# Outil d'installation d'applications sur une grille de cartes à puce de type Java à API respectant GlobalPlatform - Partie 1

## *Manuel de maintenance*

**Responsables scientifiques :**
Serge CHAUMETTE
Achraf KARRAY
**Responsable pédagogique :**
Mohamed MOSBAH

ABDELKHALEK Rached  BEN MBARKA Moez
DIYAB Rachid  ESSABIR Ayoub
GATTI Christophe  Renteria Morales Eder
TRIKI Hamza

27 avril 2006

# Table des matières

# Chapitre 1

# Package GlobalPlatform

## 1.1 Description général

Comme le nom du chapitre l'indique, nous allons décrire dans ce dernier le code construit dans le package globalPlatform. Le but étant de rendre compte des solutions apportées lors du passage des normes à l'implémentation des différentes opérations décrites dans globalPlatform.

Ce package, conforme aux normes GlobalPlatform, permet la construction des APDUs à envoyer aux lecteurs et implémente les méthodes cryptologiques nécessaires à la connexion avec la carte. Voici le diagramme des classes de ce package.

## 1.2 Diagramme des classes

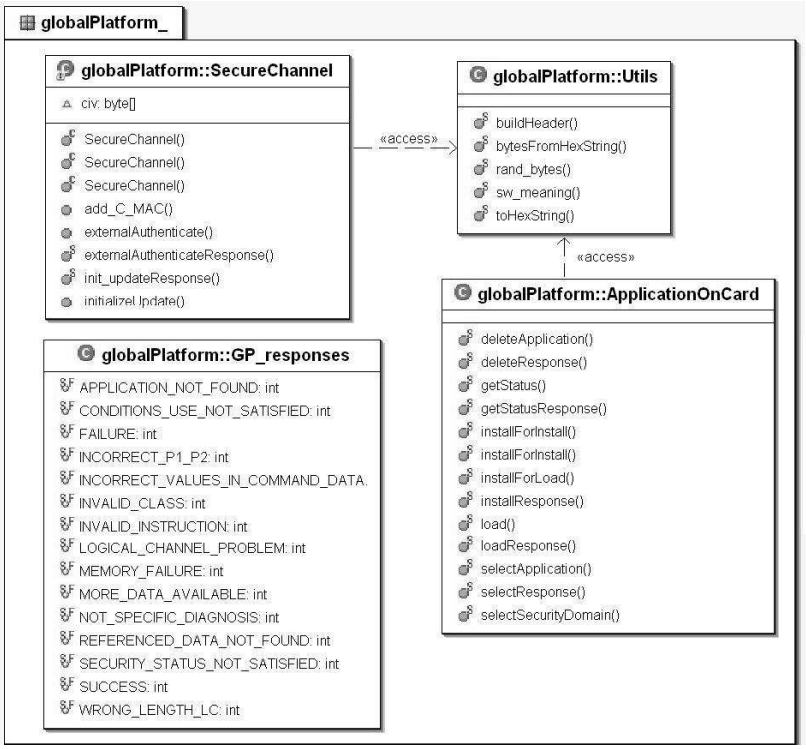


FIG. 1.1 – Package globalPlatform

Les spécifications GlobalPlatform définissent clairement la construction des différentes commandes en spécifiant le contenu des différents champs de la commande APDU.

Selon ces spécifications, certaines commandes supportent plusieurs valeurs possibles pour les champs de paramètrage $P1$ et $P2$, d'autres fixent les valeurs de ces champs. Les méthodes implémentées prennent en paramètres seulement les champs paramétrables.

## 1.3 La classe ApplicationOnCard

Les méthodes de la classe *ApplicationOnCard* ne font que construire l'APDU. L'envoie á la carte et la réception de la réponse, est fait par les méthodes du package $offCard$.

### 1.3.1 Les attributs de la classe

Cette classe n'a pas d'attributs.

### 1.3.2 Les méthodes de la classe

Toutes les méthodes de cette classes sont statiques.

**selectApplication**

- Description :
  Cette méthode permet de construire l' APDU qui va servir à sélectionner une application présente sur la carte à partir de son AID.

- Prototype :

```
public static byte[] selectApplication(byte[] aid, byte p2)
```

- Paramètres :
  byte[] aid : tableau de byte pour l'AID de l'application à selectionner sur la carte.

  byte p2 : C'est un byte qui ne peut prendre que deux valeurs qui ont la signification suivante :

  - '0' : Sélectionner la première occurrence.
  - '1' : Sélectionner l'occurrence suivante.

- Valeur renvoyé :
  La méthode renvoie un tableau de byte qui contient l'APDU correspondante à la sélection.

- Description du code :

  - Etape1 : on construit l'entête de l'APDU de la commande SELECT á l'aide du fonction Utils.builHeader.

```
byte[] header = Utils.buildHeader((byte)0x00 , (byte)0xA4,
                                  (byte)0x04, p2, (byte)aid.length);
```

  - Etape2 : on instantie un tableau de bytes de taille egale à " taille de l'entête + taille de l'AID + 1".
  - Etape3 : on construit l'APDU de la commande en copiant dans le dernier tableau l'entête de l'APDU puis l'AID et dans le dernier byte la valeur "0x00".

**deleteApplication**

- Description :
  Cette commande permet de construire l'APDU qui va servir à supprimer une application présente sur la carte à partir de son AID.

- Prototype :

```
public static byte[] deleteApplication(byte[] aid, byte p2)
```

- Paramètres :
  byte[] aid : un tableau de byte présentant l' AID de l'application à supprimer sur la carte.

  byte p2 : ne peut avoir que deux valeur se distinguant par le dernier bit :
  - '0' : Supprimer seulement l'objet ayant l'AID envoyé.

- '1' : Supprimer en plus les objets liés.

- Valeur renvoyée :
  Renvoie un tableau de byte qui contient l'APDU correspondante à la suppression.
- Description du code :

  - Etape1 : on construit l'entête de l'APDU de la commande á l'aide du fonction Utils.builHeader.

    ```
    byte[]header = Utils.buildHeader((byte)0x80 ,(byte)0xE4,
                                     (byte)0x00, p2,
                                     (byte) (aid.length+2)
                                     /*lc: Data length=
                                     length(4F+ AID.length + AID)*/);
    ```

  - Etape2 : on instantie un tableau de bytes de taille égale à " taille de l'entête + taille de l'AID + 3".

  - Etape3 : on construit l'APDU de la commande en copiant dans le dernier tableau l'entete de l'APDU, puis la valeur "0x4F", puis la taille de l'AID, puis l'AID et dans le dernier byte la valeur "0x00".

**installForLoad**

- Description :
  Cette commande permet de construire l' APDU qui va servir à préparer le chargement d'un package sur la carte à partir de son AID.

- Prototype de la fonction :

  ```
  public static byte[] installForLoad(byte[] loadFileAID,
                                      byte[] securityDomainAID,
                                      byte[] loadFileDatablockHash,
                                      byte[] loadParametersField,
                                      byte[] loadToken) ;
  ```

- Paramètres :
  byte[] loadFileAID : tableau de byte donnant l'AID du package à charger.
  byte[] securityDomainAID : tableau de byte qui donne l'AID du CardManager.
  byte[] loadFileDatablockHash byte[] loadParametersField byte[] loadToken
- Valeur renvoyée :
  Renvoie un tableau de byte correspondant à la commande installforLoad.
- Description du code :

  - Etape1 : on construit l'entete de l'APDU de la commande á l'aide du fonction Utils.builHeader.

    ```
    byte[] header = Utils.buildHeader((byte)0x80, (byte)0xE6,
                                      (byte)0x02,(byte)0x00,(byte) lc);
    ```

    avec lc égale "5 + la somme des tailles des tableaux passé en paramètre".

- Etape2 : on instantie un tableau de bytes de taille égale à " taille de l'entête + lc + 1".
- Etape3 : on construit l'APDU de la commande en copiant dans le dernier tableau dans l'ordre suivant :
  l'entête de l'APDU.
  la taille du paramètre loadFieldAID.
  le paramétre loadFieldAID.
  la taille du paramètre securityDomainAID.
  le paramétre securityDomainAID.
  la taille du paramètre loadFieldDatablockHash puis le paramétre lui même s'il n'est pas vide, si oui alors on copie la valeur 0.
  la taille du paramètre loadParametrsField puis le paramétre lui même s'il n'est pas vide, si oui alors on copie la valeur 0.
  la taille du paramètre loadToken puis le paramétre lui même s'il n'est pas vide, si oui alors on copie la valeur 0.

**La commande Load**

- Description :
  Cette commande permet de construire l'APDU qui va servir à transférer un package sur la carte, en envoyant des morceaux successives de données.
- Prototype de la fonction :

```
public static byte[] load(byte P1, byte P2,
                          int Lc, byte[]loadData)
```

- Paramètres :
  byte P1 : Ce paramètre code sur son dernier bit, s'il s'agit du dernier bloc ('1')ou si la carte doit attendre un autre bloc('0').
  byte P2 : Ce paramètre permet le comptage des blocs envoyés á la carte. En effet, Chaque bloc load doit être envoyé avec son numéro d'ordre dans la totalité des blocs.
  int Lc : byte[] loadData : La taille maximale pour un bloc est caractéristique de la carte, et dans tous les cas elle ne doit pas dépasser 255 octets.
- Valeur renvoyée :
  Cette fonction renvoie un tableau de bytes.
- Description du code :

  - Etape1 : on construit l'entete de l'APDU de la commande à l'aide du fonction Utils.builHeader.

```
byte[] header = Utils.buildHeader((byte)0x80, (byte)0xE8,
                                  P1, P2,(byte)Lc);
```

  - Etape2 : on instantie un tableau de byte de taille égale à " taille de l'entete + taille du paramétre loadData".
  - Etape3 : on construit l'APDU de la commande en copiant dans le dernier tableau l'entete de l'APDU puis le paramètre loadData.

**La commande installForInstall**

- Description :
  Cette commande permet de construire l' APDU qui va servir à installer une

application sur la carte à partir de son AID .

- Prototype de la fonction :

```
public static byte[] installForInstall(byte[] executableLoadFileAID,
                                       byte[] executableModuleAID,
                                       byte[] applicationAID,
                                       byte applicationPrivileges,
                                       byte[] installParameters)
```

- Paramètres :
  byte[] executableLoadFileAID : Tableau de bytes qui donne l'AID du package précédemment chargé.
  byte[] executableModuleAID : Tableau de bytes qui donne l'AID du module.
  byte[] applicationAID : Tableau de bytes qui donne l'AID de l'application à installer.
  byte applicationPrivileges : Donne les privilèges concernant l'application. Ce codage est spécifié dans les spécifications Global Platform.
  byte[] installParameters : Tableau de bytes qui les paramètres d'installation. Ce paramètre est optionnel

- Valeur Renvoyée :
  Cette fonction renvoie un tableau de byte.
- Description du code :
  - etape1 : on construit l'entête de l'APDU de la commande à l'aide de la fonction Utils.builHeader.

```
byte[] header = Utils.buildHeader((byte)0x80, (byte)0xE6,
                                  (byte)0x0C,(byte)0x00,(byte) lc);
```

    avec lc égale " 7 + la somme des tailles des tableaux passé en paramétre".

  - Etape2 : on instantie un tableau de byte de taille egale á " taille de l'entete + lc + 1".
  - Etape3 : on construit l'APDU de la commande en copiant dans le dernier tableau dans l'ordre suivant :
    l'entete de l'APDU.
    la taille du paramétre executableLoadFileAID.
    le paramètre executableLoadFileAID.
    la taille du paramétre executableModuleAID.
    le paramètre executableModuleAID.
    la valeur "(byte)1".
    le paramètre applicationPrivileges.
    la taille du paramètre installParameters.
    le paramètre installParameters.

**La commande getStatus**

- Description :
  Cette commande permet de construire l' APDU qui va servir à obtenir des informations sur la carte, le CardManager ou les applications présentes sur la carte.

- Prototype de la fonction :

```
public static byte[] getStatus(byte p1, byte p2, byte[] data)
```

- Paramètre de la fonction :
  byte p1 : Code le domaine sur lequel la recherche doit être effectué sur la carte.
  - '0x40' : On inclue seulement les applications et le domaine de sécurité.
  - '0x80' : On retourne seulement des informations sur le domaine de sécurité. Avec cette valeur, le critère de recherche est ignoré.

  byte p2 : Le premier bit code si on attend la première occurrence de la réponse ou la suivante. Le résultat de la recherche dans la carte ne peut pas tenir sur une seule réponse. Sur Le deuxième bit on code la forme souhaitée de la réponse. Les spécifications Global Platforme proposent plusieurs formes de réponses dépendant aussi du domaine de la recherche codé en $P1$.
  byte[] data : Ce tableau code le critère de recherche à utiliser (recherche sur une ou plusieurs application, recherche sur la carte ...).

- Valeur renvoyée :
  Cette fonction renvoie un tableau de bytes.

- Description du code :

  - Etape1 : on construit l'entete de l'APDU de la commande á l'aide du fonction Utils.builHeader.

```
byte[]header = Utils.buildHeader((byte)0x80, (byte)0xF2,
                                 p1, p2, (byte)(data.length));
```

  - Etape2 : on instantie un tableau de byte de taille égale à " taille de l'entête + taille du paramètre data + 1".
  - Etape3 : on construit l'APDU de la commande en copiant dans le dernier tableau l'entête de l'APDU puis le paramètre data et dans le dernier byte la valeur "0x00".

## 1.4 La classe SecureChannel

Cette classe implémente les méthodes nécessaires á l'initialisation du canal sécurisé entre la carte et le Host suivant le protocole Secure Channel Protocol 01-SCP01.

Trois niveaux de sécurité sont supportés par SCP01 :

1. **Authentification mutuelle :** La carte et l'utilisateur vérifient qu'ils partagent les même clés secrêtes.
2. **Intégrité des données :** La carte vérifie que le block des données reçu est exactement celui envoyé par la carte qui a fait l'authentification.
3. **Confidentialité :** Les données sont passées chiffrées à la carte.

Seul le premier niveau a été implémenté.
En préalable à toute authentification, les deux entités doivent partager une même information secrète, ici un jeu de clés statiques : S_ENC et S_MAC

### 1.4.1 Les attributs de la classe

- private byte[] HostChallenge : Tableau de bytes aléatoires généré par le Host et envoyé á la carte.
- private byte[] CardChallenge =new byte[8] : Tableau de bytes aléatoires généré par la carte.
- private byte[] civ = new byte[8] : Vecteur d'initialisation utilisé par les fonctions de cryptage triple des.
- private byte[] zero : Tableau de bytes remplis par des zéros.

- private byte[] S_ENC : Clé de session générée à partir de la clé statique Static_ENC
- private byte[] S_MAC : Clé de session générée à partir de la clé statique Static_MAC
- private byte[] static_ENC = new byte[24] : Clé statique connue par le Host et la carte.
- private byte[] static_MAC = new byte[24] : Clé statique connue par le Host et la carte.
- private final String ENCRYPTION_ALGO : Chaîne de caractères contenant le nom de l'algorithme de cryptage.
- private int keyIndex : L'index des clés à utiliser sur la carte.
- private int keyVersionNumber : La version des clés à utiliser sur la carte.
- private Log log : Utilisé pour générer un fichier log.

### 1.4.2 Les méthodes de la classe

**Les constructeurs**

- Prototype :
  La classe possède quatre constructeurs :
  ```
  public SecureChannel(String userKeys);
  public SecureChannel(String userKeys, int keyVersion, int keyIndexNumber);
  public SecureChannel(String userKeys, Log log);
  public SecureChannel();
  ```
- Paramètres :
  String userKeys
  int keyVersion
  int keyIndexNumber
  Log log

- Description du code :
  Selon les paramètres, le constructeur initialise les attributs qui vont être utilisés dans l'authentification en appelant la méthode getInitValues.

  Dans le cas où les paramètres keyVersion et keyIndexNumber sont omis, ils sont initialisés avec la valeur par défaut, à savoir la valeur nulle.

**La méthode getInitValues**

- Description :
  Initialise les attributs. et construit les clés statique MAC et ENC à partir de userKeys.

- Prototype :

```
private void getInitValues(String userKeys);
private void getInitValues(String userKeys, int KeyIndex, int keyVersionNumber);
```

- Paramètres :
  String userKeys :
  int KeyIndex :
  int keyVersionNumber :

- Valeur renvoyée :
  Cette méthode n'a pas de valeur de retour.

**La méthode makeStaticKeys**

- Description : On construit les tableaux de byte pour les clés statiques à partir de la chaine userKeys et en utilisant la mèthode de conversion *s2ba* de la classe Apdu.
- Prototype :

```
private void makeStaticKeys(String userKeys)
```

- Description du code :
  - Etape1 : on récupere la clé passé en paramétre et on le met dans un tableau de byte"static_ENC_" à l'aide la méthode "Apdu.s2ba(String usereKeys)".
  - Etape2 : on copie les 16 premiers bytes du tableau " static_ENC_" dans l'attribut " static_ENC".
  - Etape3 : on copie les 8 derniers bytes du tableau " static_ENC_" dans l'attribut " static_ENC" à partir du 16 ème case.
  - Etape4 : on copie l'attribut " static_ENC$^{'}$ dans l'attribut " static_MAC".

**La méthode initializeUpdate**

- Description :

  Construit l'APDU correspondant à la première phase de l'authentification mutulelle.
- Prototype de la fonction :

```
public byte[] initializeUpdate();
```

- Paramètres :
  Cette méthode ne prend pas de paramètres.

- Valeur renvoyée :
  La commande à envoyer.
- Description du code :

- Etape1 : on construit l'entete de l'APDU de la commande à l'aide du fonction Utils.builHeader.
  ```
  byte[] header = Utils.buildHeader((byte)0x80, (byte)0x50, (byte)keyVersionNumber,
                                    (byte)keyIndex,(byte)0x08);
  ```

- Etape2 : on crée un tableau de bytes aléatoire "HostChalenge" à l'aide de la fonction $Utils.rand_bytes(size)$qui génére aléatoirement des tableaux de bytes .
- Etape3 : on instantie un tableau de bytes de taille égale à " taille de l'entête + taille de HostChallenge + 1".
- Etape4 : on construit l'APDU de la commande en copiant dans le dernier tableau dans l'ordre suivant :
  l'entête de l'APDU.
  le tableau HostChallenge.
  la valeur "(byte)0x00".

**La méthode externalAuthenticate**

- Description :

  Construit l'APDU de la commande external-authenticate.
- Prototype de la fonction :

  ```
  public byte[] externalAuthenticate(byte[] response1,
                                     byte securityLevel) throws Exception;
  ```

- Paramètres :
  byte[] response1 : La réponse de la carte pour la commande initializeUpdate.
  byte securityLevel : Le niveau de sécurité pour les commandes suivantes

- Valeur renvoyée :
  Tableau de bytes qui contient l'APDU de $EXTERNALAUTHENTICATE$
- Description du code :

  - Etape1 : on construit l'entête de l'APDU de la commande á l'aide du fonction Utils.builHeader.
    ```
    byte[] header = Utils.buildHeader((byte)0x80 , (byte)0x82,
                                      securityLevel, (byte)0x00,(byte)0x08);
    ```

  - Etape 2 : On récupère le cardCryptogram et on vérifie sa validité.

**La méthode calculate_hostCryptogram**

- Prototype de la fonction :

  ```
  private byte[] calculate_hostCryptogram(byte[] cardChallenge,
                                          byte[] S_ENC) throws Exception;
  ```
- Paramètres :
  byte[] cardChallenge
  byte[] S_ENC

- Description du code :

- Etape1 : on copie le paramétre “cardChallenge” dans le tableau de bytes “ hostCryptogram_”.
- Etape2 : on copie l’attribut “ HostChallenge” dans ce qui suit dans le tableau “ hostCryptogram_”.
- Etape3 : on copie le tableau “padding” dans les dernieres cases libres du tableau “hostCryptogram_”.
- Etape4 : on génere le tableau de bytes “hostCryptogram” à l’aide de la fonction de cryptage
  `hostCryptogram = des3_CBC_encryption(S_ENC, hostCryptogram_, IvSpec);`
- etape5 : on retourne le tableau de bytes “ hostCryptogram”.

**La méthode des3_ECB_encryption**

- Prototype de la fonction :

```
private byte[] des3_ECB_encryption(byte[] keyBuffer,
                                   byte[] buffer) throws Exception;
private byte[] des3_CBC_encryption(byte[] keyBuffer, byte[] buffer,
                                   IvParameterSpec iv) throws Exception;
```

- Paramètres :
  byte[] keyBuffer
  byte[] buffer
  IvParameterSpec iv

- Description du code :

  Dans cette méthode on génère un tableau de bytes on utilisant des méthodes de cryptages et de géneration de clés comme
  `SecretKeyFactory.getInstance(ENCRYPTION_ALGO);`
  `kf.generateSecret(key);`
  `Cipher.getInstance(ENCRYPTION_ALGO + "/CBC/NoPadding");`

# Chapitre 2

# Package CardGridCom

## 2.1 La classe CardGridUser

### 2.1.1 Description générale

Le PC/SC est un standard de communication entre un PC et un lecteur de cartes à puce. Ce standard est développé pour assurer la compatibilité entre les cartes à puce, les lecteur/encodeurs de carte et les ordinateurs produits par différents constructeurs. Cette nouvelle technologie intégrant carte à puce et PC est construite dans la norme standard ISO 7816, et supporte les applications spécifiques telles que EMV (Europay, MasterCard, Visa) et GSM (Global Standard for Mobile Communication). Elle permet aux développeurs de tirer des avantages sur la portabilité et la sécurité des appareils qui sont deux facteurs importants pour le développement des applications sécurisées.

Les deux implémentations connues de PC/SC sont Microsoft PC/SC et pcsc-lite. Le dernier est développé sous licence libre et donc plus adopté pour notre projet. Dès le début, les clients nous ont proposés l'utilisation de JPC/SC un wrapper java utilisant JNI(Java Native Interface) pour renvoyer les appels de méthodes et les arguments Java vers la bibliothèque client fournie avec Microsoft PC/SC ou pcsc-lite(dans notre cas pcsc-lite).

D'autres solutions supportant le standard PC/SC sont disponibles en Java(comme OCFPCSC). Pour garantir l'indépendance de notre API par rapport à la couche communicant avec les lecteurs, une classe $CardGridUser$ faisant le rôle d'une couche intermédiaire a été fournie par les clients et adopté pour notre projet.
Cette classe offre deux fonctionalités principales :

- La connexion au lecteur.
- l'envoie d'un tableau d'octets à la carte et le retour de la réponse.

### 2.1.2 Les attributs de la classe

- Context ctx =new Context()
- Card card
- String[] sa : contient la liste des lecteurs.
- String szReader

### 2.1.3 Les méthode de la classe

**La méthode CardGridUser**

- Description :
  La connexion au lecteur s'effectue en instanciatiant cette classe CardGridUser avec le nom du lecteur.
- Prototype :

```
public CardGridUser(String cardName);
```

- Paramètres :
  String cardName : Nom de la carte.
- Valeur renvoyée :
  Cette méthode n'a pas de valeur de retour.
- Description du code :
  cette méthode fait appel à la méthode privée *initContext*().

**La méthode initContext**

- Description :
  Cette méthode réalise l'initialisation du contexte afin de pouvoir se connecter au lecteur.
- Prototype :

```
private void initContext();
```

- Paramètres :
  Cette méthode ne prend pas de paramètres.
- Valeur renvoyée :
  Cette fonction n'a pas de valeur de retour.
- Description du code :
  - Etape1 : on instantie un nouveau contexte "ctx".
  - Etape2 : on établie la connection avec le lecteur à l'aide de la fonction

```
ctx.EstablishContext(PCSC.SCOPE_SYSTEM, null, null);
```

**La méthode connect**

Description : Ctte méthode permet la connection avec la carte.

- Prototype :

```
private Card connect(String cardName);
```

- Paramètres :
  String cardName : nom de la carte.
- Valeur renvoyée :
  Card
- Description du code :

  on établie la connection à l'aide de la méthode

```
ctx.Connect(cardName, PCSC.SHARE_EXCLUSIVE, PCSC.PROTOCOL_T0|PCSC.PROTOCOL_T1);
```

**La méthode sendAPDU**

- Description :
  Cette méthode envoie le tableau d'octets ba au lecteur et retourne le tableau d'octets reçu.
- Prototype :

```
public byte[] sendAPDU(String cardID,  byte[] ba);
public byte[] sendAPDU(byte[] ba);
```

- Paramètres :
  String cardID
  byte[] ba

- Valeur renvoyée :
  Renvoie un tableau de bytes qui contient la tableau de bytes reçus.
- Déscription du code :
  cette méthode transmit le tableau de byte passée en paramétre à l'aide la fonction
  `card.Transmit(ba, 0, ba.length);`
  aprés elle retourne la reponse renvoyer par la carte.

# Chapitre 3

# Package OffCard

## 3.1 Diagramme des classes

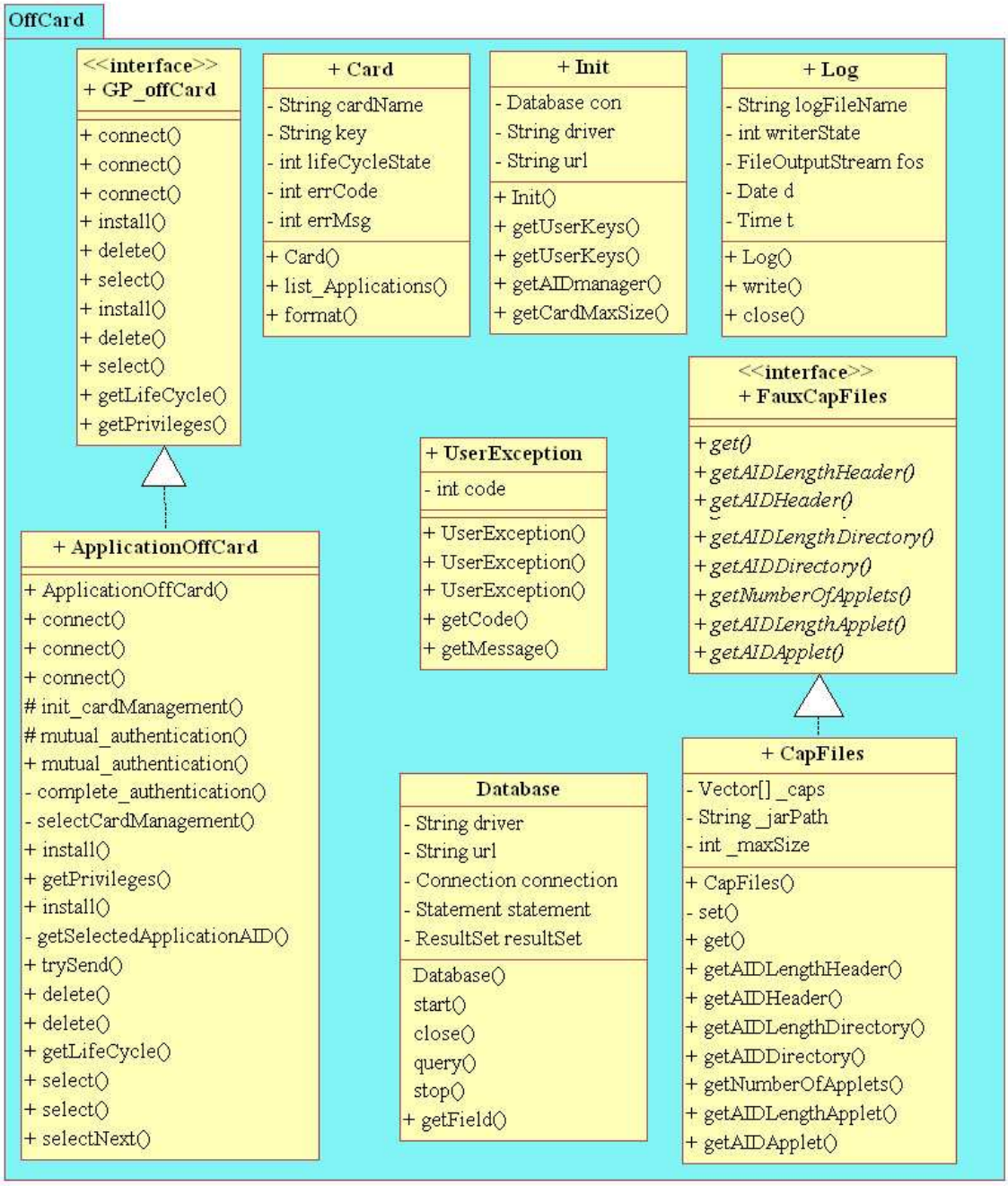


FIG. 3.1 – Package offCard

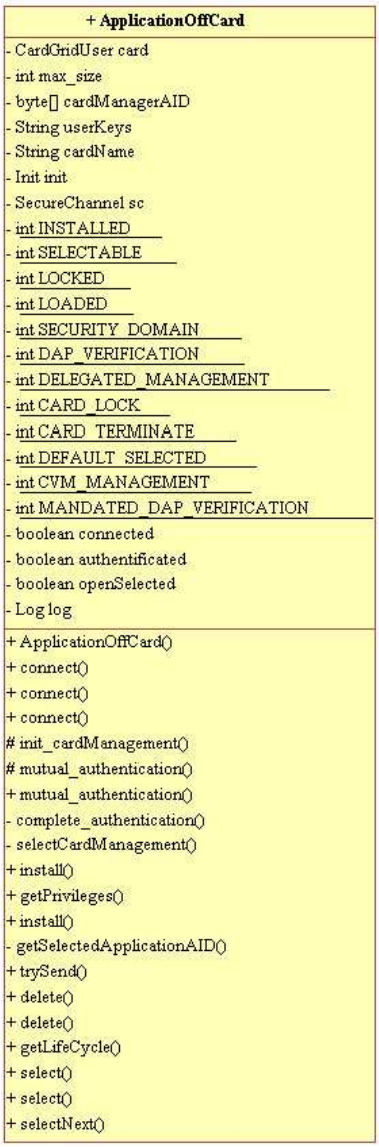


FIG. 3.2 – Classe ApplicationOnCard

## 3.2 La classe ApplicationOffCard

### 3.2.1 Le constructeur

```
public ApplicationOffCard( )
```

### 3.2.2 Les attributs

- private CardGridUser card.
- private int max_size :La taille maximum des données qu'on peut envoyer à la carte, initialisé dans notre cas á 64.
- private byte[] cardManagerAID :AID du Card manager.
- private String userKeys :clé utilisateur.
- private String cardName : nom de la carte.

- private Init init : utilisé pour obtenir des informations de la base de donnée.
- private SecureChannel sc .
- private static final int INSTALLED = 0x03 .
- private static final int SELECTABLE = 0x07 .
- private static final int LOCKED = 0x83 .
- private static final int LOADED = 0x01 .
- private static final int SECURITY_DOMAIN : Macro utilisé pour les privilèges des applications.
- private static final int DAP_VERIFICATION : Macro utilisé pour les privilèges des applications.
- private static final int DELEGATED_MANAGEMENT : Macro utilisé pour les privilèges des applications.
- private static final int CARD_LOCK : Macro utilisé pour les privilèges des applications.
- private static final int CARD_TERMINATE : Macro utilisé pour les privilèges des applications.
- private static final int DEFAULT_SELECTED : Macro utilisé pour les privilèges des applications.
- private static final int CVM_MANAGEMENT : Macro utilisé pour les privilèges des applications.
- private static final int MANDATED_DAP_VERIFICATION : Macro utilisé pour les privilèges des applications.
- private boolean connected : initialisé à faux, et vaut vrai quand la carte est connectée.
- private boolean authentificated : initialisé à faux, et vaut vrai après l'authentification mutuelle.
- private boolean openSelected : initialisé à faux, et vaut vrai après la sélection du card manager.
- private Log log = new Log(0) .

### 3.2.3 Les méthodes

**connect**

- Description :
  Cette méthode ouvre une nouvelle session avec une carte.
- Prototype :
  ```
  public void connect (String cardName) throws UserException;
  public void connect(String cardName, String userkeys) throws UserException;
  public void connect(String cardName, int keyVersion, int keyIndexNumber) throws UserExcep
  ```
- Paramètres :
  String cardName : le nom de la carte.
  int keyVersion : clé utilisateur.
  int keyIndexNumber : l'index du clé ndans la base de donnée.

- Valeur renvoyé :
  Ne renvoie rien.
- Description du code :

  - Etape1 : on initie la connection avec la carte et on selectionne le *cardmanager* à l'aide de la fonction
    `init_cardManagement(cardName) ;`
  - Etape2 : Dans le cas où on ne passe pas les clés de l'authentification en paramètre, on les récupère à l'aide de la fonction

`userKeys= init.getUserKeys(cardName) ;`
- etape3 : On établie l'authentification mutuelle à l'aide la fonction `mutual_authentication(userKeys) ;`

**mutual_authentication**

- Description :
  L'authentification mutuelle est une phase durant laquelle une carte et une entité extérieure vérifient l'identité de leur correspondant.

  Cette étape a pour objectif de vérifier que chacun est celui qu'il prétend être. L'authentification mutuelle est nécessaire avant l'envoie de toute commande de gestion de contenu.

- Prototype :

```
protected void mutual_authentication(String userKeys,
int keyVersion, int keyIndexNumber) throws UserException;

public void mutual_authentication(String userKeys) throws UserException;
```

- Paramètres :
  String cardName : le nom de la carte.
  int keyVersion : clé utilisateur.
  int keyIndexNumber : l'index du clé ndans la base de donnée.

- Valeur renvoyé :
  Ne renvoie rien.
- Description du code :

  - Etape1 : on instancie un canal sécurisé.
  - Etape2 : on appelle initialize-update pour commencer l'authentification.

**initialize_update**

- Description :
  Envoie la commande initialize-update à la carte.
- Prototype :

```
private void initialize_update() throws UserException
```

- Description du code :

  - Etape 1 : On construit la commande init-update avec la méthode initializeUpdate de SecureChannel.
  - Etape 2 : On sauvegarde le niveau de sécurité actuel et on le met à 0.
  - Etape 3 : On envoie la commande à la carte.
  - Etape 4 : On remet le niveau de sécurité précédent.

**complete_authentication**

- Description :
  Termine l'authentification en envoyant la commande external-authentication.
- Prototype :

```
private void complete_authentication(byte[] initResp) throws UserException
```

- Paramètre :
  byte[] initResp : La réponse de la carte à la commande init-update.
- Description du code :

  - Etape 1 : On construit la commande external-authenticate avec la méthode externalAuthenticate de SecureChannel et en utilisant la réponse initResp ainsi que le niveau de sécurité securityLevel.
  - Etape 2 : Si l'authentification réussit on met l'attribut *authenticated* à *true*.

**set_securityLevel**

- Description :
  Change le niveau de sécurité. Si l'authentification a été déjà faite, on appelle la méthode mutual_authentication pour la réinitialiser.
- Prototype :

```
public void set_securityLevel(int level) throws UserException
```

- Paramètre :
  int level : Le nouveau niveau de sécurité.

**install**

- Description :
  Cette méthode enregistre sur la carte un package. Un package est constitué d'un ou plusieurs applets. Cette fonction nécessite que $authenticated = true$ et $openSelected = true$.

  La première commande global platform à envoyer à la carte est $InstallForLoad$.
  L'envoie de cette commande suppose que le *cardmanager* a été sélectionné et que l'authentification mutuelle a réussit. Dans le cas contraire, une exception 'Conditions d'utilisation non satisfaits' sera levée.
  Au cours de cette étape, on envoie l'AID du fichier à charger.
  L'envoie des données est réalisée avec la commande $LOAD$. Il s'agit d'envoyer les applets sous formes de blocks de données successives. La taille maximale d'un block est spécifique à la carte et elle lue de la base des donné.
  Le champ P1 de la commande permet de spécifier si la carte doit attendre un nouveau block ou il s'agit du dernier.

  L'installation du package utilise la commande $InstalForInstall$. Un package, peut contenir plusieurs applets. Il s'agit donc d'envoyer cette commande successivement avec les AID de ces différents applets.

- Prototype :

  - Le premier prototype suppose qu'un appel à *connect* a été fait et donc une connexion avec la carte a été initialisée. Ceci est vérifiée en testant que les variables : *connecte*, *authenticated* et *openselected* sont toutes à *true*.

    ```
    public void install(String packagePath) throws UserException ;
    ```

  - Le deuxième prend tous les parmètres nécessaires pour initialiser une connexion avec la carte et installer le package.

    ```
    public void install(String card, String userKey, String packagePath)
      throws UserException;
    ```

- Paramètres :
  String card, String userKey, String packagePath :

- Valeur renvoyée :
  Ne renvoie rien.
- Description du code :

  - Etape1 : on verifie que l'initialisation de la connection avec la carte a été effectuée.
  - Etape2 : on sélectionne le card manager à l'aide de la méthode .
    `selectCardManagement(cardName) ;`
  - Etape3 : on charge tous les applets qu'on veut installer sur la carte à l'aide de la méthode de bas niveau
    `ApplicationOnCard.load(byte P1, byte P2, int Lc, byte[]loadData);`
  - Etape4 : on install les applets à l'aide la fonction de bas niveau
    ```
    ApplicationOnCard.installForInstall(byte[] executableLoadFileAID,
    byte[] executableModuleAID,
    byte[] applicationAID,
    byte applicationPrivileges,
    byte[] installParameters,
    byte[] installToken)
    ```

**getPrivileges**

- Description :
  Cette méthode retourne les privilèges associés à une application.
  Les privilèges associés à une applications sont codés sur 8 bits. Ils sont divers et variés et concernent la sécurité, le blocage d'une carte, la terminaison d'une carte, la vérification DAP, l'opération par défaut etc...
- Prototype :

  ```
  public String getPrivileges(byte[] AID) throws  UserException;
  ```
- Paramètres :
  byte[] AID : tableau de bytes qui contient l'AID de l'application.
- Valeur renvoyé :
  La réponse retournée est une chaîne de caractère contenant le privilège associé est renvoyée.
- Description du code :

  - Etape1 : on verifie que l'initialisation de la connection avec la carte a été effectuée.
  - Etape2 : on sélectionne le card manager à l'aide de la méthode .
    `selectCardManagement(cardName) ;`
  - Etape3 : on utilise la fonction de bas niveau *getStatus* pour obtenir les informations sur la carte de l'AID passée en paramétre.
  - Etape4 : on cherche dans la reponse renvoyé par la carte le privilége de l'AID, et le retoune.

**trySend**

- Prototype :

  ```
  public byte[] trySend(byte[] apdu) throws UserException;
  ```

- Paramètres :
  byte[] apdu : tableau de bytes qui contient l'APDU.
- Valeur renvoyé :
  Renvoie un tableau de byte.

- Description du code :

  - Etape1 : on envie l'APDU passée en paramétre à l'aide la fonction de bas niveau
    `card.sendAPDU(apdu) ;`
  - Etape2 : on retourne la repnse renvoyé par la carte.

**delete**

- Description :

  Cette méthode supprime un package se trouvant sur la carte. Elle nécessite une authentification préalable et aussi la sélection du *cardmanager*( $authenticated = true$ et $openSelected = true$).
- Prototype :
  Deux signatures sont disponibles :

```
public void delete(String packagePath) throws UserException;

public void delete(String card, String userKey, String packagePath)
  throws UserException;
```

  La première signature suppose qu'une connexion a été initialisé avec *connect*.
- Paramètres : String packagePath :

- Valeur renvoyé : Ne renvoie rien.
- Description du code :

  - Etape1 : on verifie que l'initialisation de la connection avec la carte a été effectuée.
  - Etape2 : on sélectionne le card manager à l'aide de la méthode .
    `selectCardManagement(cardName) ;`
  - Etape3 : on supprime les applets à l'aide de la méthode de bas niveau
    `ApplicationOnCard.deleteApplication(byte[], byte );`
  - Etape4 :on supprime le package avec la même fonction utilisé ci dessus.

**getLifeCycle**

- Description :
  Cette méthode retourne l'état de vie de l'application : sélectionnée, chargée ou installée...
- Prototype :

```
    public String getLifeCycle(byte[] AID) throws UserException{
```

- Paramètres : byte[] AID :

- Valeur renvoyé :
  Cette méthode renvoie un string qui contient l'état de l'application.

- Description du code :

  - Etape1 : on verifie que l'initialisation de la connection avec la carte a été effectuée.
  - Etape2 : on sélectionne le card manager à l'aide de la méthode . `selectCardManagement(cardName) ;`
  - Etape3 : on utilise la fonction de bas niveau *getStatus* pour obtenir les informations sur la carte de l'AID passée en paramétre.
  - Etape4 : on cherche dans la reponse renvoyé par la carte l'état du cycle de vie de l'AID, et le retoune.

**select**

- Description :
  Cette méthode sélectionne une application disponible sur la carte dans l'état *Selectable*. La sélection d'une application sur la carte signifie que c'est cette application qui va reçevoir les commandes suivantes. Pour cette raison on met à la fin *openSelected* à $false$. Ceci permettra de refaire la sélection du *cardmanager* à l'appel suivant.
  Comme pour les méthodes précédentes,
- Prototype :
  Deux signatures sont disponibles pour cette méthode :

  ```
  public void select(String packagePath) throws UserException;

  public void select(String card, String userKey,
                     String packagePath ) throws UserException;
  ```

  La première signature suppose également qu'une connexion a été initialisée avec *connect*.

  A ces méthodes s'ajoute une autre, permettant de sélectionner l'occurence suivante de l'application spécifié :
- Paramètres :
  String card, String userKey, String packagePath.
- Valeur renvoyé :
  Ne renvoie rien.

## 3.3 La classe Card

### 3.3.1 Les constructeurs

```
public Card()

public Card(String cardName, String key) throws UserException
```

La différence entre les deux est que le deuxième dispose des paramètres de connexion, donc il appelle directement la méthode connect(), alors que l'instanciation avec le premier nécessite après un appel à la méthode connect() avant de pouvoir effectuer des opérations sur la carte.

### 3.3.2 Les attributs

- private String cardName.
- private String key.
- private ApplicationOffCard appOffCard.
- private Log log.

### 3.3.3 Les méthodes

**connect**

- Description :
  Initialise une connexion avec la carte. Elle appelle simplement la méthode *connect*() de l'instance de ApplicationOffCard.
- Prototypes :

```
public void connect(String cardName) throws UserException

public void connect(String cardName, String key) throws UserException

public void connect(String cardName,int keyVersion,
  int keyIndexNumber) throws UserException
```

**list_Applications**

- Description :
  Il s'agit de demander à la carte la liste des application qu'elle contient.
  Comme toutes les commandes de gestion de contenue, cette fonction nécessite aussi l'authentification et la sélection du *cardmanager*.

- Prototype :

```
public Collection list_Applications(String card, String key)throws UserException;
```

- Paramètres :
  String card :le nom de la carte.
  String key : la clé utilisateur.

- Valeur renvoyé :
  Renvoie une collection qui contient la liste des application .
- Description du code :
  Cette méthode appelle simplement la méthode list_applications_command deux fois :
  - Avec '20'comme valeur pour $P1$ pour avoir les AIDs des packages chargés.
  - '40' pour $P1$, pour avoir les AID des applications.

**list_Applications_command**

- Description :
  Envoie la commande *getStatus* à la carte avec la valeur $P1$ reçu en paramètre, puis appelle la méthode privée parses_response pour parcourir le tableau de la réponse.

- Prototype :

```
private void list_applications_command(Collection listApplications,
                                       byte p1) throws UserException
```

- Paramètres :
  Collection listApplications : une liste des application qui sont dans la carte.
  byte p1.
- Valeur renvoyé :
  Renvoie une collection qui contient la liste des application .

**parses_response**

- Description :
  Parcourt la réponse de la carte à une commande getStatus.
- Prototype :

```
private void parses_response(Collection listApplications,
                             byte[] response) throws UserException
```

- Paramètres :
  Collection listApplication : C'est la collection à remplir par les AID.
  byte[] response : La réponse de la carte à la commande GETSTATUS.
- Valeur renvoyé :
  Rien. Cette méthode effectue un effet de bord sur la collection passée en paramètre.

**format**

- Description :
  Cette méthode supprime toutes les applications de la carte.
- Prototype :

```
public void format(String cardName, String key) throws UserException;
```

- Paramètres :
  String cardName : le nom de la carte.
  String key : la clé d'utilisateur.
- Valeur renvoyé :
  Ne renvoie rien.
- Description du code :
  - Etape1 : on récupere les applications qui sont sur la carte à l'aide de la méthode $list_Applications()$.
  - Etape2 : on initialise un iterator de la collection retourner par la méthode $list_Applications()$.
  - Etape3 : on parcoure la collection et on supprime les applications une par une à l'aide de la fonction `ApplicationOnCard.deleteApplication(byte[], byte);`

**getApplicationOffCard**

- Description :
  Retourne l'attribut privé appOffCard de la classe ApplicationOffCard.
- Prototype :

```
public ApplicationOffCard getApplicationOffCard()
```

# Chapitre 4

# Evolutions

## 4.1 Supporter le troisième niveau de sécurité : La confidentialité des données

Notre API implémente seulement les deux premiers niveaux de sécurité, à savoir : l'authentification et l'intégrité des données.

Le troisième niveau consiste à assurer en plus la confidentialité des données : Les APDU échangées entre la carte et le PC sont cryptées et non accessibles pour une entité non authentifiée.

Pour implémenter ce niveau, il faut implémenter deux nouvelles méthodes dans la classe SecureChannel :
- encrypt_APDU : Crypte une APDU à envoyer à la carte.
- decrypt_APDU : Décrypte une APDU reçue de la carte.

Ensuite, pour contrôler le cryptage et le décryptage des commandes APDU, il faut ajouter un nouveau bloc à la méthode trySend de la classe ApplicationOffCard. Ce bloc est similaire à celui qui permet d'ajouter un C-MAC à une commande si le niveau de sécurité est égale à 1 :

```
 if (securityLevel == 1 ){
   try{
     apdu=sc.add_C_MAC(apdu) ;
   }catch(Exception e){
     e.printStackTrace();
     throw new UserException("add_C_MAC" + e.getMessage()) ;
   }
 }

/* Bloc à ajouter */
if (securityLevel == 3 ){
  try{
    apdu=sc.encrypt_APDU(apdu) ;
  }catch(Exception e){
    e.printStackTrace();
    throw new UserException("encrypt_APDU" + e.getMessage()) ;
  }
}
```

Il faut faire le même contrôle pour la réponse de la carte :

```
...
  apduResp = sc.decrypt_APDU(card.sendAPDU(apdu)) ;
...
```

D'autre part, pour changer le niveau de sécurité, une méthode set_securityLevel est déjà implémentée :

```
 public void set_securityLevel(int level) throws UserException{
log.write("SecurityLevel set to " + level + ".") ;
this.securityLevel = level ;
if (authenticated){
    mutual_authentication(userKeys) ;
}
    }
```

Cette méthode réinitialise l'authentification si celle-ci a été déjà faite avec un autre niveau de sécurité.

## 4.2 Utilisation d'une autre solution que JPC/SC pour la communication avec une carte

La solution que nous avons utilisé pour la communication avec une carte est JPCSC. D'autres solutions supportant le standard PC/SC sont disponibles en Java(comme OCFPCSC). Pour garantir l'indépendance de notre API par rapport à la couche communicant avec les lecteurs, une classe *CardGridUser* du package *cardGridCom* faisant le rôle d'une couche intermédiaire a été fournie par les clients et adopté pour notre projet.

Donc, pour échanger JPCSC avec une autre bibliothèque, il suffit de modifier cette classe. A commencer par l'import :

```
import com.linuxnet.jpcsc.*;
```

Ensuite, il faut modifier les appels aux différentes méthodes de jpcsc avec celles correspondantes dans la nouvelle bibliothèque à utiliser.

Cependant, il est nécessaire pour le fonctionnement de l'API de garder le package com.linuxnet.jpcsc. Plusieurs méthodes de la classe *Apdu* de ce package sont utilisées telle que les fonctions de conversion entre byte[] et String (et vice versa).

D'autre part, il est naturel de faire des modifications sur les scripts de compilation et d'exécution pour prendre en compte la nouvelle bibliothèque.