletPersonalData.jar

MIDlet-Name: MidletPersonalData

MIDlet-Vendor: Vendor

MIDlet-Version: 1.0

MicroEdition-Configuration: CLDC-1.1

MicroEdition-Profile: MIDP-2.0

Il n'y a plus qu'à emmener tout ce qui est nécessaire sur le mobile, en espérant qu'il est capable de vérifier une signature sur base d'une chaîne de certificats et pas d'un certificat unique ... Et à propos de certificats, sont-ils toujours valides ?

18. La vérification d'un certificat par OCSP

On se souviendra que les certificats des cartes d'identité peuvent être vérifiés soit par consultation locale de fichiers **CRL** (**Certificate Revocation List**) contenant la liste des certificats révoqués, soit par une requête à un serveur **OCSP** (**Online Certification Status Protocol**) chargé de répondre en temps réel aux demandes de vérification de validité.

Nous allons programmer ici la vérification du certificat de signature par une requête OCSP, plus simple pour le client, utilisant des informations fraîches et assurant aussi que c'est le serveur qui effectuera l'éventuelle remontée d'une chaîne de certificats. Mais ceci implique d'en savoir un peu plus sur ce protocole, puisque nous allons devoir rédiger une requête conforme à ce protocole et être capable d'en décoder une réponse.

18.1 Description du protocole OCSP

OCSP (**Online Certification Status Protocol**) est un protocole applicatif qui s'appuie sur HTTP. Il est défini dans la RFC 2560, dans le contexte PKI X509.

a) la requête

Elle comporte typiquement

1) une composante de base (obligatoire pour une requête valide) qui contient, outre la version du protocole (par défaut, c'est la version 1), un certain nombre (donc une liste de 1 élément minimum) de requêtes de validité d'un certificat :

- ♦ le digest du nom (DN) de l'émetteur (ce qui peut être calculé à partir du certificat en question);

- ◆ le digest de la clé publique de l'émetteur (ce que l'on peut calculer à partir du certificat de l'émetteur);
- ◆ l'identification de l'algorithme de hashage utilisé pour le calcul de ces deux digests;
- ◆ le numéro de série du certificat dont on veut vérifier la validité.

2) une composante optionnelle qui est en fait la signature du demandeur (flanqué de l'identification de l'algorithme et éventuellement de la chaîne de certificats de vérification).

Diverses extensions sont possibles dans ces deux composantes. Plus précisément; la structure de la trame est définie en ASN.1 comme suit :

```

OCSPRequest ::= SEQUENCE {
    tbsRequest      TBSRequest,
    optionalSignature [0] EXPLICIT Signature OPTIONAL }

TBSRequest ::= SEQUENCE {
    version        [0] EXPLICIT Version DEFAULT v1,
    requestorName  [1] EXPLICIT GeneralName OPTIONAL,
    requestList     SEQUENCE OF Request,
    requestExtensions [2] EXPLICIT Extensions OPTIONAL }

Signature ::= SEQUENCE {
    signatureAlgorithm AlgorithmIdentifier,
    signature          BIT STRING,
    certs              [0] EXPLICIT SEQUENCE OF Certificate OPTIONAL}

Version ::= INTEGER { v1(0) }

Request ::= SEQUENCE {
    reqCert       CertID,
    singleRequestExtensions [0] EXPLICIT Extensions OPTIONAL }

CertID ::= SEQUENCE {
    hashAlgorithm   AlgorithmIdentifier,
    issuerNameHash OCTET STRING, -- Hash of Issuer's DN
    issuerKeyHash   OCTET STRING, -- Hash of Issuers public key
    serialNumber    CertificateSerialNumber }

```

b) la réponse

La réponse, quel que soit son contenu, doit être signée

- ◆ soit par le CA qui a émis le certificat;
- ◆ soit par un organisme dont le certificat est accepté par le requérant;
- ◆ soit par un autre CA qui se fait admettre en exhibant un certificat émis par le CA émetteur de base dans le but de valider des réponses OCSP pour ce CA émetteur.

Une réponse OCSP est formée par un code de retour (succès, requête mal formée, erreur interne, il faut réessayer plus tard, etc) et d'un objet réponse proprement dit qui va fournir essentiellement le nom du répondant, la signature calculée sur le hashage de la réponse et,

pour chaque certificat de la liste de requête, le statut (good, revoked ou unknown) et l'intervalle de validité.

Plus précisément; la structure de la trame est définie en ASN.1 comme suit :

```
OCSPResponse ::= SEQUENCE {
    responseStatus      OCSPResponseStatus,
    responseBytes       [0] EXPLICIT ResponseBytes OPTIONAL }
```

```
OCSPResponseStatus ::= ENUMERATED {
    successful          (0), --Response has valid confirmations
    malformedRequest    (1), --Illegal confirmation request
    internalError       (2), --Internal error in issuer
    tryLater            (3), --Try again later
                        --(4) is not used
    sigRequired         (5), --Must sign the request
    unauthorized        (6)  --Request unauthorized
}
```

```
ResponseBytes ::= SEQUENCE {
    responseType   OBJECT IDENTIFIER,
    response        OCTET STRING }
```

-- For a basic OCSP responder, **responseType** will be id-pkix-ocsp-basic.

```
id-pkix-ocsp      OBJECT IDENTIFIER ::= { id-ad-ocsp }
id-pkix-ocsp-basic  OBJECT IDENTIFIER ::= { id-pkix-ocsp 1 }
```

```
BasicOCSPResponse ::= SEQUENCE {
    tbsResponseData   responseData,
    signatureAlgorithm AlgorithmIdentifier,
    signature         BIT STRING,
    certs            [0] EXPLICIT SEQUENCE OF Certificate OPTIONAL }
```

```
responseData ::= SEQUENCE {
    version          [0] EXPLICIT Version DEFAULT v1,
    responderID     ResponderID,
    producedAt       GeneralizedTime,
    responses        SEQUENCE OF SingleResponse,
    responseExtensions [1] EXPLICIT Extensions OPTIONAL }
```

```
ResponderID ::= CHOICE {
    byName           [1] Name,
    byKey            [2] KeyHash }
```

KeyHash ::= OCTET STRING -- SHA-1 hash of responder's public key
(excluding the tag and length fields)

```
SingleResponse ::= SEQUENCE {
    certID           CertID,
    certStatus       CertStatus,
```

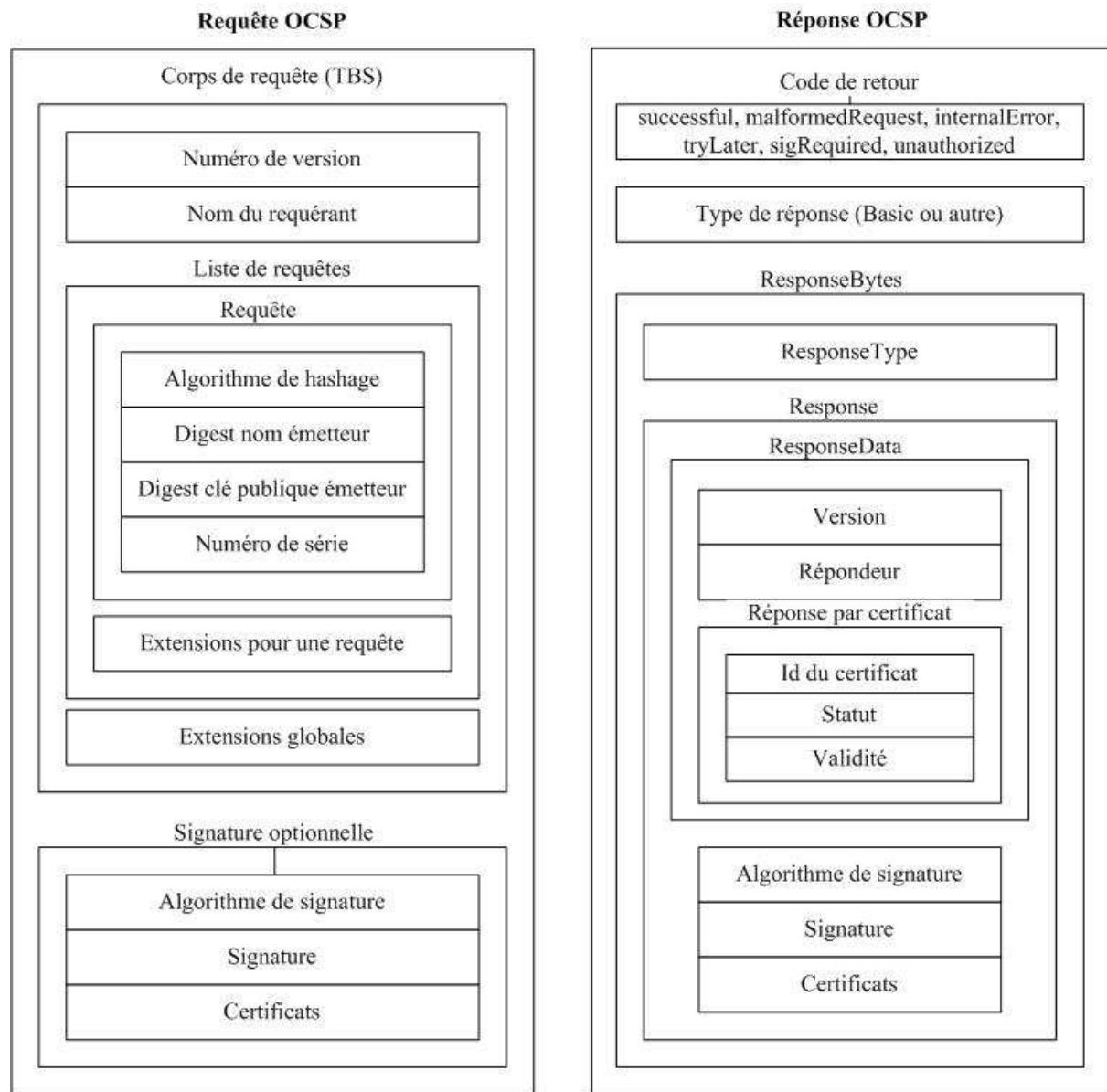
```
thisUpdate           GeneralizedTime,
nextUpdate [0]      EXPLICIT GeneralizedTime OPTIONAL,
singleExtensions [1] EXPLICIT Extensions OPTIONAL }
```

```
CertStatus ::= CHOICE {
    good   [0]  IMPLICIT NULL,
    revoked [1] IMPLICIT RevokedInfo,
    unknown [2] IMPLICIT UnknownInfo }
```

```
RevokedInfo ::= SEQUENCE {
    revocationTime      GeneralizedTime,
    revocationReason [0] EXPLICIT CRLReason OPTIONAL }
```

UnknownInfo ::= NULL -- this can be replaced with an enumeration

En résumé, on peut encore visualiser le protocole comme ceci :



18.2 Les classes OCSP de Bouncy Castle

La librairie Bouncy Castle fournit un package dédié à OCSP (org.bouncycastle.ocsp).

1) La classe de base est **OCSPReqGenerator** qui, comme son nom l'indique, est capable de générer une requête OCSP valide. Instanciée au moyen de son constructeur par défaut, elle est initialisée au moyen de la méthode :

```
public void addRequest(CertificateID certId)
```

puis génère la requête au moyen de sa méthode

```
public OCSPReq generate() throws OCSPException
```

2) Un objet instance de **CertificateID** a pour rôle d'identifier le certificat qui fait l'objet de la requête : il comporte donc le numéro de série de ce certificat avec le certificat de son émetteur ainsi que l'indication de l'algorithme de signature utilisé – c'est que confirme son constructeur d'initialisation :

```
public CertificateID(String hashAlgorithm, X509Certificate issuerCert,  
                     BigInteger serialNumber) throws OCSPException
```

3) La classe **OCSPReq** encapsule tous les champs prévus par la RFC 2650 et sa méthode

```
public byte[] getEncoded() throws IOException
```

en fournit d'ailleurs une représentation ASN.1.

4) Mais surtout, nous pouvons construire la réponse à partir des bytes lus en réponse dans le tunnel HTTP au moyen du constructeur :

```
public OCSPResp(java.io.InputStream in) throws IOException
```

Nous pourrons extraire l'objet réponse par :

```
public Object getResponseObject() throws OCSPException
```

5) Bien sûr, à priori, la réponse peut être de nature (presque) quelconque, mais nous supposerons qu'il s'agit d'une réponse de type "OCSP Basic", ce qui est matérialisé par un casting en BasicOCSPResp. Nous extrairons les réponses pour chacun des certificats sous la forme d'objets **SingleResp** au moyen de la méthode

```
public SingleResp[] getResponses()
```

Enfin, pour chacun d'entre eux, nous obtiendrons la réponse attendue par

```
public Object getCertStatus()
```

18.3 La vérification des certificats d'un keystore PKCS#12

Nous supposons donc disposer un keystore de format PKCS12 qui contient les divers certificats issus de la carte d'identité (disons qu'il se nomme KeystoreCarteIdentite). A l'image d'un utilisateur Web, nous allons confier notre requête à une trame HTTP (mais ce pourrait être un autre protocole comme LDAP) : celle-ci a de particulier qu'elle utilise la méthode POST avec un champ MIME valant "**application/ocsp-request**" et une URL de destination qui est "http://ocsp.eid.belgium.be/". En fait, nous allons utiliser plus précisément un tunnel **HTTP** par lequel nous pourrons récupérer les bytes qui vont constituer une instance de la classe OCSPResp dont l'un des constructeurs :

```
public java.lang.Object getresponseObject() throws OCSPException
```

n'attend manifestement que le tableau de bytes-mémoire que nous auront récupéré. Vu le paragraphe précédent, le reste du code se lit sans peine :

VerifCertificatSignCI.java

```
package carteidentite;

import java.io.*;
import java.net.*;
import java.security.*;
import java.security.cert.*;
import java.util.*;
import org.bouncycastle.ocsp.*;

/**
 * @author Vilvens
 */

public class VerifCertificatSignCI
{
    static public String OCSPServURL = "http://ocsp.eid.belgium.be/";
    static private String nomsFichierKeystore = "KeystoreCarteIdentite";
    static private String[] alias = {"BelgiumRootCert", "CitizenCert",
        "VilvensSignatureCert"};
    public static void main(String[] args)
    {
        // 1.Chargement du keystore
        System.out.println("*** Chargement du keystore en mémoire ***");
        String passwd = "pwdpwd";
        KeyStore KSP12 = null;
        String file = System.getProperty("user.dir") + System.getProperty("file.separator") +
            "KeystoreCarteIdentite";
        try
        {
            KSP12 = KeyStore.getInstance("PKCS12", "BC");
            KSP12.load(new FileInputStream(file), passwd.toCharArray());
        }
        catch(Exception e) { System.out.println("Aie Aie " + e.toString());System.exit(0); }
```

```

System.out.println(":-) Chargement du keystore effectué !");
System.out.println(">>> Contenu du keystore");
try
{
    for(Enumeration en = KSP12Aliases(); en.hasMoreElements());
        System.out.println((new StringBuilder()).append("Entry:").append(en.nextElement()).
            append(" ***").toString()));
}
catch(Exception e)
{
    System.out.println("Aie Aie " + e.toString());System.exit(0); }

//2. Construction et envoi de la requête OCSP
HttpURLConnection con = null;
try
{
    System.out.println("/** Envoyer une requête OCSP **");
    X509Certificate certAVerifier = (X509Certificate)KSP12.getCertificate(
        "VilvensSignatureCert");
    X509Certificate certAutorite = (X509Certificate)KSP12.getCertificate(
        "CitizenCert");
    System.out.println(":-) Certificats récupérés");
    CertificateID cid = new CertificateID(CertificateID.HASH_SHA1, certAutorite,
        certAVerifier.getSerialNumber());
    System.out.println(":-) Id du certificat construit");
    System.out.println((new StringBuilder()).append("N° de série: ").
        append(certAVerifier.getSerialNumber().toString(16)).
        append(" ***").toString());
    OCSPReqGenerator gen = new OCSPReqGenerator();
    gen.addRequest(cid);
    OCSPReq req = gen.generate();
    URL url = new URL(OCSPServURL);
    con = (HttpURLConnection)url.openConnection();
    con.setRequestMethod("POST");
    con.setDoOutput(true);
    con.setRequestProperty("Content-Type", "application/ocsp-request");
    OutputStream os = null;
    os = con.getOutputStream();
    os.write(req.getEncoded());
    os.close();
    System.out.println(":-) Requête envoyée");
}
catch(Exception ex)
{
    System.err.println((new StringBuilder()).append("Erreur? ").
        append(ex.toString().toString()));System.exit(0);
}

//3. Réception de la réponse
System.out.println("/** Réception de la réponse OCSP **");
ByteArrayOutputStream baos = null;

```

```

try
{
    baos = new ByteArrayOutputStream();
    InputStream in = con.getInputStream();
    for(int b = in.read(); b != -1; b = in.read()) baos.write(b);
    baos.flush();
    in.close();
    con.disconnect();
}
catch(Exception ex)
{
    System.err.println((new StringBuilder()).append("Erreur !!! ").
        append(ex.toString().toString()));System.exit(0);
}
System.out.println(":-) Réponse reçue !");

//4. Interprétation de la réponse
System.out.println("/** Conclusion de la réponse OCSP **");
byte[] respBytes = baos.toByteArray();
try
{
    OCSPResp response = new OCSPResp(new ByteArrayInputStream(respBytes));
    BasicOCSPResp brep = (BasicOCSPResp)response.getresponseObject();
    SingleResp singleResps[] = brep.getResponses();
    for(int i = 0; i < singleResps.length; i++)
    {
        SingleResp singleResp = singleResps[i];
        if(singleResp.getCertStatus() == null)
            System.out.println((new StringBuilder()).append(" +++ Certificat ").
                append(" :-) valide! ***").toString());
        else
            System.out.println((new StringBuilder()).append(" +++ Certificat ").
                append(" :-( NON valide !!! ***").toString());
    }
}
catch(Exception ex)
{
    System.err.println((new StringBuilder()).append("Erreur !!! ").
        append(ex.toString().toString()));
}
}
}

```

Un exemple d'exécution pourrait être :

```

*** Chargement du keystore en mémoire ***
:-) Chargement du keystore effectué !
>>> Contenu du keystore
Entry: BelgiumRootCert ***
Entry: VilvensSignatureCert ***

```

Entry: CitizenCert ***

*** Envoi d'une requête OCSP ***

:-) Certificats récupérés

:-) Id du certificat construit

N° de série: 100000000000b9dab5177df45c4199a5 ***

:-) Requête envoyée

*** Réception de la réponse OCSP ***

:-) Réponse reçue !

*** Conclusion de la réponse OCSP ***

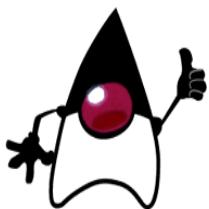
+++ Certificat :- valide! ***

Si on observe le trafic réseau avec un sniffer, on peut visualiser les trames OCSP échangées - d'abord la requête :

					OCSP	Request
5	0.035958	192.168.1.6	64.18.25.27		TCP	http > 2564 [ACK]
6	0.061408	64.18.25.27	192.168.1.6		TCP	[TCP segment of a
7	0.140889	64.18.25.27	192.168.1.6		TCP	[TCP segment of a
8	0.160528	64.18.25.27	192.168.1.6		TCP	[TCP segment of a
9	0.160571	192.168.1.6	64.18.25.27		TCP	2564 > http [ACK]
10	0.190606	64.18.25.27	192.168.1.6		TCP	[TCP segment of a
11	0.197780	64.18.25.27	192.168.1.6		OCSP	Response [Malformed]
12	0.197822	192.168.1.6	64.18.25.27		TCP	2564 > http [ACK]
13	0.204853	192.168.1.6	64.18.25.27		TCP	2564 > http [FIN]
+ Transmission Control Protocol, Src Port: 2564 (2564), Dst Port: http (80), seq:						
+ [Reassembled TCP Segments (302 bytes): #4(219), #5(83)]						
+ Hypertext Transfer Protocol						
+ POST / HTTP/1.1\r\n						
Connection: keep-alive\r\n						
User-Agent: Java/1.6.0_13\r\n						
Host: ocsp.eid.belgium.be\r\n						
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2\r\n						
Content-Length: 83\r\n						
Content-Type: application/ocsp-request\r\n						
\r\n						
+ Online Certificate Status Protocol						
+ tbsRequest						
+ requestList: 1 item						
+ Item						
+ reqCert						
+ hashAlgorithm (SHA-1)						
+ issuerNameHash: 35D7CFFF66B46868784CEB945655E2BE830ECD41						
+ issuerKeyHash: 79DA9134B34327CE76955E47088D9AD906527299						
+ serialNumber : 0x100000000000b9dab5177df45c4199a5						

puis la réponse :

8 0.160528	64.18.25.27	192.168.1.6	TCP	[TCP segment of a
9 0.160571	192.168.1.6	64.18.25.27	TCP	2564 > http [ACK]
10 0.190606	64.18.25.27	192.168.1.6	TCP	[TCP segment of a
+ Ethernet II, Src: D-Link_1c:ac:ed (00:17:9a:1c:ac:ed), Dst: 00:1d:7d:7a:f2:a1 (00:1d:7d:7a:f2:a1)				
+ Internet Protocol, Src: 64.18.25.27 (64.18.25.27), Dst: 192.168.1.6 (192.168.1.6)				
+ Transmission Control Protocol, Src Port: http (80), Dst Port: 2564 (2564), seq: 1, ack: 1, length: 1024				
+ Hypertext Transfer Protocol				
- Online Certificate Status Protocol				
responseStatus: successful (0)				
- responseBytes				
ResponseType Id: 1.3.6.1.5.5.7.48.1.1 (id-pkix-ocsp-basic)				
- BasicOCSPResponse				
tbsResponseData				
responderID: byName (1)				
producedAt: 2009-06-01 12:34:56 (Z)				
responses: 1 item				
Item				
certID				
hashAlgorithm (SHA-1)				
issuerNameHash: 35D7CFFF66B46868784CEB945655E2BE830ECD41				
issuerKeyHash: 79DA9134B34327CE76955E47088D9AD906527299				
serialNumber : 0x100000000000b9dab5177df45c4199a5				
certStatus: good (0)				
good				
thisUpdate: 2009-06-01 12:34:56 (Z)				
nextUpdate: 2009-06-01 12:35:56 (Z)				
responseExtensions: 1 item				



A suivre ...

Avec de tels outils, la sécurité semble être à la portée de toutes les applications professionnelles. Mais comment ces applications sont-elles gérées et maintenues ? Comment manipulent-elles et donnent-elles accès de manière adéquate à ce qui constitue leur raison d'être : les données ? Eléments de réponse dans Java VI !

Ouvrages consultés

Ouvrages imprimés et électroniques

Boulanger, M. Etude de la plate-forme mobile Android et développement d'un prototype d'application cartographique dédiée aux activités en extérieur. TFE Haute Ecole de la Province de Liège (In.Pr.E.S.). 2010.

Bouzefrane, S. & Cordry, J. Interaction avec la carte à puce selon les protocoles ISO 7816-3 et ISO 7816-4. Conservatoire National des Arts et Métiers (Master "systèmes embarqués et mobiles" [SEM]). 2008.

Chen, Z. Java Card Technology for Smart Cards. The Java Series. Reading, Massachusetts, U.S.A. Addison-Wesley Publishing Company. 2000.

Geiebienne, O. Windows et la carte d'identité électronique : signature et cryptage en C#. TFE Haute Ecole de la Province de Liège (In.Pr.E.S.). 2006.

Hassler, V., Manning, M., Gordeev, M. & Müller, C. Java Card for E-Payment Applications. Norwood, Massachusetts, U.S.A. Artech House, Inc. 2002.

Ippolito, S. Etude des fonctionnalités d'authentification et de signature de la carte d'identité électronique belge et application à un contexte de protocoles médicaux. TFE Haute Ecole de la Province de Liège (I.S.I.L.). 2002.

Jeunesse, F. Etude synthétique de techniques de sécurité utilisées par les applications réseaux. Seraing, Belgique. TFE Haute Ecole de la Province de Liège (In.Pr.E.S.). 2009.

Lekenne, N. Programmation et utilisation des Java Cards pour le commerce électronique. TFE In.Pr.E.S. 2002.

Licata, J. Etude et conception d'une architecture pour le commerce électronique. TFE Université de Liège. 2004.

Mamay, P. & Germian, T. Gestion informatisée de la traçabilité des denrées alimentaires utilisées par un restaurant de collectivité. TFE Haute Ecole de la Province de Liège (In.Pr.E.S.). 2011.

Micciché, R. Passerelles applicatives pour une interconnexion avec un environnement Web (lecteur de carte et mobile). TFE Haute Ecole de la Province de Liège (In.Pr.E.S.). 2011.

Nyström, M. PKCS #15 - A Cryptographic-Token Information Format Standard. Usenix Association. 1999.

Ponthieu, S. & Campello Fuentes, M. Conception et développement d'un site Web à vocation culturelle avec étude et intégration des technologies mobiles et des cartes à puces. TFE Haute Ecole Rennequin Sualem (In.Pr.E.S.). 2006.

Schembri, G. Etude de faisabilité de l'intégration d'applications cryptographiques au sein d'une carte à puce de type AVR. TFE Haute Ecole Rennequin Sualem (I.S.I.L.). 2004.

Stern, M. & Bogaert, K. Belgian eID Toolkit Developer's guide. CSC Computer Sciences. 2006 (document électronique).

Verkenne, M. Etude de la technologie Google Android et utilisation dans une application de gestion de notes de cours. TFE Haute Ecole de la Province de Liège (In.Pr.E.S.). 2010.

Vilvens, C. Langage Java (I) : Programmation de base. Seraing, Belgique. A.S.B.L. DEFI. 2010.

Vilvens, C. Langage Java (II) : Programmation avancée des applications classiques. Seraing, Belgique. A.S.B.L. DEFI. 2010.

Vilvens, C. Langage Java (IV) : Programmation de protocoles applicatifs et de techniques de sécurité. Seraing, Belgique. A.S.B.L. DEFI. 2010.

et

RSA. PKCS-11 Protocol for Enterprise Key Management. 2007 (document électronique).

SPF Technologie de l'Information et de la Communication (Fedict). La carte d'identité électronique et Firefox. 2006 (document électronique).

SPF Technologie de l'Information et de la Communication (Fedict). La carte d'identité électronique et Thunderbird Courriel. 2006 (document électronique).

Sites Internet

(août 2011)

<http://java.sun.com/>

avec en particulier :

<http://developer.java.sun.com/>

<http://developer.java.sun.com/developer/onlineTraining>

<http://developers.sun.com/mobility/midp/articles/deploy/>

<http://www.bejug.org/>

Site du Belgian Java User Group

<http://developer.android.com/reference/packages.html>

Références des classes Android

<http://www.openhandsetalliance.com>

Site de l'Open Handset Alliance

<http://developer.android.com/index.html>

Site des développeurs Android

<http://www.ibm.com/developerworksopensource/library/os-android-devel>

Site IBM pour Android



<http://www.datelec.fr/fiches/Les%20cartes%20a%20puces%20asynchrones.htm>

Spécifications techniques des cartes à puces en France

<http://fr.tech-faq.com/iso-7816.shtml>

Qu'est-ce que l'ISO 7816 ?

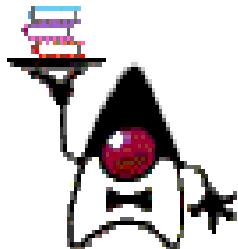
<http://www.pcscworkgroup.com/>

Le site officiel de PC/SC.

http://www.belgium.be/zip/eid_datacapture_fr.html

Pour télécharger le run-time de la carte d'identité belge

Annexe : Introduction au développement sous Eclipse



Les lois inutiles affaiblissent les lois nécessaires.

(Montesquieu, De l'esprit des lois)

1. La fondation Eclipse et sa communauté

Pour le développeur Java, Eclipse est l'EDI [Environnement de Développement Intégré] concurrent open-source de NetBeans, EDI nominal de Sun.

En fait, le projet Eclipse a été créé en 2001 par IBM qui a fait don du code initial et fondé un consortium au sein duquel on trouvait notamment Borland, société bien connue dans le domaine des EDIs. En 2004, le consortium Eclipse est ensuite devenu une **fondation** (<http://www.eclipse.org/>), permettant ainsi l'entrée de nouveaux partenaires. On y trouve ainsi dès 2006, outre IBM et Borland, des sociétés comme BEA et Oracle ... mais pas Sun, qui développait (et développe toujours) son propre EDI NetBeans. Le nom de la fondation était de plus ressenti par Sun comme une provocation ;-) ... A l'heure actuelle, on distingue les membres "Strategic" (comme Nokia ou Oracle), "Add-in providers" (comme Adobe, Ericsson, Intel, Sybase, Spring aussi et ... Google) et "Associate" (comme Atos Origin, Nec ou Siemens. A côté de la fondation se place la communauté Eclipse (<http://www.eclipse.org/community/>), qui regroupe des milliers de contributeurs :

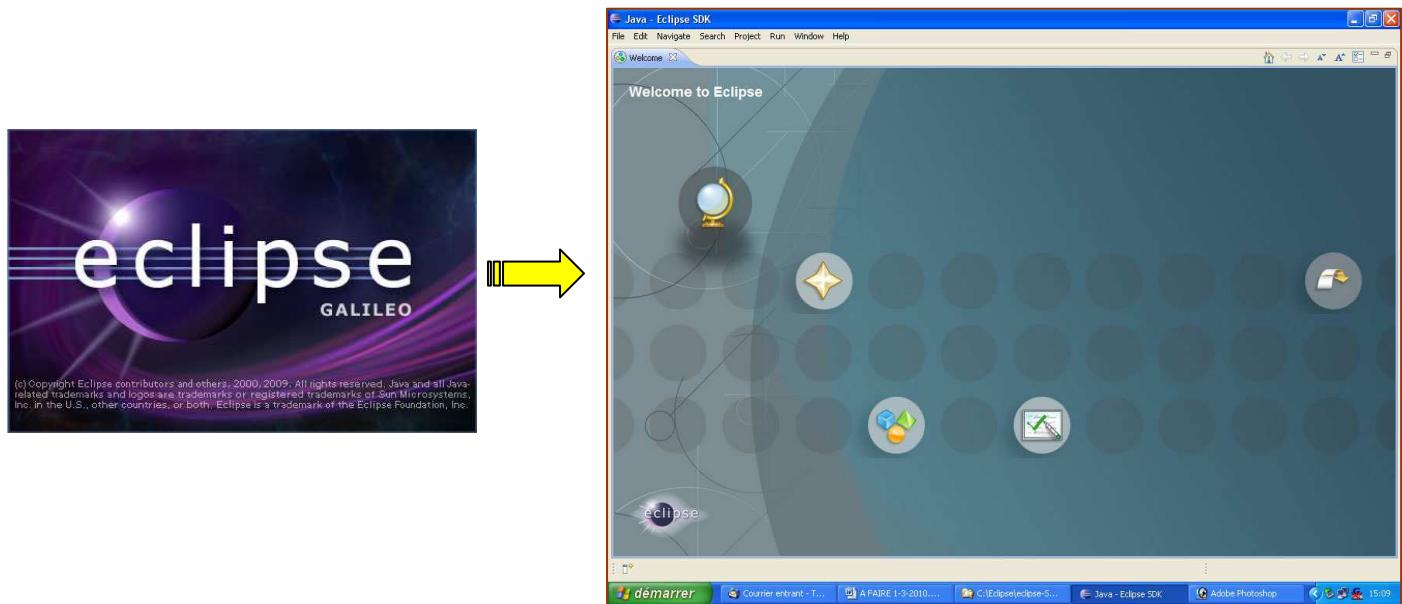


The screenshot shows the Eclipse Community Page. The top navigation bar includes links for HOME, DOWNLOADS, USERS, MEMBERS, COMMITTERS, RESOURCES, PROJECTS, and ABOUT US. There are also links for CONTACT, LEGAL, and a search bar. The main content area is titled "Welcome to the Eclipse Community Page". It features several sections with icons and descriptions:

- Events**: Shows a calendar icon with the number 31, indicating events like conferences and gatherings.
- Portals**: Shows a globe icon, describing community sites and blogs.
- Training**: Shows a presentation icon, listing Books, Online Courses, Webinars, and Tutorials.
- Consulting & Services**: Shows a yellow shield icon, listing Consulting, development, and training services.
- Evangelism**: Shows a speaker icon, listing what the Eclipse Evangelist Wayne Beaton is talking about.
- Plugins**: Shows a blue folder icon, listing many Eclipse-based plugins.
- Project Resources**: Shows an envelope icon, listing Mailing Lists, Newsgroups, and Project Wikis.
- RCP Catalog**: Shows a blue document icon, listing Open Source and Commercial Rich Client Platform applications.

La fondation développe différents projets, mais c'est bien entendu le projet Eclipse visant à développer un socle de développement Java qui nous intéresse ici. On peut télécharger la version la plus adaptée à ses développements sur <http://www.eclipse.org/downloads/>. Une fois obtenu le fichier `eclipse-SDK-3.5.1-win32.zip`

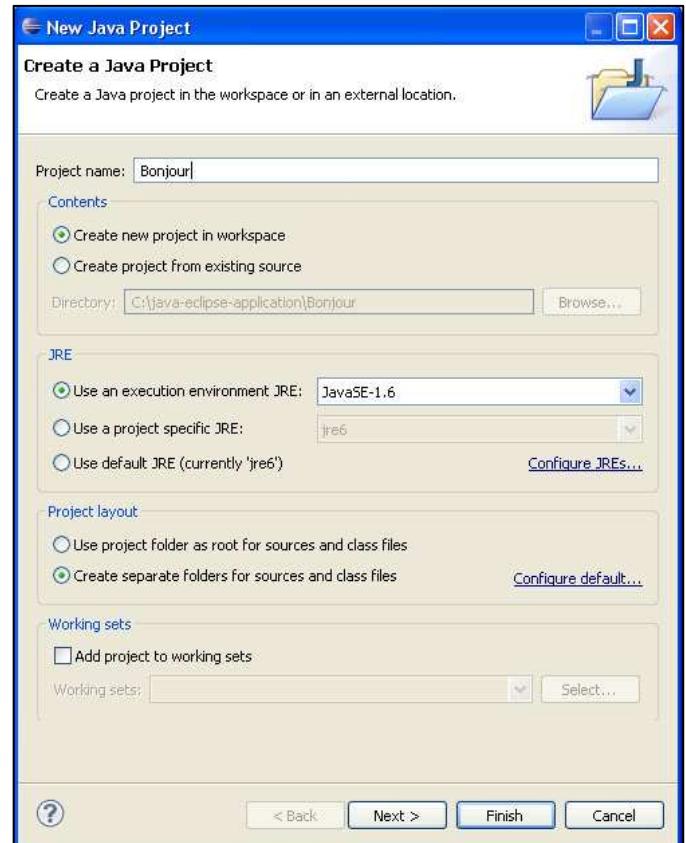
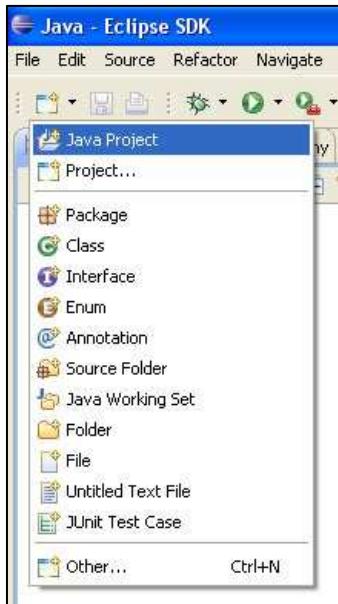
(dans le cas de Windows), il suffit de décompresser celui-ci pour disposer d'un exécutable eclipse qui peut faire démarrer l'EDI (il n'y a donc pas de procédure d'installation habituelle) :

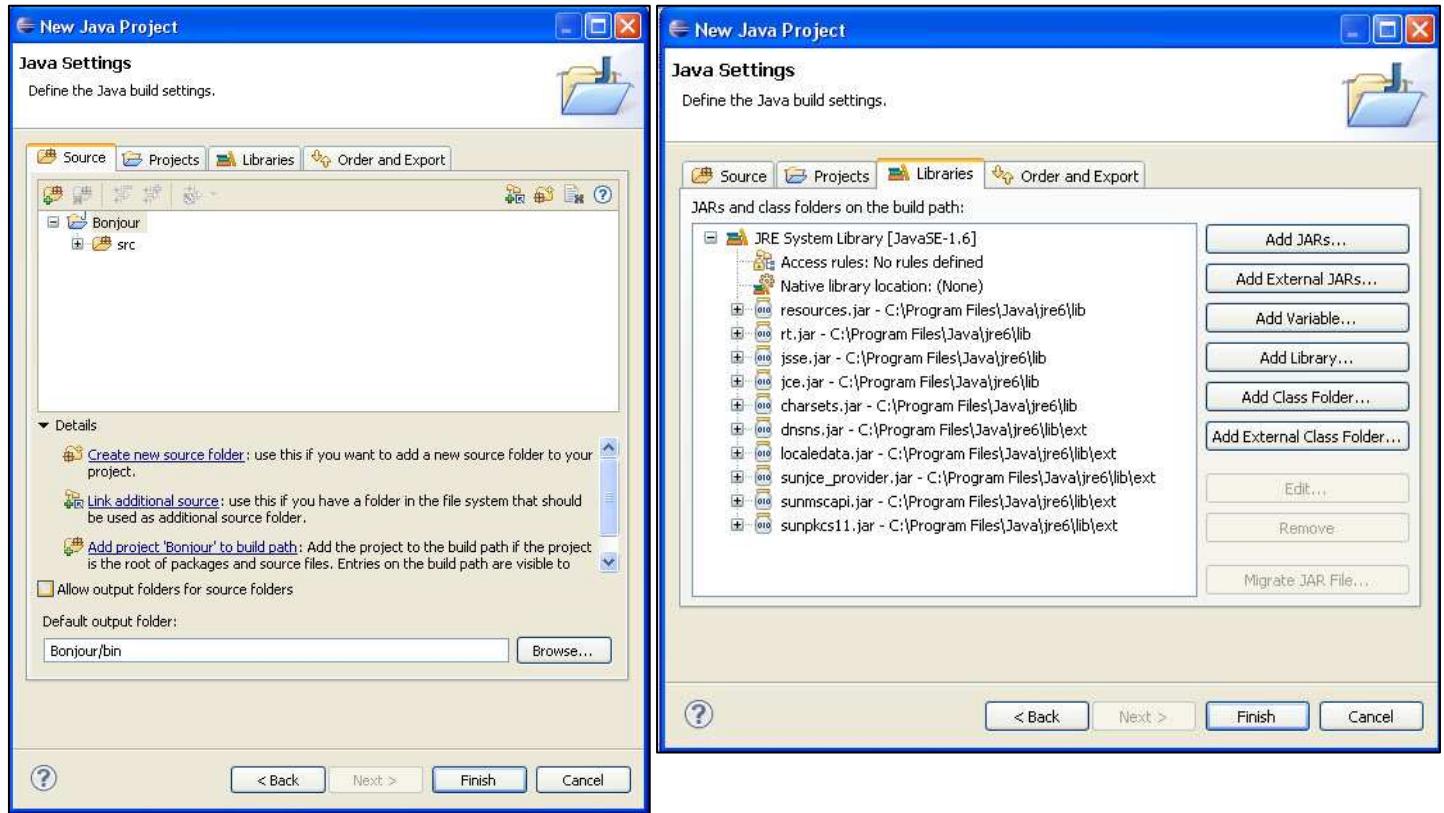


2. Développement d'une application console

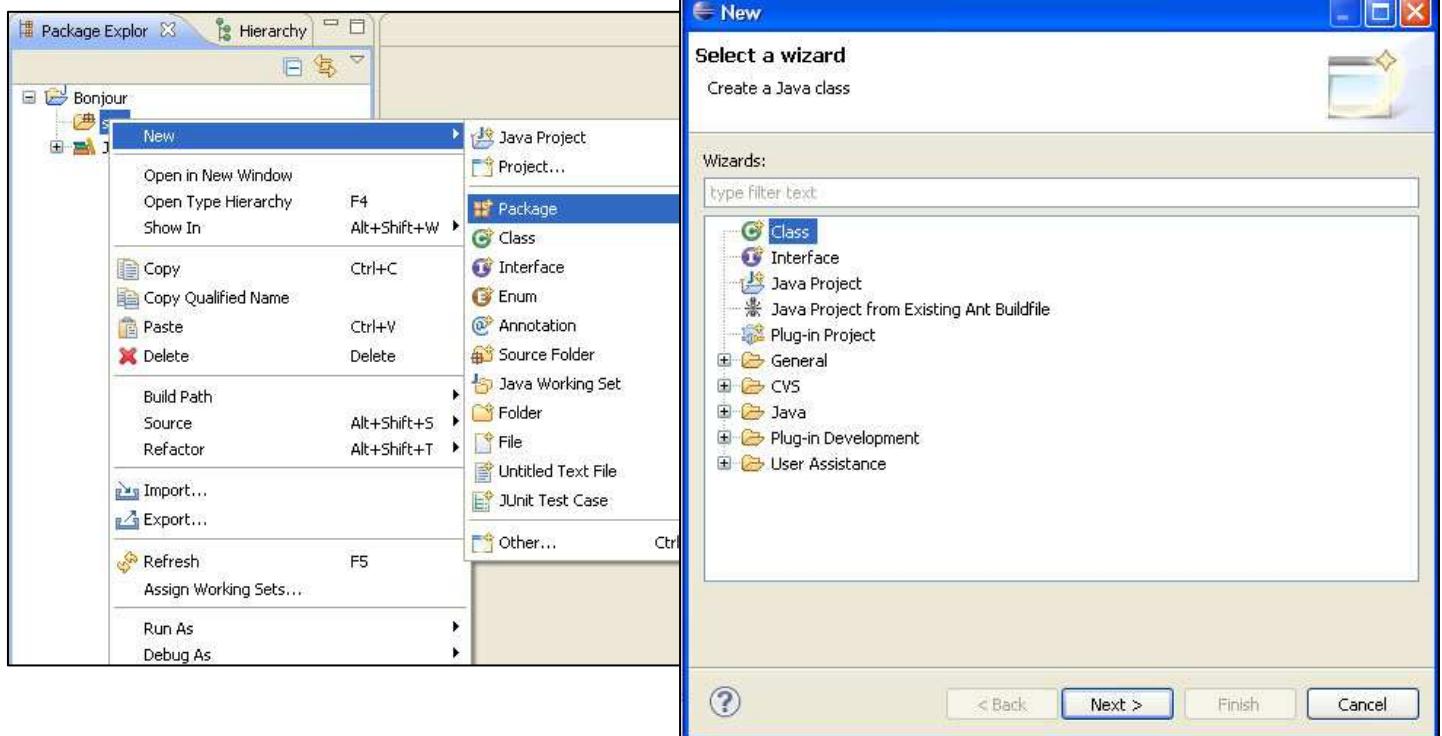
Pour un développeur chevronné, développer une application console est élémentaire :

1) création d'un projet :

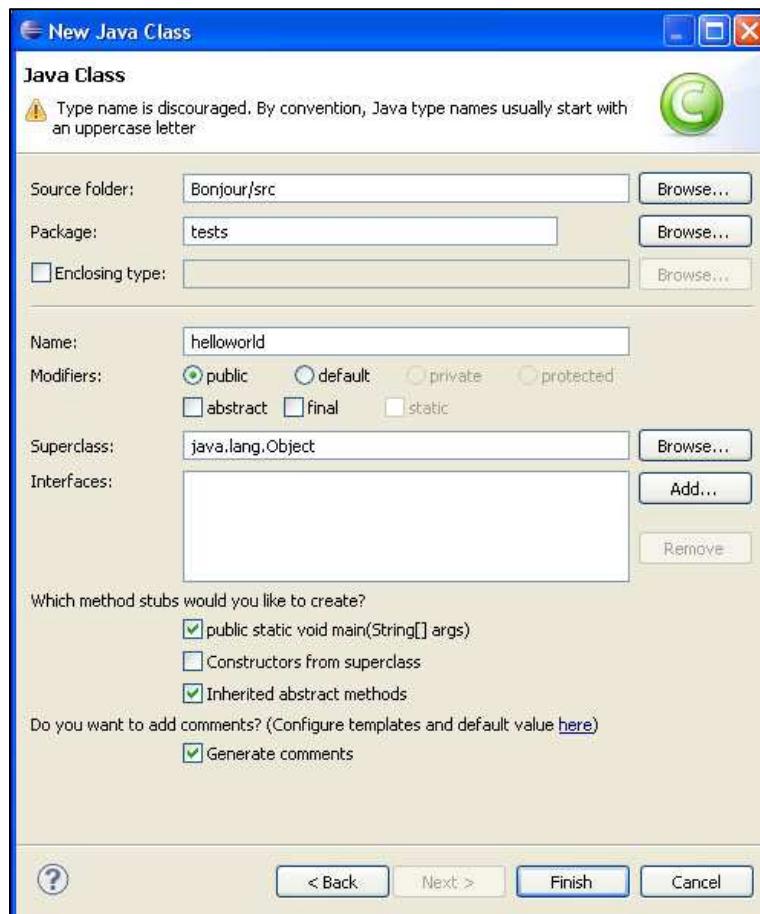




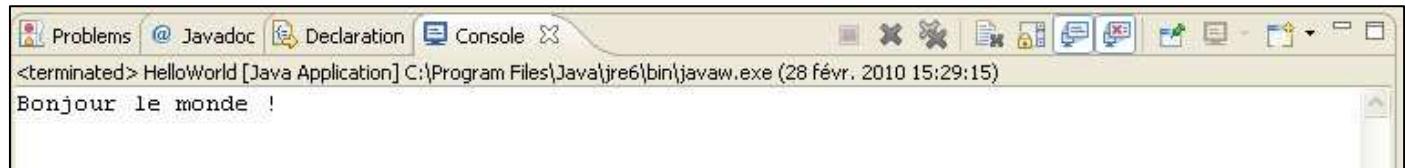
2) nouvelle classe :



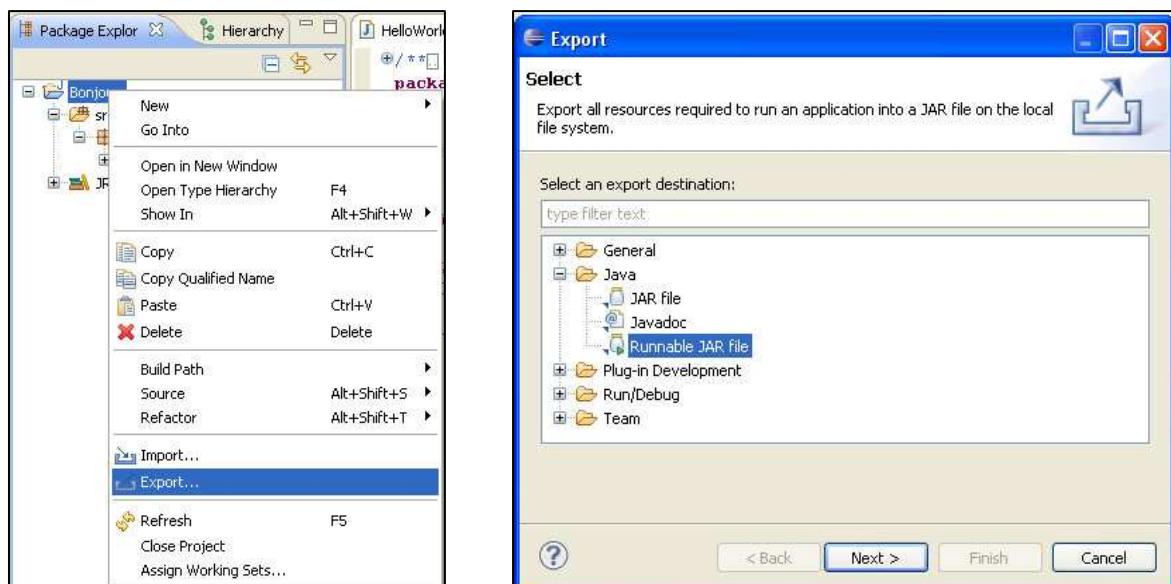
avec paramétrage :

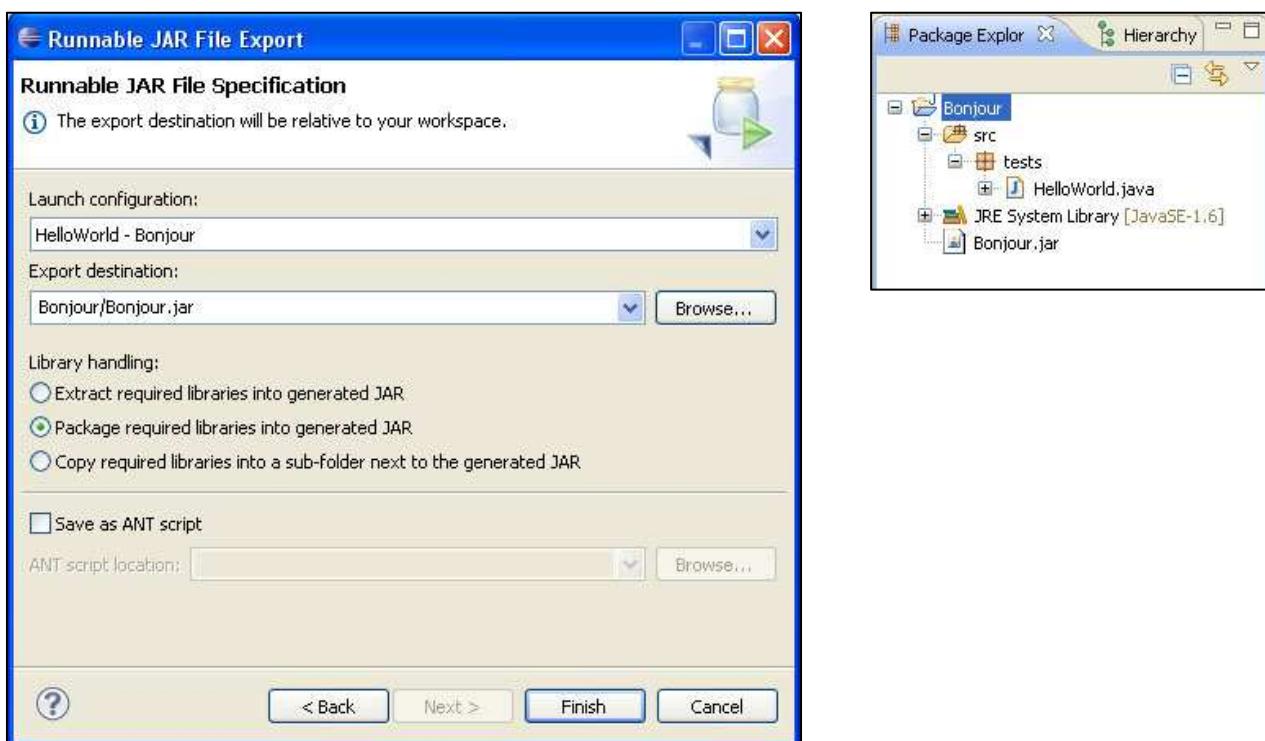


3) édition, compilation et exécution :



4) empaquetage en jar : un jar est construit en tant que résultat d'une exportation du projet :





Sur la console :

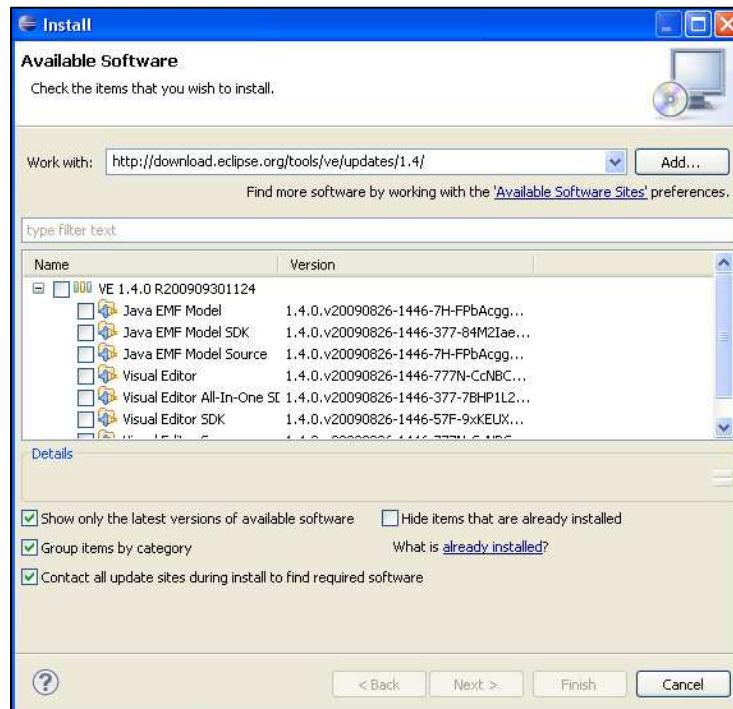
```
C:\java-eclipse-application\Bonjour>java -jar Bonjour.jar
Bonjour le monde !
```

3. Installation d'un éditeur visuel pour applications GUI

Pour une application GUI, il n'y a pas d'outil installé par défaut pour la conception d'un interface graphique :-o (liberté totale ;-)!). On peut cependant installer différents **plugins** pour remédier à cette absence en choisissant dans le menu

Help → Install New Software

et en désignant le site <http://download.eclipse.org/tools/ve/updates/1.4/> :



On peut cocher ce que l'on désire (ici, c'est surtout le Visual Editor, mais nous prenons tout, y compris les librairies EMF [Eclipse Modeling Framework]) – après téléchargement, nous obtenons :

Install Details

Review the items to be installed.

Name	Version	Id
Java EMF Model	1.4.0....	org.eclipse.jem.feature.group
Java EMF Model SDK	1.4.0....	org.eclipse.jem.sdk.feature.group
Java EMF Model Source	1.4.0....	org.eclipse.jem.source.feature.group
Visual Editor	1.4.0....	
Visual Editor All-In-One SDK	1.4.0....	
Visual Editor SDK	1.4.0....	
Visual Editor Source	1.4.0....	

Review Licenses

Licenses must be reviewed and accepted before the software can be installed.

Items with licenses:

Name	Version
Java EMF Model	1.4.0.v20090826-1446-7H-
Java EMF Model SDK	1.4.0.v20090826-1446-377
Java EMF Model Source	1.4.0.v20090826-1446-7H-
Visual Editor	1.4.0.v20090826-1446-777
Visual Editor All-In-One SDK	1.4.0.v20090826-1446-377
Visual Editor SDK	1.4.0.v20090826-1446-57F
Visual Editor Source	1.4.0.v20090826-1446-777

License text:

ECLIPSE FOUNDATION SOFTWARE USER AGREEMENT
March 17, 2005

Usage Of Content

THE ECLIPSE FOUNDATION MAKES AVAILABLE SOFTWARE, DOCUMENTATION, INFORMATION AND/OR OTHER MATERIALS FOR OPEN SOURCE PROJECTS (COLLECTIVELY "CONTENT"). USE OF THE CONTENT IS GOVERNED BY THE TERMS AND CONDITIONS OF THIS AGREEMENT AND/OR THE TERMS AND CONDITIONS OF LICENSE AGREEMENTS OR NOTICES INDICATED OR REFERENCED BELOW. BY USING THE CONTENT, YOU AGREE THAT YOUR USE OF THE CONTENT IS GOVERNED BY THIS AGREEMENT AND/OR THE TERMS AND CONDITIONS OF ANY APPLICABLE LICENSE AGREEMENTS

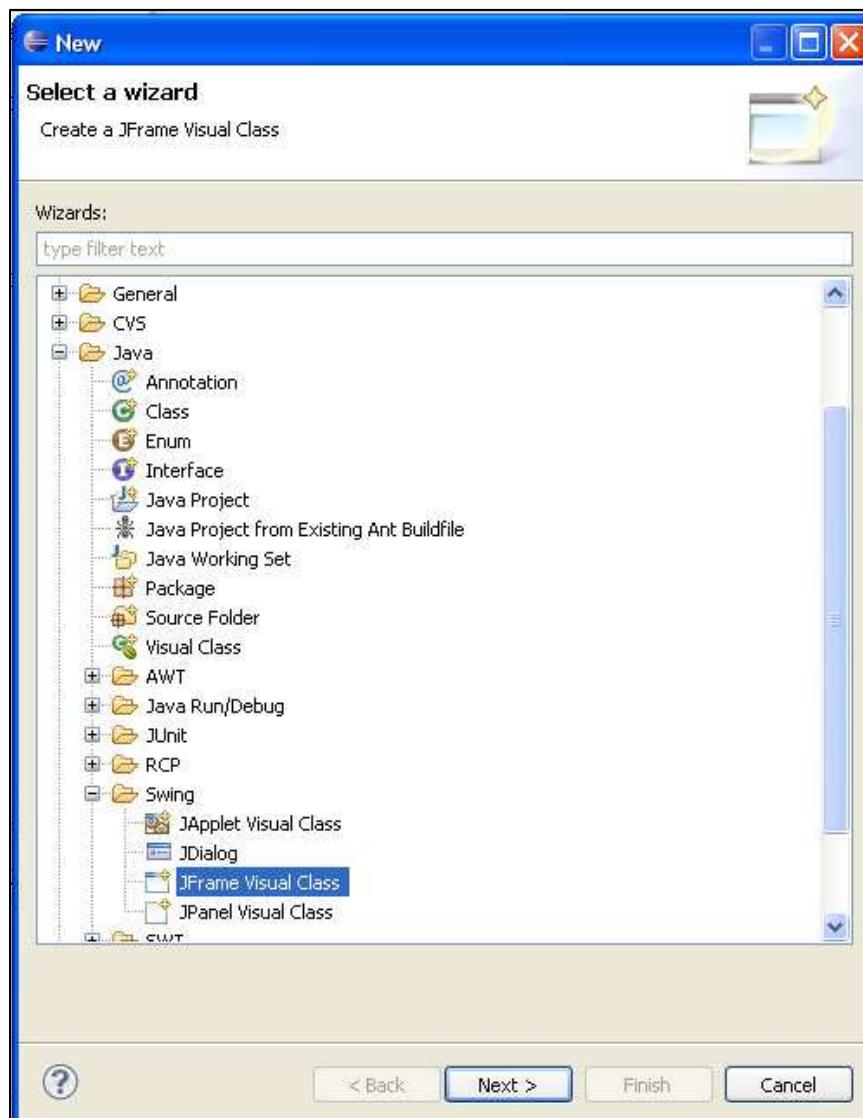
I accept the terms of the license agreements
 I do not accept the terms of the license agreements

L'installation proprement dite peut alors s'effectuer (ce n'est pas court !) et un redémarrage est demandé :

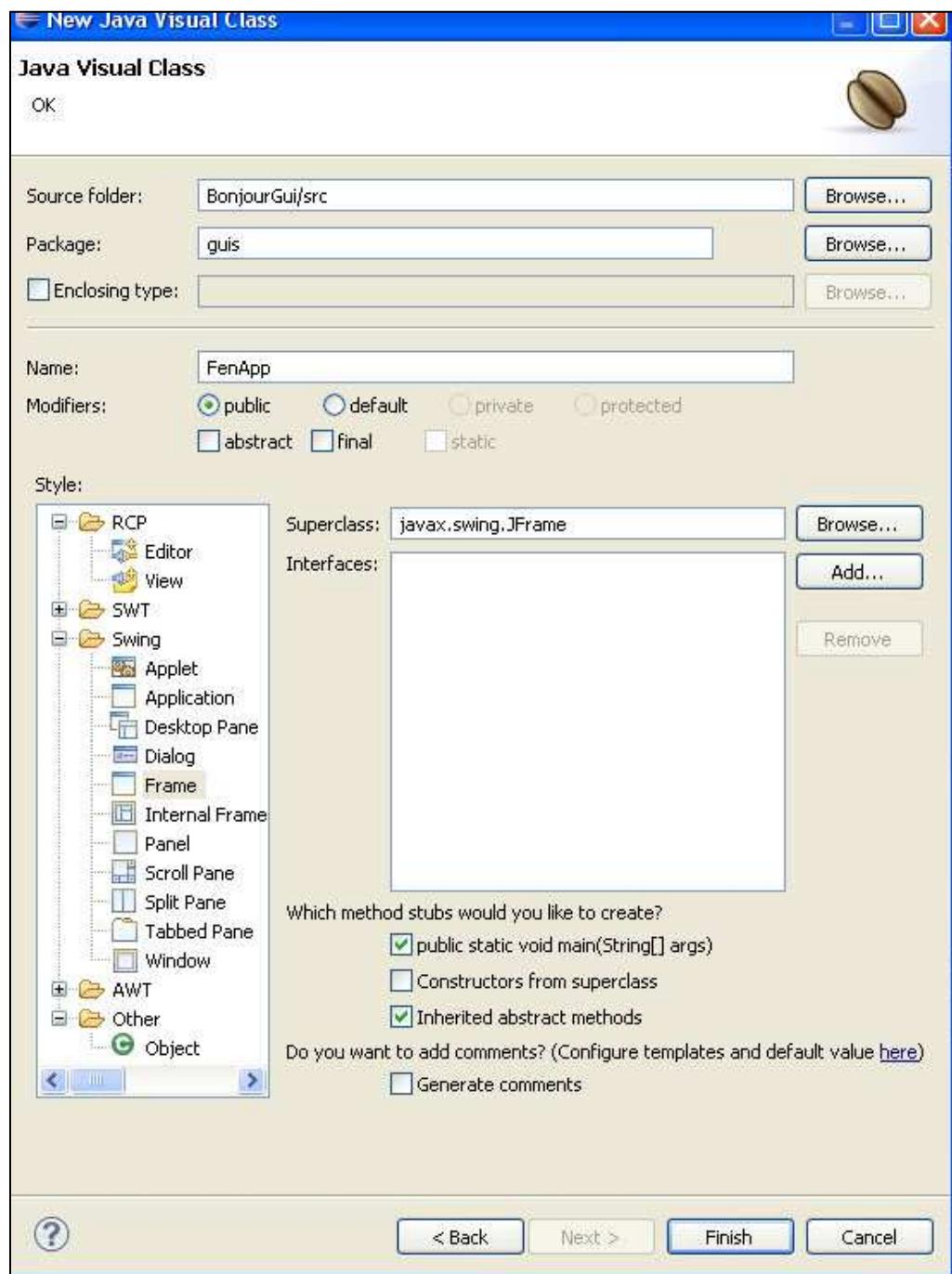


4. Développement d'une application GUI

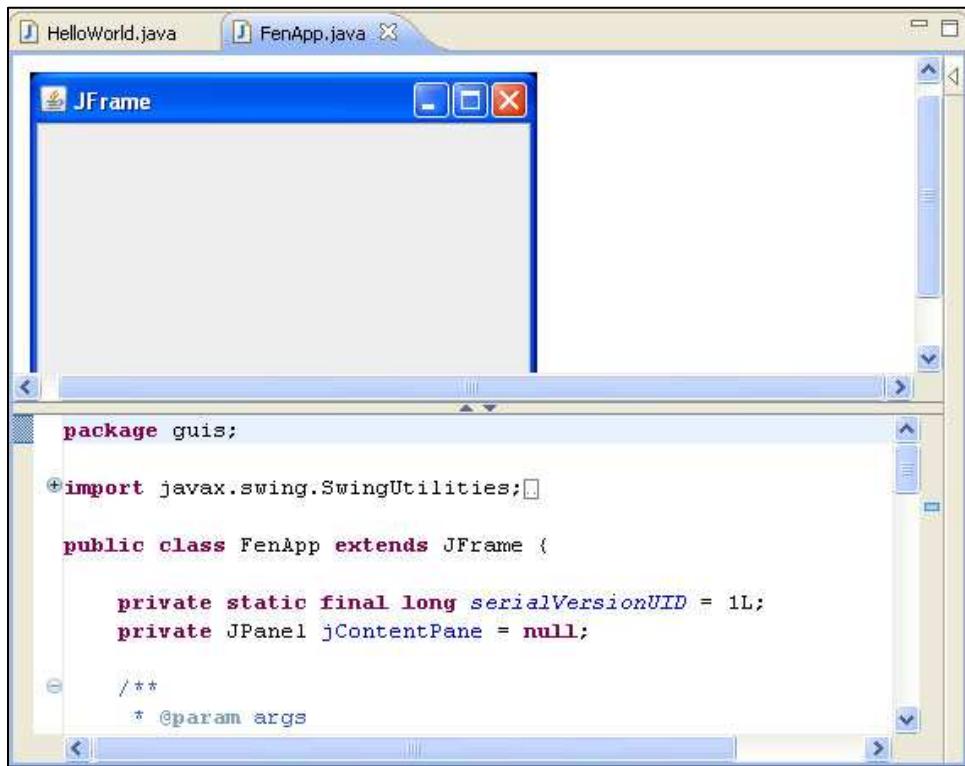
La sollicitation de l'icône New (ou de l'item du menu contextuel équivalent) est cette fois plus riche en propositions, avec la possibilité d'intégrer un composant Swing :



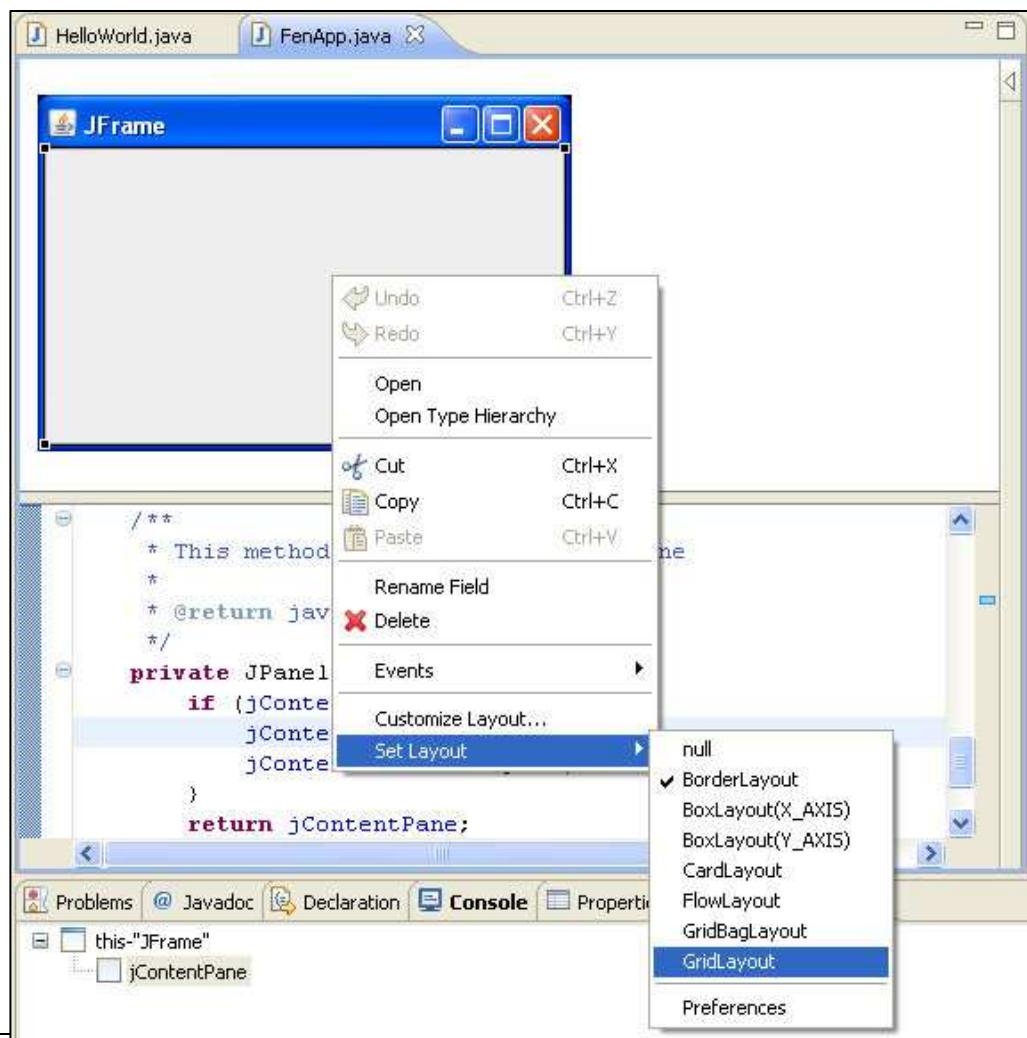
et la suite devient assez évidente :



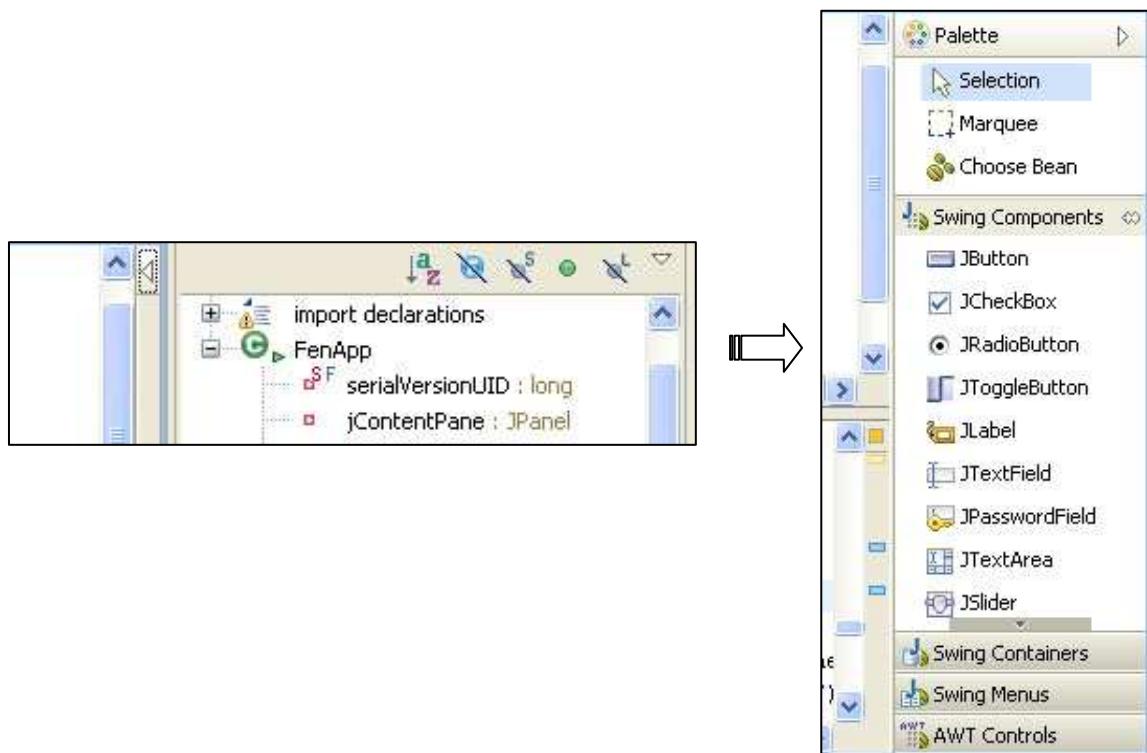
comme le résultat :



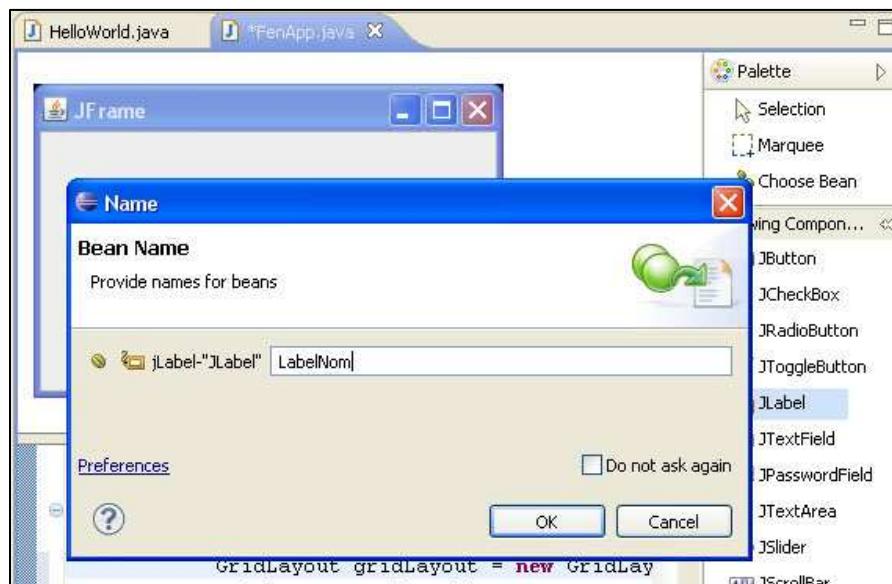
Un clic droit dans la fenêtre :



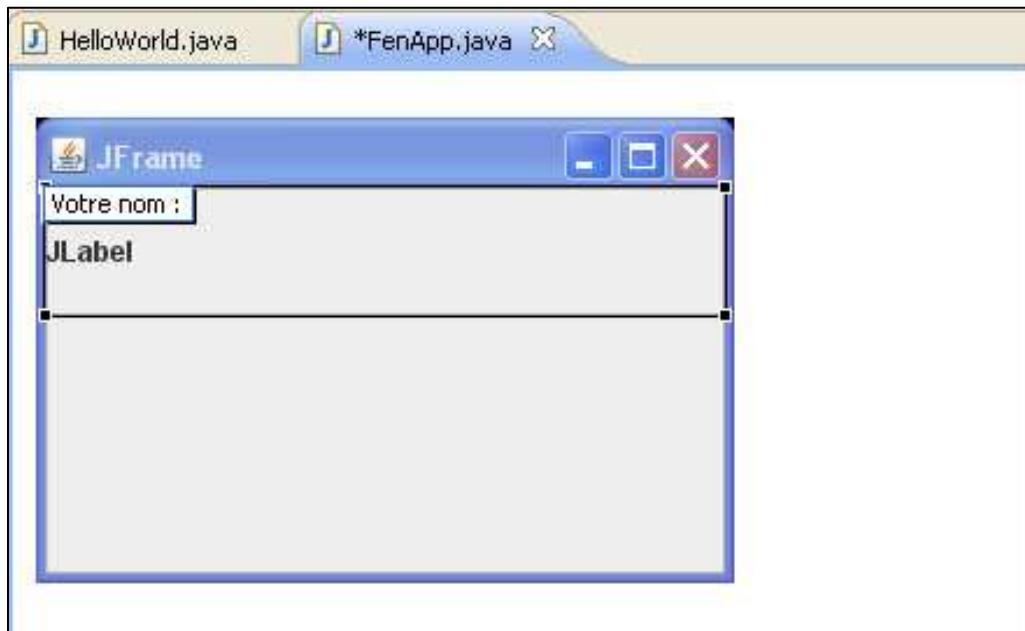
Un clic droit et le choix "Customize layout" permet de fixer le nombre de lignes et de colonnes dans le GridLayout choisi. Après, c'est la routine : on choisit des composants dans la palette, que l'on peut faire apparaître en sollicitant la flèche d'expansion :



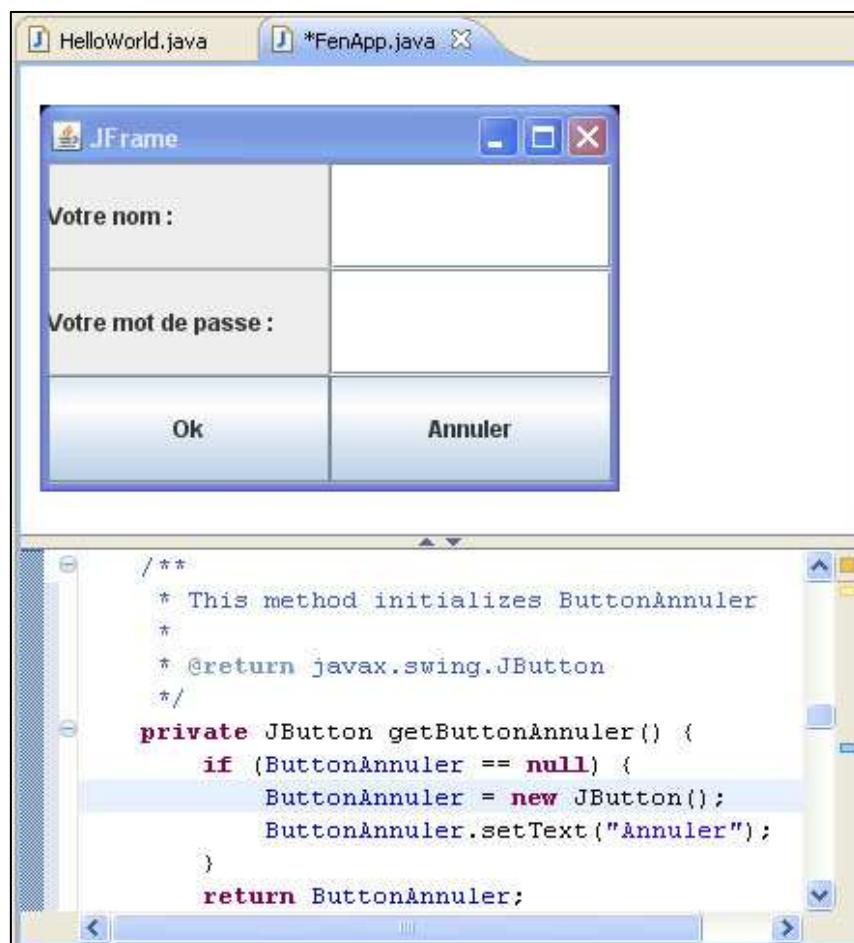
On y sélectionne un élément (par exemple un JLabel) et on clique dans la fenêtre :



Un clic droit dans le composant et le choix Set Text permet de fixer le texte :



ce qui permet d'arriver sans problème à :



On peut remarquer que le code généré utilise les ficelles de la P.O.O. :

FenApp.java

```
package guis;

import javax.swing.SwingUtilities;
import java.awt.BorderLayout;
import javax.swing.JPanel;
import javax.swing.JFrame;
import java.awt.GridLayout;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.JButton;

public class FenApp extends JFrame {

    private static final long serialVersionUID = 1L;
    private JPanel jContentPane = null;
    private JLabel LabelNom = null;
    private JTextField TextFieldNom = null;
    private JLabel LabelPwd = null;
    private JTextField TextFieldPwd = null;
    private JButton ButtonOk = null;
    private JButton ButtonAnnuler = null;
    /**
     * This method initializes TextFieldNom
     *
     * @return javax.swing.JTextField
     */
    private JTextField getTextFieldNom() {
        if (TextFieldNom == null) {
            TextFieldNom = new JTextField();
        }
        return TextFieldNom;
    }

    /**
     * This method initializes TextFieldPwd
     *
     * @return javax.swing.JTextField
     */
    private JTextField getTextFieldPwd() {
        if (TextFieldPwd == null) {
            TextFieldPwd = new JTextField();
        }
        return TextFieldPwd;
    }
}
```

```

/**
 * This method initializes ButtonOk
 *
 * @return javax.swing.JButton
 */
private JButton getButtonOk() {
    if (ButtonOk == null) {
        ButtonOk = new JButton();
        ButtonOk.setText("Ok");
        ButtonOk.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent e) {
                System.out.println("actionPerformed()");
                // TODO Auto-generated Event stub actionPerformed()
            }
        });
    }
    return ButtonOk;
}

/**
 * This method initializes ButtonAnnuler
 *
 * @return javax.swing.JButton
 */
private JButton getButtonAnnuler() {
    if (ButtonAnnuler == null) {
        ButtonAnnuler = new JButton();
        ButtonAnnuler.setText("Annuler");
    }
    return ButtonAnnuler;
}

/**
 * @param args
 */
public static void main(String[] args) {
    // TODO Auto-generated method stub
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            FenApp thisClass = new FenApp();

            thisClass.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            thisClass.setVisible(true);
        }
    });
}

/**
 * This is the default constructor
 */

```

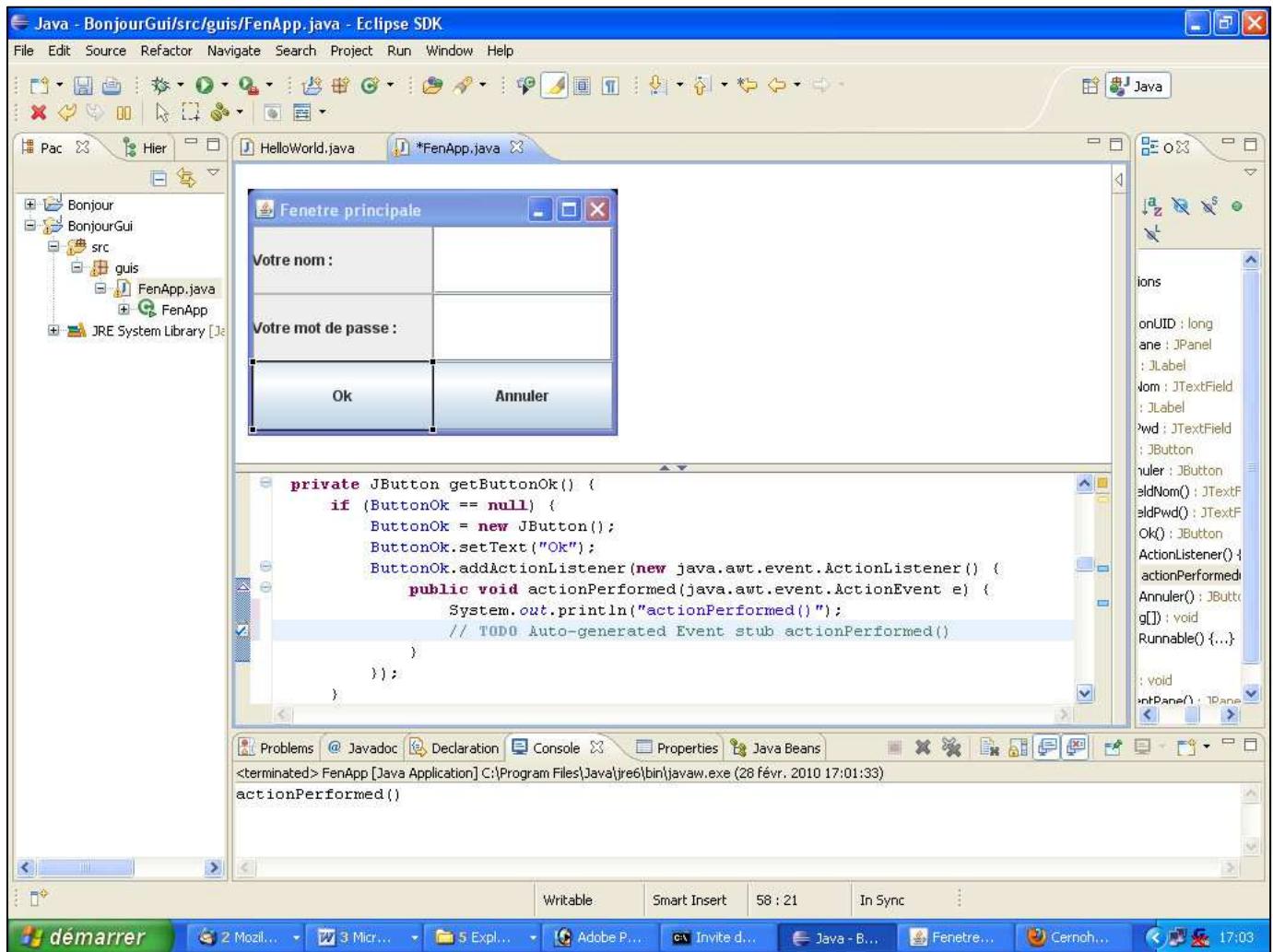
```
public FenApp() {
    super();
    initialize();
}

/**
 * This method initializes this
 *
 * @return void
 */
private void initialize() {
    this.setSize(300, 200);
    this.setContentPane(getJContentPane());
    this.setTitle("Fenetre principale");
}

/**
 * This method initializes jContentPane
 *
 * @return javax.swing.JPanel
 */
private JPanel getJContentPane() {
    if (jContentPane == null) {
        LabelPwd = new JLabel();
        LabelPwd.setText("Votre mot de passe :");
        LabelNom = new JLabel();
        LabelNom.setText("Votre nom :");
        GridLayout gridLayout = new GridLayout();
        gridLayout.setRows(3);
        gridLayout.setColumns(2);
        jContentPane = new JPanel();
        jContentPane.setLayout(gridLayout);
        jContentPane.add(LabelNom, null);
        jContentPane.add(getTextFieldNom(), null);
        jContentPane.add(LabelPwd, null);
        jContentPane.add(getTextFieldPwd(), null);
        jContentPane.add(getButtonOk(), null);
        jContentPane.add(getButtonAnnuler(), null);
    }
    return jContentPane;
}

}
```

On remarquera que la gestion de l'appui sur le bouton Ok a déjà été assurée :



De manière plus générale, on gère les événements par un clic droit sur le composant :

