Università degli studi di Firenze
Facoltà di Ingegneria
Corso di Informatica Industriale - Prof. A. Fantechi
Elaborato di fine corso

Javacard: uno standard per lo sviluppo di applicazioni su Smart Card con Java

Gerardo Poggiali
gerardo@gestum.it - Febbraio 2002

Indice

1 Tecnologie e standard di mercato **2**
- 1.1 Le Smartcard . . . 2
- 1.2 Lo standard ISO-7816 . . . 4
- 1.3 Javacard . . . 7
- 1.4 subset del linguaggio . . . 9
- 1.5 La sicurezza . . . 10
- 1.6 Teconologie per lo sviluppo di applicazioni off-card . . . 12

2 Sviluppo di una applicazione **14**
- 2.1 L'ambiente standard di sviluppo . . . 14
  - 2.1.1 Sviluppo su workstation . . . 15
  - 2.1.2 Deployment su carta . . . 16
- 2.2 Case study . . . 17
- 2.3 Applet *Libretto* . . . 19
  - 2.3.1 test enviroment . . . 21
  - 2.3.2 inizializzazione dell'applet . . . 22
  - 2.3.3 selezione dell'applet . . . 23
  - 2.3.4 risposta ai comandi APDU . . . 23
  - 2.3.5 Inizializzare le chiavi . . . 24
  - 2.3.6 Il codice PIN . . . 27
  - 2.3.7 Codifica di una stringa . . . 28
  - 2.3.8 altro . . . 29
- 2.4 la classe *Dati* . . . 30

3 conclusioni **32**

4 Appendice **33**
- 4.1 Codice di Firma.java . . . 33
- 4.2 Codice di Dati.java . . . 42
- 4.3 Testing dell'applet . . . 45

Capitolo 1

Tecnologie e standard di mercato

1.1 Le Smartcard

Una *smartcard* si presenta all'aspetto come una carta di credito sulla cui faccia superiore sono presenti dei piccoli contatti dorati (figura ??). Ad ogni contatto corrisponde il pin di un piccolo microchip che contiene essenzialmente il pool di memorie ed il processore. I contatti permettono inoltre alimentazione del dispositivo (l'alimentazione è sempre esterna) e lo scambio di dati attraverso una comunicazione seriale.

Figura 1.1: Foto di una smartcard

Il processore è tipicamente un microcontrollore ad 8 o 16 bit come ad esempio il PIC 16C84 (noto soprattutto perché usato nelle smartcard dei decoder digitali satellitari) e il Dallas 5000. Modelli più recenti possono integrare nel microchip circuiti per l'elaborazione rapida di algoritmi crittografici, come il DES e l'MD5.

Il pool di memorie è così composto:

ROM Questo tipo di memoria è usato per contenere quello che potrebbe essere identificato come il sistema operativo della smartcard.

RAM Contiene heap, stack (necessari all' esecuzione del codice) ed altre strutture dati di natura temporanea.

EEPROM Garantisce la persistenza dei dati e del codice nella smartcard. La memoria di tipo PROM garantisce che i dati siano conservati anche in assenza di alimentazione.

Tipicamente si hanno processori ad 8 bit, 512 bytes di RAM, 32KB di rom e 16KB di EEPROM.

Per leggere una smartcard viene usato un dispositivo di tipo CAD (*card acceptance device*), esso fornisce alimentazione e garantisce la connessione seriale ad un dispositivo esterno.

Figura 1.2: Foto di un dispositivo CAD

L'host esterno può dunque mandare comandi alla smartcard la quale risponde grazie all'esecuzione delle applicazioni memorizzate nella sua EEPROM; è importante notare come la comunicazione avviene sempre con una richiesta da parte dell'unità esterna ed una risposta della smartcard, mai viceversa: in questo senso la smartcard rappresenta un *reactive communicator*.

Le fasi di funzionamento della smartcard compongono il suo ciclo di vita; esso inizia quando la smartcard viene rilasciata e termina quando essa viene distrutta:

1. la smartcard viene inizializzata. In questa fase vengono memorizzati nella sua memoria le applicazioni e i dati di default per la funzione che dovrà andare a svolgere.

2. La smartcard rimane inattiva fino a quando non viene inserita in un CAD. Al momento dell'inserimento inizia il dialogo con il dispositivo di lettura. Per ogni richiesta viene attivata l'applicazione opportuna capace di interpretare il comando e di fornire una risposta consistente. In questa fase possono essere aggiornate le informazioni presenti nella EEPROM e anche aggiunte o rimosse applicazioni.

3. Nel momento in cui viene espulsa dal CAD, la smartcard perde l'alimentazione; perde cioè tutto quanto memorizzato nella RAM conservando però l'informazione sulla EEPROM.

Le possibilità di uso di un dispositivo come questo sono evidentemente limitate dalla sola fantasia dei progettisti, tuttavia esiste una precisa fascia di mercato alla quale le smartcard si rivolge: si tratta nella stragrande maggioranza dei casi di applicazioni di controllo di accesso a risorse e di moneta elettronica. Qui di seguito propongo una serie di esempi delle situazioni più diffuse:

telefonia mobile La tecnologia delle smartcard è alla base dello standard GSM: la smartcard viene usata sia per l'identificazione del terminale nella rete che per la memorizzazione delle informazioni dell'utente (numeri di telefono, SMS, impostazioni ecc.).

moneta elettronica Servizi come "Fai da te" di Agip usano la smartcard come moneta elettronica. Il pregio rispetto alla tradizionale striscia magnetica sta nella maggiore sicurezza determinata dal fatto che risulta molto più difficile alterare il dispositivo, sono maggiormente affidabili (la striscie magnetiche si alterano facilmente) e contengono una maggiore quantità di informazioni.

televisione satellitare La tecnologia delle trasmissione di canali via satellite prevede la crittografazione del segnale per i canali a pagamento. La smartcard viene usata per memorizzare la chiave di decodifica. In questo caso risulta molto utile la facilità con cui è possibile riprogrammare le smartcard: i gestori possono aggiornare la smartcard direttamente durante il suo funzionamento mentre è inserita nel decoder.

servizi integrati Molte comunità (aziende, comuni, servizi sanitari, università) stanno adottando le smartcard per la loro versatilità. E' infatti possibile aggiungere e rimuovere applicazioni dalla carta per poter svolgere più funzioni contemporaneamente. Una azienda, ad esempio, può usare la stessa smartcard per il controllo di accesso ai locali, per l'identificazione nel sistema informativo, per la gestione dei buoni mensa e per l'acquisto dei suoi dipendenti nei negozi convenzionati.

1.2 Lo standard ISO-7816

Lo standard ISO-7816 descrive le specifiche fisiche, elettriche e di comunicazione con le smartcard; è composto di sei parti di cui alcune sono ancora in fase di definizione. Brevemente, le sei parti hanno il seguente contenuto:

Parte 1: Caratteristiche Fisiche Definisce le dimensioni, la loro resistenza a correnti elettrostatiche, radiazioni elettromagnetiche e stress meccanici. Specifica inoltre la posizione di una eventuale striscia magnetica e un area dove vengono inseriti i caratteri in rilievo.

Parte 2: Dimensione e posizione dei contatti Definisce posizione, funzione e caratteristiche elettriche degli 8 contatti metallici, di cui solo 6 sono effettivamente utilizzati

1. VCC: alimentazione
2. RST: reset
3. CLK: segnale di clock

4. GND: Ground
5. VPP: alimentazione di programmazione EEPROM
6. I/O: data input/output

Parte 3: Segnali e Protocollo di trasmissione Definisce il voltaggio, le correnti, il timing per i contatti e due tipi di protocollo:

1. T=0: protocollo half duplex per la trasmissione di caratteri
2. T=1: protocollo half duplex per la trasmissione di blocchi

Parte 4: Protocollo APDU Codifica la struttura dei comandi impartiti dal CAD e delle risposta fornita dalla carta, specifica i codici dei comandi che forniscono le funzioni basilari della carta come selezione dell'applicazione e lettura e scrittura di informazioni, infine fornisce il significato di determinate sequenze di byte come codici di errore e di stato della carta.

I comandi APDU (*Application Protocol Data Unit*) sono codificati attraverso una sequenza di byte di lunghezza variabile con la struttura descritta in figura 1.1. Dove:

CLA Classe di istruzione. Indica il formato e la categoria del comando e della risposta APDU. E' importante notare che alcuni valori sono riservati, come `0x00` che viene usato per la selezione dell'applicazione attiva. Altri comandi possono essere comunque non disponibili allo sviluppatore perché usati dal sistema operativo della smartcard (ad esempio Cyberflex lascia libere soltanto le classi `0xC0 0xA0 0xF0`).

INS Specifica l'istruzione nella classe CLA di comandi

P1 e P2 Parametri addizionali per l'istruzione

LC Numero di byte della parte dati dell'istruzione

Dati Sequenza di byte che costituisce la parte dati

LE Massima lunghezza ammissibile per la parte dati della risposta. Questa informazione serve a scongiurare problemi di overflow sul buffer del CAD.

Header obbligatorio				Corpo opzionale		
CLA	INS	P1	P2	LC	Dati	LE
1 byte	1 byte	1 byte	1 byte	1 byte	n byte	1 byte

Tabella 1.1: Schema del comando APDU

La codifica della risposta avviene invece secondo la schema di figura 1.2, dove:

Dati Sequenza di byte fornita dalla card come risposta al comando

SW1 e SW2 *status word* che indica lo stato di funzionamento della carta. Alcune status word sono definite dallo standard: `0x9000` indica lo stato di funzionamento corretto o di operazione portata a buon fine. I codici tra `0x6000` e `0x6FFF` indicano codici di errore, anche essi definiti nello standard ISO.

Header Obbligatorio		Corpo Opzionale
SW1	SW2	Dati
1 byte	1 byte	n byte

Tabella 1.2: Schema della risposta APDU

Parte 5: Codici di applicazione Ogni applicazione dentro la smartcard è univocamente determinata attraverso un codice composto da sequenza di byte. La convenzione per i nomi segue lo schema in figura 1.3. E' importante che notare come il codice RID sia univocamente assegnato – in tutto il mondo – alle organizzazioni che ne fanno richiesta.

Application Identifier (AID)	
National registered application provider (RID)	Proprietary application identifier extension (PIX)
5 bytes	0 - 11 bytes

Tabella 1.3: Schema del codice AID

Parte 6: codifica dell'informazione nella carta Questa parte è tuttora in fase di definizione. L'obiettivo è di definire una codifica standard per le informazioni strutturate come foto, caratteristiche del possessore (nome, data di nascita, nazionalità, ecc..) e altro utile a rendere possibile l'interscambio di informazioni tra applicazioni diverse.

Lo standard ISO – imponendo delle direttive precise sul funzionamento della smartcard – garantisce che qualsiasi card sia leggibile da qualsiasi dispositivo CAD e che la comunicazione fra essi avvenga secondo un protocollo determinato.

Lo standard invece non descrive in nessun modo come deve essere fatta l'elettronica all'interno della card: non pone nessun tipo di requisito sulle caratteristiche del processore o delle memorie. La determinazione di questi parametri è lasciata al singolo costruttore il quale fornisce la sua particolare implementazione dello standard.

In questo caso viene dunque definita una architettura proprietaria scegliendo il tipo di microcontrollore, la dimensione e tipo delle memorie, ecc.

In pratica però – per far si che la soluzione offerta sia effettivamente utilizzabile – è necessario che il costruttore completi la fornitura con adeguati strumenti per lo sviluppo. Un kit di questo tipo è tipicamente composto da questi elementi:

1. Una serie di routine preinstallate nella carta per facilitare lo sviluppo (in pratica si tratta di fornire già pronti i comandi per scrivere sulle EEPROM, per leggere e scrivere i comandi APDU sulla seriale ecc..)

2. Qualora fosse possibile programmare con linguaggi di alto livello è necessario un compilatore (tipicamente linguaggio C).

3. Software per trasferire e inizializzare il software sulla carta

4. Un ambiente di testing off-card. Il problema del testing è particolarmente sentito in quanto la carta non prevede nessun meccanismo di reset: qualora durante l'esecuzione la carta si portasse in uno stato di funzionamento inconsistente risulterebbe dunque impossibile sbloccarla e sarebbe quindi inutilizzabile.

L'insieme di questi strumenti permette lo sviluppo delle applicazioni *on-card*. E' però evidente come applicazioni scritte per una determinata piattaforma risultino così completamente incompatibili rispetto ad altre senza nessuna possibilità di trasferirle così come sono. Vale la pena anche di notare come risulti comunque complicato scrivere del codice che sia portabile da una piattaforma e l'altra: quando si ha a che fare con memoria di 256 byte e funzioni implementate on-chip risulta praticamente impossibile scrivere del codice generico.

Fintanto che un dispositivo è fornito da una singola organizzazione è lecito aspettarsi che soltanto questa si occupi di inserire applicazioni dunque questo tipo di approccio non pone nessun tipo di limitazione: l'organizzazione stessa, in base ai suoi parametri di convenienza, sceglie nel mercato il prodotto che meglio corrisponde alle sue esigenze.

Qualora invece più organizzazioni desiderassero usare lo stesso dispositivo per funzioni diverse il problema si complica enormemente: risulta difficile trovare una piattaforma in commercio che accontenti tutti ed inoltre nella soluzioni proprietarie come noto non permettono mai di sfruttare al meglio gli avanzamenti tecnologici nel settore.

Da questo è nata la richiesta del mercato di avere soluzioni di tipo trasversale, quali appunto le *Javacard*.

1.3 Javacard

Javacard è una tecnologia basta su Java per la realizzazione di applicazioni da eseguire su smartcard.

Analogamente a quanto avviene per gli applicativi Java su personal computer, una applicazione per Javacard può essere eseguita su qualsiasi smartcard qualora sia presente in essa l'ambiente runtime di esecuzione JCRE (*Java Card Runtime Environment*).

Le applicazioni per Javacard (comunemente chiamate *applet*) sono costituite da codice bytecode indipendente dalla piattaforma che soltanto a runtime viene interpretato ed eseguito dall'ambiente JCRE: in questo modo un costruttore di smartcard può rendere il suo prodotto compatibile con lo standard Javacard una volta implementato per esso una versione del JCRE compatibile con la sua architettura.

La figura (**??**) mostra uno schema della struttura hardware e software della javacard.

I due livelli più bassi (*Hardware* e *smartcard specific OS*) dipendono dal singolo vendor che realizza la smartcard.

Il *Javacard Runtime Environment*(JCRE) è una applicazione standard: esso è composto dalla *Java Card Virtual Machine*(JCVM) e da una serie di classi

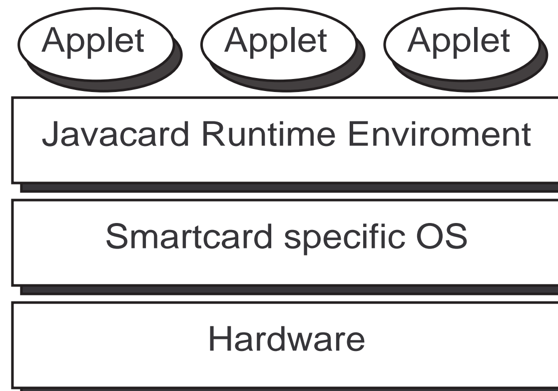

Figura 1.3: Struttura dell'ambiente Javacard

standard. La JCVM è fornita da SUN come sorgente C, è cura del singolo produttore di smartcard il porting sulla sua particolare piattaforma. Anche le classi standard sono fornite da SUN come sorgente Java corredato da ulteriore codice nativo in C di cui ancora è necessario fare il porting; ad esse il produttore di javacard può aggiungere ulteriori classi che permettono allo sviluppatore di sfruttare ulteriori feature di quella particolare implementazione.

La JCVM è una versione ridotta rispetto a quella che si trova nel *Java Runtime Environment (JRE)* disponibile sui personal computer, per questo alcune caratteristiche del linguaggio risultano mancanti, il paragrafo successivo descrive in dettaglio questo aspetto.

Infine sono presenti le applicazioni vere e proprie, le cosiddette *applet*. Le applet sono costituite da file compilati in bytecode `.class` esattamente come le classi java per il JRE standard. In effetti una classe che utilizzi comandi e caratteristiche del linguaggio presenti in entrambe le implementazioni può essere usata indifferentemente nel JCRE che nel JRE standard.

E' però importante notare che i file `.class` non vengono memorizzati nelle smartcard nello stesso modo in cui vengono memorizzati nel file system di un personal computer; questo dipende dal fatto che le EEPROM delle smartcard non implementando una vera e propria struttura di directory: risulta necessario usare il codice AID descritto precedentemente nel protocollo ADPU per gestire il nome di classi e package. Inoltre i normali file `.class` portano con se, oltre al codice eseguibile, tutta una serie di informazioni per permettere l'*introspection* e il *run time type identification*: entrambe queste caratteristiche non sono implementate.

Si ottiene così un nuovo standard di memorizzazione delle classi che risulta notevolmente più compatto, indicato tipicamente come file `.cap` (*converted applet*), dal estensione che viene data questo tipo di file nell'ambiente di sviluppo.

1.4 subset del linguaggio

Come detto precedentemente le JCRE supporta solo un subset del linguaggio java standard:

Caricamento dinamico delle classi Il caricamento dinamico delle classi è una delle caratteristiche più interessanti del linguaggio Java: il fatto che tutti i riferimenti vengono risolti a runtime permette di poter di aggiungere classi durante l'esecuzione del programma e fare riferimento ad esse. Il prezzo da pagare per tutto ciò è un notevole overhead nella gestione dei riferimenti a funzioni ed oggetti durante l'esecuzione del codice; questo overhead è naturalmente inaccettabile su un sistema con prestazioni così ridotte come le javacard. Quindi i programmi in esecuzione sulla carta possono fare riferimento soltanto alle classi che sono già presenti nella carta.

thread La javacard non è multithread, ciò nella smartcard c'è sempre una unica applicazione all'interno di essa in funzione; quindi tutte le keyword del linguaggio che si occupano della sincronizzazione, per esempio *synchronize*, non possono essere usate.

Sicurezza Ci sono delle differenze notevoli tra il modello di sicurezza in Java e quello implementato nelle Javacard. Il paragrafo successivo approfondisce questo argomento.

Garbage Collection Le javacard, per la particolare struttura del funzionamento, non hanno bisogno di un garbage collector in quanto la memoria volatile verrà liberata al momento della disconnessione della carta dal CAD. La javacard non permette neanche di deallocare esplicitamente la memoria occupata da un oggetto; in questo contesto si capisce che il metodo $finalize$, che viene eseguito al momento della distruzione dell'oggetto da parte del GC, non ha più senso e non debba essere usato.

Tipi supportati I tipi standard *char*, *double*, $float$ e *long* non sono supportati; non è inoltre possibile realizzare array con più di una dimensione. In pratica rimangono i tipi *boolean*, *short* , *int* e *byte*.

Classi In generale, quasi nessuna delle classi presenti nelle core API del java standard sono supportate: in pratica rimangono solo la classe $object$ e la classe *throwable* in quanto necessarie rispettivamente nel meccanismo di ereditarietà e di gestione delle eccezioni. Da notare che anche in questo caso si tratta di subset delle classi presenti nel java standard: la classe object delle javacard non supporta il metodo *clone*. Allo stesso modo non è presente la classe $java.lang.System$, questa è sostituita dalla corrispondente classe $javacard.framework.JCSystem$.

Cosa rimane dunque? Sulle Javacard abbiamo ancora le regole per i package con i quali vengono controllati gli accessi tra classi, per l'accesso invece tra classi di package diversi possiamo usare l'interfaccia *sharable*; è naturalmente presente la possibilità di creare oggetti dinamicamente, l'uso delle interfacce , è possibile fare il type checking con il comando $istanceof$ e l'uso dell'eccezioni.

Se una eccezione non viene catturata, arriva al JCRE che la trasforma in una stringa APDU di errore che viene spedita al CAD.

Per il resto tutto quello che è necessario per sviluppare l'applicazione si trova nelle classi *javacard.framework.** e *javacardx.framework.** che verranno descritte successivamente nella parte relativa allo sviluppo di una applicazione.

E' inoltre interessante notare come il linguaggio non richieda una keyword speciale per distinguere se un oggetto deve diventare persistente (ovvero deve rimanere memorizzato tra un inserimento nel CAD e un altro) oppure deve essere volatile (cioè all'estrazione della javacard dal CAD deve essere cancellato). Tutto ciò non è necessario perché il JCRE può prendere agevolmente queste decisioni per noi: vengono mantenuti tra una sessione ed un altra tutti i riferimenti ad oggetti e tipi che sono dichiarati *static* nelle classe; inoltre al momento dell'installazione della classe (cioè quando viene creata una istanza dell'applet) viene considerata persistente l'istanza dell'applet e quindi memorizzati in maniera persistente i riferimenti e tipi dichiarati nella classe; sono quindi volatili tutti i riferimenti e oggetti creati all'interno di metodi la cui visibilità è limitata al metodo stesso.

1.5 La sicurezza

Storicamente java è sempre stato attento alla sicurezza e nelle sue API standard sono state sempre presenti delle librerie che si occupavano di questo delicato capitolo. In java, gran parte delle caratterstiche di sicurezza sono implementate nella classe `java.lang.SecurityManager`; nelle Javacard invece le policy di sicurezza sono implementate direttamente nella JCVM.

Il meccanismo si basa sul concetto di *contesto*; una applet gira in un determinato contesto ed il contesto viene cambiato soltanto quando cambia l'applet in esecuzione. Ad ogni accesso della memoria, la JCVM controlla se quella locazione appartiene al contesto in esecuzione, soltanto in caso affermativo permette l'accesso ai dati.

La semplicità di questo meccanismo garantisce che la memoria di due applet diverse sia sempre distinta, di contro però non abbiamo nessun modo per condividere risorse tra applet diverse.

Il meccanismo invece per ottenere tutto ciò è l'uso dell'interfaccia *Shareable*: i metodi di una interfaccia che deriva da questa possono essere acceduti anche da altre applet. L'idea dunque su cui si basa lo sviluppo di metodi accessibili ad altre applet è il seguente:

- supponiamo di avere una applet di nome *AppletA*
- Un altra applet *AppletB* necessita di usare il metodo *publicMethod* di *AppletA*, in altre parole *AppletA.publicMethod* deve essere accessibile a tutte le applet che girano nella smartcard.
- Per fare tutto ciò è necessario creare una interfaccia, chiamiamola *Applet ASharedInterface* che presenta tutti i metodi di *AppletA* che si vogliono rendere pubblici, *Applet SharedInterface* deve derivare da *javacard. framework. Shareable.*

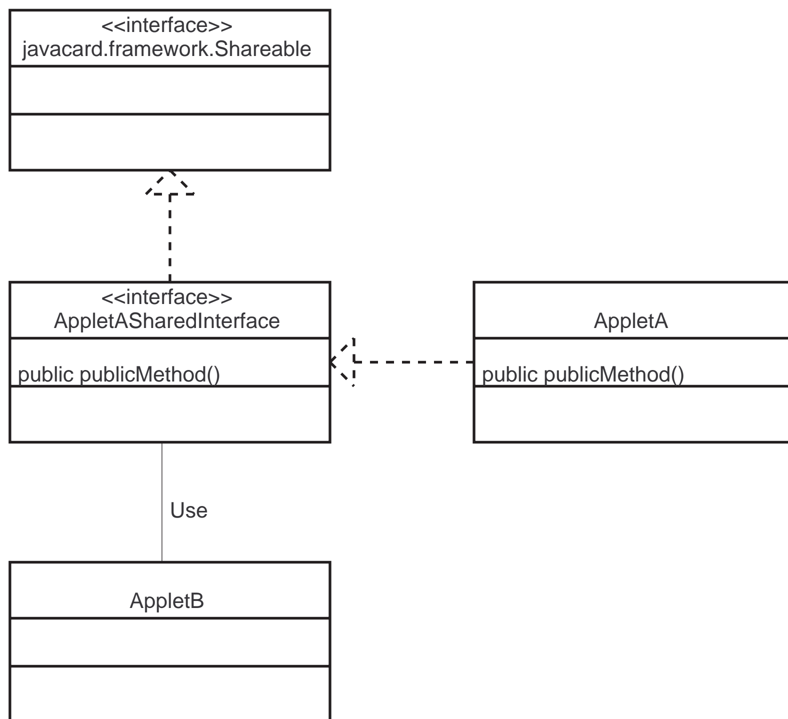

Figura 1.4: Class Diagram su l'uso di *Shareable*

La figura 1.4 mostra quanto detto attraverso un diagramma delle classi UML .

AppletB, per poter usare i metodi di *AppletSharedInterface*, deve ottenere un riferimento ad essa; per farlo usa il metodo *JCSystem. get Appled Shared Interface Object* che richiede sia specificato l'AID dell'applet base (in questo caso dovrebbe usare l'AID di *AppletA*).

E' importante notare che nei metodi pubblici non è possibile passare qualsiasi tipo di dato: in pratica si possono passare i dati base (*byte*, *short*, ecc.) e interfacce che derivano da *Shareable*.

1.6 Teconologie per lo sviluppo di applicazioni off-card

Chiamiamo applicazioni *offcard* tutta la parte di software che si trova nel dispositivo host collegato alla smartcard. Il problema principale nella realizzazione di questo tipo di applicazioni è nell'ottenere una astrazione del singolo dispositivo CAD e rendere quindi la programmazione indipendente dal particolare hardware che connette elettricamente smartcard e host. I CAD possono essere infatti collegati al computer in vari modi, usando porte parallele, seriali, USB ecc. , inoltre ogni dispositivo CAD presenta una sua particolare struttura (presenza di un tastierino per il PIN, soluzioni per l'espulsione della carta ecc.); in determinati campi si vuole avere un framework di sviluppo che ci permetta di non preoccuparci di questi dettagli, sia per convenienza a livello di progetto, sia perché il tipo di CAD che verrà utilizzato non è effettivamente noto. Si pensi ad esempio ad un servizio via Internet che usa l'autenticazione attraverso smartcard: non è certo possibile fissare a priori il tipo di CAD che verrà usato.

Sul mercato sono disponibili due framework di sviluppo:

PC/SC Microsoft in collaborazione con altre industrie del settore ha sviluppato un set di API per comunicare con le smartcard per applicazioni sviluppati su piattaforma Win32 (Windows 9x, ME, 2000 e XP). I produttori di CAD forniscono i driver di sistema compatibili *PC/SC*, le API fornite da microsoft forniscono l'astrazione del CAD per rendere lo sviluppo indipendente dal dispositivo di lettura.

Opencard Si tratta di una soluzione completamente sviluppata in Java. In questo caso si va oltre la sola gestione dell'interfaccia tra CAD e codice, si realizza una struttura di oggetti che rende standard la struttura (a livello di OOD) di questo tipo di applicazioni.

Questo documento si concentra sugli aspetti di sviluppo di applicazioni presenti sulla smartcard e non vorrei approfondire troppo lo sviluppo offcard, è comunque interessante dare uno sguardo all'idea che sta dietro al framework opencard.

A livello di codice Java sarebbe desiderabile che la smartcard fosse rappresentata come un oggetto i cui metodi svolgono le funzioni richieste, senza in effetti preoccuparsi di come sono effettivamente implementati i comandi APDU e come avvenga il dialogo con il dispositivo.

Il sistema opencard fornisce, ai livelli software superiori, una rappresentazione della smartcard attraverso un oggetto *proxy* (figura 1.5); attraverso il meccanismi dell'ereditarietà e implementando le corrette interfacce, il programmatore è agevolato nello scrivere l'oggetto proxy trovandosi già pronti molti servizi concentrandosi così solo sull'aspetto di funzionamento logico del componente.

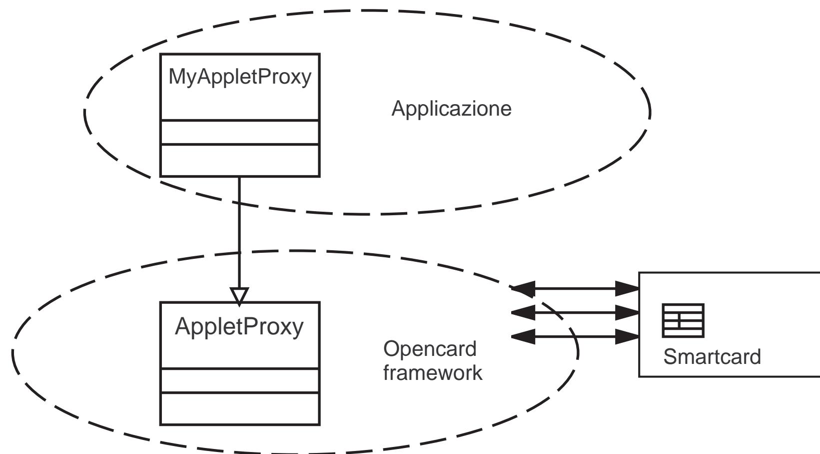

Figura 1.5: Il meccansimo di astrazione di Opencard

Capitolo 2

Sviluppo di una applicazione

In questo capitolo entriamo nel merito delle problematiche relative allo sviluppo dell'applicazione. Nella prima sezione si descrivono gli strumenti forniti dal Java Card Development Kit fornito da Sun; nella seconda sezione si esegue un esempio di applicazione analizzando e commentando il codice sorgente e vedendo i vari passi per effettuare il testing.

2.1 L'ambiente standard di sviluppo

Sun come al solito fornisce gratuitamente l'ambiente di sviluppo (Java Card Development Kit, JCDK) attraverso il suo sito (`http://javacard.sun.com`). All'interno di un file compresso disponibile sia per piattaforma Solaris che per Windows (cambiano fra i due alcuni script), si trovano tutti gli strumenti necessari per sviluppare applicazioni per Javacard.

E' naturalmente necessario avere anche l'ambiente di sviluppo Java (JDK) per workstation e le Javacomm (java communication api). Quest'ultima è una estensione della libreria standard di Java per gestire le porte di comunicazione (seriale, parallela) e deve essere scaricata a parte. Tra l'altro le Javacomm non sono disponibili per tutte le piattaforme (la versione Linux è stata sviluppata da un programmatore indipendente e non da Sun stessa, personalmente io ho trovato problemi nell'usarla), motivo per cui la versione per Solaris non può essere agevolmente adattata ad altri sistemi Unix-like.

All'interno del pacchetto la dotazione di strumenti è completa ma il tutto è decisamente spartano; in piena coerenza con la politica di Sun il JCDK deve servire come base e da reference per la realizzazione – da parte di altri produttori di software – di strumenti più evoluti e user-friendly, nel rispetto degli standard imposti da Sun. Questo è tanto più vero per le Javacard: i costruttori di smartcard possono partire dal JCDK e personalizzarlo e completarlo in base alla loro piattaforma dando del valore aggiunto alla loro soluzione.

Per quanto riguarda la documentazione è presente un gruppo di file in formato pdf che, seppur utilissimi, risultano spesso un po' troppo stringati ed essenziali.

Vediamo quindi quali sono gli strumenti e il loro utilizzo nelle varie fasi di sviluppo di una applicazione distinguendo tre situazioni:

- Sviluppo su workstation
- Deployment su carta da inizializzare
- Deployment su carta già operativa

2.1.1 Sviluppo su workstation

Come già detto, una applet per Javacard è a tutti gli effetti una normale applicazione Java che sfrutta un sottoinsieme ben definito di classi e di costrutti sintattici del linguaggio, può dunque essere compilata con i tradizionali strumenti Java disponibili su workstation. Una volta compilata è però necessario eseguirla in un ambiente che simuli l'ambiente presente su smartcard; per questo sono necessari due strumenti il `jcwde` (Java Card Workstation Development Environment) e l'`apdutool`.

Il primo è a tutti gli effetti un simulatore di Javacard scritto in Java; attraverso file di configurazione si specifica il nome delle classi che vogliamo mettere in questa Javacard virtuale e l'AID associato ad ognuna di esse; al momento dell'avvio questo si collega ad un socket TCP/IP che fa le veci della connessione seriale presente sui contattini dorati della smartcard.

L'`apdutool` è il più semplice strumento per comunicare con il `jcwde`: questo non fa altro che prendere i comandi APDU scritti su un file di testo e spedirli alla porta seriale della smartcard virtuale simulando il dialogo tra smartcard e CAD.

In questa fase dello sviluppo delle applicazioni è possibile inserire del codice "non javacard" all'interno delle applet che verrà rimosso prima di effettuare il deployment su smartcard. Questo è utilissimo in fase di testing perché ovviamente la smartcard, avendo soltanto una comunicazione seriale con l'esterno, non può fornire molte informazioni di debug a runtime sul suo stato; nello sviluppo su workstation invece niente ci vieta di aggiungere codice che visualizzi su console o scriva su file informazioni di questo tipo.

In questo però ci scontra su una lacuna del linguaggio Java non ancora colmata: mancando un preprocessore del linguaggio è impossibile effettuare una compilazione condizionale e dunque necessario, una volta finita la fase di testing, ripulire il codice a mano o realizzare strumenti ad-hoc autonomamente.

La figura 2.1 mostra lo schema del testing su workstation.

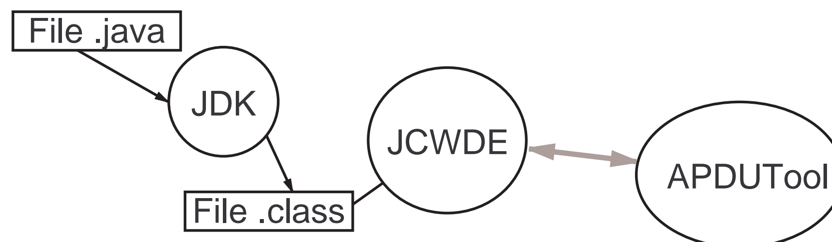

Figura 2.1: Esecuzione su ambiente Workstation

2.1.2 Deployment su carta

La prima cosa da fare per iniziare il deployment è quello di preparare i file `.class` per essere ospitati dalla smartcard. Come detto precedentemente il formato bytecode dei file `.class` risulta essere inutilmente dispendioso portandosi dietro informazioni inutili per il debug e il RTTI; inoltre è necessario tenere conto dei riferimenti a altre classi e package presenti nella smartcard e assegnare alla nostra applicazione il sui AID. Vediamo quindi come, in dettaglio, funziona lo strumento `converter` e come vengono prodotti i file `.jca`, (Java card Assembly).

Una applet fa riferimento nel suo funzionamento a diverse classi che si presumono essere presenti all'interno della javacard; tradizionalmente una JVM risolve questi riferimenti cercando nel filesystem (attraverso la variabile di ambiente `CLASSPATH`) le classi di cui ha bisogno, dato che il nome della classe è una stringa Unicode. Nelle Javacard non esiste file system e il meccanismo di risoluzione dei link deve essere il più ottimizzato ed efficiente possibile; per questo nei file `.jca` le stringhe Unicode sono sostituite da un sistema basato su token.

Tutto ciò comporta che al momento della traduzione del file da `.class` in JCA dobbiamo conoscere la corrispondenza tra token e nomi di classe per tutte le classi di cui fa uso la nostra applet; questa informazione viene memorizzata negli *export files*, che mantengono sulla workstation una "fotograifia" dei riferimenti all'interno della smartcard. Il tool che effettua questa operazioni è il `converter`, in figura 2.2 è schematizzato il suo funzionamento.

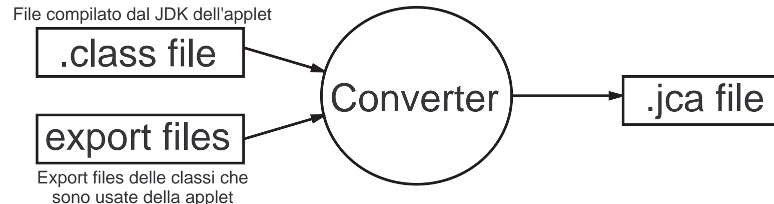

Figura 2.2: Funzionamento dello strumento converter

Il trasferimento dell'applet sulla carta può avvenire in due momenti distinti: quando la carta non è ancora inizializzata e quando la carta è già funzionante.

Nel primo caso l'applet va a fare parte della dotazione software iniziale della carta, quella che nel gergo delle smartcard viene chiamata *mask*. Va notato che questa parte di software può essere considerata fissa per tutto il funzionamento della smartcard e convenientemente posta nella ROM, lasciando libera la preziosa flash ram. In questo caso il tool `maskgen` permette di trattare i file `.jca` e di incorporarli nel runtime enviroment, il risultato è del codice sorgente C o del codice Assembly (attualmente è supportato l'assembler per il microcontrollore 8051).

Nel secondo caso il trasferimento dell'applicazione nella smartcard avviene quando la carta è già in funzione. Nella smartcard deve essere presente una

particolare applet (chiamata *Installer*) che si occupa di questa particolare funzione. Il codice dell'applet viene quindi codificato in una serie di comandi APDU che l' *Installer* riconosce; questa codifica si ottiene, sotto forma di file di testo, generando un file `.cap` con il tool `capgen`.

2.2 Case study

Per vedere con maggiore dettaglio le particolarità dello sviluppo di una applet per Javacard vediamo un piccolo case study: il libretto universitario elettronico. L'idea è di studiare come sia possibile sostituire con una smartcard il tradizionale libretto universitario mettendo nella carta tutte le funzioni che svolge l'equivalente cartaceo.

L'esempio non vuole essere esaustivo, l'obiettivo è quello di sperimentare su qualcosa di reale per vedere nel codice i principali concetti legati allo sviluppo di questo tipo di applicazione.

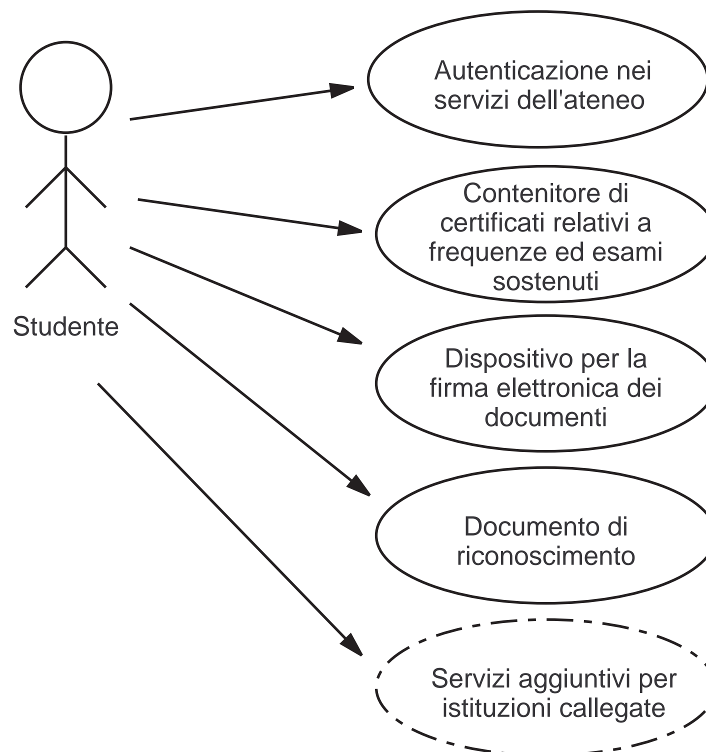

Figura 2.3: Use-case del libretto universitario elettronico

In figura 2.3 viene mostrato il diagramma UML *usecase* del dispositivo. Si individuano i seguenti casi d'uso:

Autenticazione nel sistema informativo dell'ateneo L'accesso a computer, locali e servizi automatici può essere fatto attraverso la smartcard

Repository dei certificati e relativi a frequenze e esami sostenuti Come il libretto reale sulla smartcard si possono memorizzare certificati di frequenza e degli esami sostenuti.

Dispositivo per la firma digitale dei documenti La smartcard, memorizzando la coppia di chiavi pubblica e privata, può servire per la firma digitale di documenti quale il verbale di un esame o altre certificazioni interne.

Documento di riconoscimento Una foto stampata sulla smartcard può servire da riconoscimento da parte di un agente umano

Servizi aggiuntivi per istituzioni collegate La possibilità di installare applet aggiuntive permette a istituzioni collegate all'Ateneo di inserire particolari servizi (si pensi al servizio mensa, sconti sui mezzi di trasporto ecc.).

Nel nostro esempio ci concentriamo sull'aspetto di firma digitale e di repository per i certificati.

Per chiarire ancora meglio il concetto vediamo quale sarebbe l'uso della carta nel contesto di un esame:

1. Lo studente inserisce nel dispostivo host la propria smartcard che viene usata come dispositivo di riconoscimento sia informatico (antraverso l'accesso elettronico alla smartcard) che attraverso l'uso della foto stampata sulla carta stessa.

2. Viene sostenuto l'esame

3. Si passa alla fase di verbalizzazione: il professore compila sul dispositivo host il verbale dell'esame ed inserisce tutti i dati relativi.

4. Il professore inserisce la propria smartcard che, previo inserimento del PIN corretto, viene usata per "firmare" i dati inseriti (tipicamente questa operazione si effettua ricavando un codice hash – detto appunto $firma$ –dall'insieme delle informazioni inserite e codificando la stringa ottenuta con la chiave privata del professore che risiede sulla smartcard).

5. Lo studente prende visione dei dati inseriti inserisce la proria smartcard che, dopo che è stato inserito il PIN corretto, viene usata sia per codificare la firma hash con la chiave privata dello studente che per memorizzare i dati e la firma hash codificata dal professore dei dati dell'esame; notiamo dunque che sulla smartcard è presente una certificazione dell'esame autenticata dal fatto di essere firmata digitalmente dal docente.

6. I dati dell'esame, la firma dello studente e del prefessore vengono memorizzati in un repository centralizzato dell'università; dunque nel sistema informativo dell'ateneo è presente una certificazione incontestatibile dato che presenta la firma di entrambi i soggetti coinvolti.

Questo sistema garantisce un elevato livello di sicurezza perché la chiave privata dei soggetti coinvolti non lascia mai la smartcard in quanto la crittografia avviene direttamente all'interno della smartcard.

2.3 Applet *Libretto*

L'applet svolge le seguenti funzioni:

1. memorizzare chiave pubblica e privata
2. effettuare la crittografazione con la chiave privata di una stringa
3. fornire la chiave pubblica del possessore
4. inserire e leggere i certificati degli esami

Notiamo che è necessario che alcune di queste attività siano protette da PIN, in modo che in caso di furto non sia possibile usare il dispositivo senza conoscere il codice di sblocco.

In definitiva possiamo riassumere i comandi necessari in due gruppi, come descritto dalla tabella 2.1.

Nome Comando	INS	Descrizione	PIN?
Comandi di inizializzazione			
SET_PIN	0x10	Imposta il PIN della carta	NO
SET_PRIKEY	0x20	Imposta la chiave privata	NO
SET_PUBKEY	0x30	Imposta la chiave secondaria	NO
Comandi di funzionamento			
VER_PIN	0x15	Verifica il PIN	-
SET_PIN	0x10	Modifica il PIN della carta	SI
GET_PUBKEY	0x40	Restituisce la chiave pubblica	NO
SIGN	0x50	Codifica una stringa	SI
ADD_FREQ	0xB0	Frequenza di un esame	SI
ADD_PAS_EX	0xC0	Conseguimento di un esame	SI
GET_MEDIA	0xD0	Media dello studente	SI

Tabella 2.1: Elenco dei comandi APDU di *Libretto*

Il funzionamento dell'applet è diviso in due fasi: nella prima fase dobbiamo inizializzare l'applet fornendogli tutte le informazioni necessarie (chiave pubblica, privata e pin), nella seconda abbiamo la fase operativa vera e propria.

Notiamo che nella prima fase il PIN non viene mai chiesto: si suppone infatti che l'inizializzazione avvenga in un ambiente controllato prima del rilascio della carta; naturalmente i valori della chiave private e pubblica, una volta scritti, non possono essere più modificati.

Discorso diverso per il PIN: è possibile che l'utente, per ragioni di sicurezza o di convenienza necessiti di cambiarlo.

Dalla tabella 2.1 si nota che sia per l'inizializzazione che per il cambio del PIN durante la fase operativa si usa lo stesso comando APDU: nel secondo caso è però necessario che la carta sia stata predentemente sbloccata usando il vecchio PIN.

Anche il comando di codifica di stringhe richiede che sia inserito il PIN, non c'è ragione invece per chiederlo quando si rilascia la chiave pubblica.

Nella tabella è segnato anche il codice INS relativo ad ogni camando: vogliamo infatti che i comandi della nostra applet siano identificati nella stringa APDU con il valore CLA 0xB0 e il valore INS pari a uno dei valori sopra specificati.

L'assegnazione del valore CLA è sempre delicata in quanto molte piattaforme riservano solo particolari valori per l'utente; ad esempio la classe 0x00 è usata per molte chiamate di sistema (come ad esempio la *select*, cioè quando viene selezionata l'applet attiva).

Vediamo quindi in dettaglio le varie parti del codice che realizza l'applet.

Qualsiasi applet per Javacard deriva dalla classe *javacard. framework. applet*; il JRCE chiama in causa la nostra applet usando una ben determinata interfaccia:

- *public static void install*([...]) Questo metodo viene chiamato dal JCRE subito dopo che l'applet è stata trasferita nella memoria della carta (durante il funzionamento o alla prima alimentazione della carta se questa si trova nella rom).
- *public void select*() questo metodo viene chiamato quando l'applet viene selezionata, cioè quando l'applet in oggetto diventa quella che riceve i successivi comandi APDU
- *public static void process*(*APDUapdu*) Per ogni comando APDU ricevuto viene chiamato questo metodo che lo processa e fornisce la risposta adeguata.
- *public void deselect*() questo metodo viene chiamato quando viene selezionata una applet diversa.

Ecco dunque quindi lo scheletro della nostra applet:

```
package infind;

import javacard.framework.*;

public class Libretto extends Applet
{
  public static void install(
    byte[] bArray, short bOffset, byte bLength)
  { ... }
  public void select()
  { ... }
  public void process(APDU apdu)
  { ... }
  public void deselect()
  { ... }
}
```

Possiamo fare subito alcune osservazioni: inanzitutto abbiamo definito un package, `infind`, che contiene tutte le classi necessarie. Inoltre le uniche classi che importiamo sono le *javacard.framework.∗*; nel corredo standard del

JCRE oltre a queste troviamo anche le *javacardx.framework.∗* che come lascia dedurre la x in coda al nome, sono delle estensioni accessorie (dunque non supportate su tutte le piattaforme).

2.3.1 test enviroment

Come detto precedentemente effetturemo il testing attraverso il *jcwde*, l'ambiente javacard simulato su macchina host. Per impostare il *jcwde* è necessario un file di configurazione che contenga le informazioni sulle applet che desideriamo siano presenti nella mask della nostra smartcard virtuale. Il file per il nostro esempio appare così (file `infind.app`):

```
//                  RID------------------|PIX----------------
infind.Libretto     0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x6:0x1
```

Le righe che iniziano con // sono semplicemente commenti. Come si vede questo file specifica per ogni classe (la classe deve essere rintracciabile nel classpath) il suo AID.

Una volta lanciato da shell il comando

```
$ jcwde infind.app
```

viene creata la smartcard virtuale la quale si mette in ascolto sulla porta TCP/IP 9095 che svolge la stessa funzione della porta seriale della smartcard "vera".

Per mandare i comandi APDU al *jcwde* possiamo usare il comando *apdutool*; si tratta anche questo caso di un tool a linea di comando che prende i comandi APDU contenuti in un file di testo e li manda alla porta TCP/IP del *jcwde*.

Ad esempio posto il seguente file `select.apdu`:

```
powerup;
// |CLA |INS |P1  |P2  |LC  |DATI                                    |LE  |
// |    |    |    |    |    |RID                 PIX                 |    |
    0x00 0xA4 0x04 0x00 0x0a 0xa0 0x0 0x0 0x0 0x62 0x3 0x1 0xc 0x6 0x1 0x7F;
powerdown;
```

lanciando il comando

```
$ apdutool select.adpu
```

Si ottiene di selezionare come applet attiva l'applet *Libretto*. Nel file i comandi `powerup` e `powerdown` connettono e disconnettono la smartcard; sono presenti dei commenti indicati come di consueto e le stringhe APDU espresse come sequenze di caratteri decimali.

La codifica APDU è già stata descritta nel paragrafo 1.2; in questo caso si trata del comando con CLA 0x00 e INS 0xA4, questo viene sempre intercettato dal JCRE e corrisponde alla attivazione dell'applicazione con il codice specificato nel campo CDATA del comando APDU; in questo caso è presente proprio il codice AID che avevamo specificato nel file `infind.app` per l'applet: in definitiva questo comando non fa altro che selezionare *Libretto* come l'applicazione attiva nel JCRE.

Il tool `apdutool` ci restituisce l'output della smartcard in questo modo:

```
CLA: 00, INS: a4, P1: 04, P2: 00,
Lc: 0a, a0, 00, 00, 00, 62, 03, 01, 0c, 06, 01,
Le: 00, SW1: 90, SW2: 00
```

Cioè sono sia ripetuti i dati spediti in ingresso che riportati i campi della risposta APDU, in questo caso le due status word SW con valore 0x90 0x00 ci dicono che la selezione è andata a buon fine, mentre il valore LE pari a 0 ci dice che non è presente il campo dati nella risposta.

2.3.2 inizializzazione dell'applet

Passiamo a descrivere il codice che inizializza l'applet. Come visto il JCRE chiama il metodo *install* per inizializzare la nostra applet. Il metodo è statico perché in quel momento non esiste nessuna istanza di questo oggetto. Si presume dunque che in questo metodo vengano eseguite tutte le operazioni per sistemare l'applet e renderla operativa; tipicamente in questo metodo si fa semplicemente riferimento al costruttore. Dunque il metodo appare più o meno così:

```
public static void install
  (byte[] bArray, short bOffset, byte bLength)
{
    new Libretto();
}
```

Si tratta di un design pattern interessante: l'idea è che nel motodo install si facciano tutte le verifiche per valutare se la classe possa essere effettivamente inizializzata, è inoltre possibile ricevere dei parametri che sono codificati nel campo CDATA di un comando APDU (nel caso in cui l'installazione sia data esplicitamente attraverso un comando APDU come succede quando l'applet viene trasferita quando la carta è già attiva).

Il costruttore tipicamente presenta il modificatore di accesso *protected*, questo avviene perché solo il metodo install e le classi derivate possono avere bisogno di chiamarlo.

La prima cosa da fare nel costruttore è quella di chiamare il metodo *register*(); la nostra applet lo eredita già pronto da *javacard.framework.Applet*, il suo scopo è quello di aggiungere nel JCRE un riferimento all'istanza appena creata. E' inoltre buona norma allocare qui tutte le strutture dati di cui l'applet avrà bisogno nel suo funzionamento: nelle smartcard la memoria è sempre limitata, dunque non è detto che quando se ne farà richiesta ce ne sia ancora disponibile.

Il nostro costrutture risulta dunque:

```
protected Libretto()
{
  publicKey = new byte[LENGTH_PUBLIC_KEY];
  privateKey = new byte[LENGTH_PRIVATE_KEY];
  pin = new OwnerPIN(PIN_TRY_LIMIT, MAX_PIN_SIZE);
  stringBuffer = new byte[MAX_IN_STRING];

  register();
```

```
  state = NO_KEYS;
}
```

Le strutture dati a cui si fa riferimento nel codice verrano descritte successivamente.

2.3.3 selezione dell'applet

Non abbiamo bisogno di fare niente di particolare al momento della selezione dunque non c'è bisogno di riscrivere il metodo già presente nella classe padre.

2.3.4 risposta ai comandi APDU

All'arrivo di un comando APDU viene chiamato il metodo `process`, ad esso viene passato un riferimento all'oggetto APDU che rappresenta il comando ricevuto dal CAD.

Il comando APDU viene memorizzato in una parte della memoria della carta chiamato $buffer$. E' importante notare che mon è assolutamente detto che tutto il comando possa essere memorizzato nel buffer della carta, in quel caso si dovrà leggerlo in più passi.

Per prima cosa dobbiamo ottenere dal JCRE un riferimento al buffer che contiene il comando:

```
byte buffer[] = apdu.getBuffer();
```

Il buffer non è altro che una stringa di byte che contiene il comando APDU esattamente come è stato inviato.

Posto che le istruzioni per l'applet abbiano tutte il codice CLA uguale a 0xB0, è buona pratica assegnare questo valore ad una costante per migliorare la lettura del codice e per rendere facili le modifiche. In Java il metodo per esprimere le costanti è usare variabili di classi definite $final$:

```
final static byte Applet_CLA = 0xB0;
```

Poiché anche il comando di `select` viene passato alla *process*, questo deve essere identificato e filtrarto. Quindi:

```
if ((buffer[ISO7816.OFFSET_CLA] == 0)
   && (buffer[ISO7816.OFFSET_INS] == (byte) 0xA4))
   return;
}
```

Notiamo come le posizoni dei campi nella stringa APDU siano già codificata all'interno dell'interfaccia $ISO7816$; in Java è infatti abitudine raccogliere un insieme di costanti in una interfaccia (ricordiamo infatti che non esiste l'eredità multipla, dunque se per convenienza si vogliono avere queste costanti nel namespace corrente è sufficiente implementare l'interfaccia, il linguaggio non pone limiti al numero di interfacce implementate). Nel caso di codice CLA e INS relativi alla *select* ci si limita semplicemente a non fare nulla e restituire il controllo al JCRE. Per invece escludere i comandi che non competono a questa applet si ha:

```
if (buffer[ISO7816.OFFSET_CLA] != Applet_CLA) {
  ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
}
```

In questo comando si fa riferimento ad un altro capitolo importante nello sviluppo delle applicazioni per smartcard: l'eccezioni.

Il tipo *ISOException* prevede che gli sia passato un codice di 2 byte che rappresenta il codice dell'eccezione. Se non viene intercettato prima, l'eccezione arriva al JCRE che usa il codice per comporre la risposta APDU, inserendolo nei campi SW1 e SW2.

Dunque nel caso di un comando APDU di questo tipo:

```
CLA: 20, INS: 20, P1: 00, P2: 00, Lc: 00, Le: 00
```

Si ha una come risposta:

```
SW1: 6e, SW2: 00
```

Dove appunto il codice 0x6e00 è appunto quello associato alla eccezione relativa ad un errore generato dalla ricezione di un comando non supportato.

A questo punto, essendo sicuri che il comando è di pertinenza della nostra applet, usiamo l'istruzione `switch` per smistare i vari comandi in altrettante funzioni:

```
switch (buffer[ISO7816.OFFSET_INS]) {
  case SET_PIN:
    setPin(apdu);
    return;
  case VER_PIN:
    verPin(apdu);
    return;
  case SET_PRIKEY:
    setPrivateKey(apdu);
    return;
  case SET_PUBKEY:
    setPublicKey(apdu);
    return;
  case GET_PUBKEY:
    getPublicKey(apdu);
    return;
  case SIGN:
    sign(apdu);
    return;
  default:
    ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    return;
}
```

2.3.5 Inizializzare le chiavi

Nella fase di inizializzazione dobbiamo trasferire le stringhe che rappresentano le chiavi private e pubbliche per la crittografia di tipo RSA.

Un alternativa che aumenterebbe ancora di più la sicurezza potrebbe essere quella di far generare le chiavi direttamente alla smartcard; in questo modo la chiave privata nascerebbe dentro la smartcard e non ci sarebbe nessun modo per intercettarla; la cosa è però improponibile perché per quanto efficiente possa essere l'algoritmo di generazione, con processori di questo tipo, ci vorrebbero almeno tre minuti (cfr. sito web della *Litronic*, http://www.litronic.com), un tempo di questo tipo andrebbe di molto oltre il timeout previsto per la risposta della carta al comando APDU (che è di solito di pochi secondi).

Per quanto riguarda invece la firma e la verifica RSA si riesce agevolmente a stare sotto il secondo anche con chiavi da 1024 byte. Nel mio esempio, per semplicità, le chiavi sono di solo 128 byte.

Il comando 0x20 è assegnato al trasferimento della chiave privata; il camando APDU che effettua questo trasferimento è

```
CLA |INS |P1  |P2  |LC  |                                                |LE  |
0xB0 0x20 0x00 0x00 0x08 0x20 0x20 0x20 0x20 0x20 0x20 0x20 0x20  0x7F
```

La chiave è inserita nel campo CDATA del comando, va notato che il campo LC contiene la lunghezza in byte dei dati (in questo caso 8).

Dovremo anche aggiungere alla nostra classe una variabile di instanza che contenga la chiave; si tratta di memoria persistente in quanto una variabile di instanza viene conservata nella flash ram.

```
private byte[] privateKey;
```

Ecco il codice completo della funzione che esegue l'inserimento:

```
void setPrivateKey(APDU apdu) {

 // Se la chiave privata e' gia' stata memorizzata impedisce che sia
 // memorizzata di nuovo
 if ((state & HAVE_PRIKEY) == HAVE_PRIKEY)
  ISOException.throwIt(ISO7816.SW_COMMAND_NOT_ALLOWED);

 byte buffer[] = apdu.getBuffer();

 // Controlla se la chiave specificata e' della lunghezza corretta
 if (buffer[ISO7816.OFFSET_LC] != LENGTH_PRIVATE_KEY)
  ISOException.throwIt(ISO7816.SW_WRONG_DATA);

 // Ricava il numero di byte letti del comando APDU
 short bytesRead = apdu.setIncomingAndReceive();
 // Posiziona il cursore all'inizio del buffer
 short echoOffset = (short)0;
 // Scorre il buffer e copia i dati nella chiave privata
 while ( bytesRead > 0 ) {
  Util.arrayCopyNonAtomic(buffer,
   ISO7816.OFFSET_CDATA, privateKey, echoOffset, bytesRead);
   echoOffset += bytesRead;
   bytesRead = apdu.receiveBytes(ISO7816.OFFSET_CDATA);
  }
```

```
 // Aggiorna lo stato della carta
 state = (byte) (state | HAVE_PRIKEY);
}
```

Inanzitutto vogliamo che la chiave venga inserita una sola volta; il metodo migliore per tenere traccia di questo è usare un flag di stato e, poiché la memoria nelle smartcard è sempre poca, conviene economizzare usandolo per più attività.

Definiamo quindi un byte di stato i cui bit hanno il seguente significato:

HAVE_PIN alto se il pin inserito, basso viceversa

HAVE_PUBKEY alto se la chiave pubblica è inserita

HAVE_PRIKEY alto se la chiave privata è inserita

La nostra smartcard sarà quindi operativa quando questi tre bit saranno tutti alti.

E' interessante anche osservare le routine che legge la stringa e la copia in *privateKey*:

```
 short bytesRead = apdu.setIncomingAndReceive();
 short echoOffset = (short)0;
 while ( bytesRead > 0 ) {
  Util.arrayCopyNonAtomic(buffer,
   ISO7816.OFFSET_CDATA, privateKey, echoOffset, bytesRead);
   echoOffset += bytesRead;
   bytesRead = apdu.receiveBytes(ISO7816.OFFSET_CDATA);
  }
```

La funzione *APDU.setIncomingAndReceive* trasferisce un primo pezzetto della stringa APDU fino a esaurire il buffer della carta: la dimensione di questo dispositivo non è nota a priori è può variare da un tipo di smartcard ad un altra; questo problema si aggira continuando a chiamare la funzione *APDU.receiveButes*() fino a che non ci sono più byte da leggere.

La procedura che imposta la chiave pubblica non presenta particolari differenze rispetto a questa (vedi appendice per il codice completo).

Vale invece la pena di dare un'occhiata alla funzione che manda in output la chiave pubblica:

```
void getPublicKey(APDU apdu) {
  // Controlla se la carta è attiva e correttamente inizializzata
  if (state != (HAVE_PUBKEY | HAVE_PRIKEY | HAVE_PIN))
      ISOException.throwIt(ISO7816.SW_COMMAND_NOT_ALLOWED);

  apdu.setOutgoing();
  apdu.setOutgoingLength( (short) LENGTH_PUBLIC_KEY);
  Apdu.sendBytesLong( publicKey, (short) 0, LENGTH_PUBLIC_KEY );
}
```

Il comando *ADPU.sendBytesLong* nasconde molta della complessità insita nel trasferimento dei dati da Smartcard a CAD; notiamo infatti che il comando APDU presenta il campo LE che indica il numero massimo di byte che il CAD accetta in una risposta (il campo LE è una caratteristica del protocollo T=1, nel protocollo T=0 il campo dati ha una lunghezza massima fissata dal protocollo stesso). Se si vuole rispondere con un blocco di dati più lungo del buffer del CAD è necessario usare più comandi consecutivi secondo una specifica del protocollo detta *blockchaining*; il comando *sendBytes* fa tutto questo per noi.

2.3.6 Il codice PIN

Le API delle Javacard forniscono già pronte le routine per gestire il PIN, in particolare l'oggetto *OwnerPIN* è tutto ciò che ci serve. Posto come mebro della classe

```
OwnerPIN pin;
```

se ne crea una istanza nel costruttore passandogli il numero massimo di tentativi prima che la carta sia bloccata e la sua lunghezza massima:

```
pin = new OwnerPIN(PIN_TRY_LIMIT, MAX_PIN_SIZE);
```

L'oggetto *OnerPIN* fornisce i metodi *check* per verificare la correttezza del PIN e il metodo *update* per verificarne il valore. Se il metodo *check* fallisce per più di PIN_TRY_LIMIT volte consecutive, continuerà a restituire *false* anche se il codice inserito è corretto.

Detto questo il metodo di verifica del PIN è:

```
void verPin(APDU apdu) {
  // Controlla che la carta sia completamente inizializzata
  if (state != (HAVE_PUBKEY | HAVE_PRIKEY | HAVE_PIN))
   ISOException.throwIt(ISO7816.SW_COMMAND_NOT_ALLOWED);

  byte buffer[] = apdu.getBuffer();

  byte byteRead = (byte) (apdu.setIncomingAndReceive());

  if ( pin.check(buffer, ISO7816.OFFSET_CDATA, byteRead) == false)
   ISOException.throwIt(SW_VERIFICATION_FAILED);
 }
```

Mentre il metodo per inserire e modificare il valore del pin è:

```
void setPin(APDU apdu) {

  if (((state & HAVE_PIN) == HAVE_PIN) && ( ! pin.isValidated() ))
   ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);

  byte buffer[] = apdu.getBuffer();

  // Controlla se la lunghezza del pin e' corretta
  if (buffer[ISO7816.OFFSET_LC] > MAX_PIN_SIZE)
```

```
   ISOException.throwIt(ISO7816.SW_WRONG_DATA);

  byte byteRead = (byte) (apdu.setIncomingAndReceive());
  pin.update(buffer, ISO7816.OFFSET_CDATA, byteRead);

  // Aggiorna lo stato
  state = (byte) (state | HAVE_PIN);
}
```

dove si nota il doppio funzionamento della funzione (sia per inizializzare che per modificare il PIN); la lettura del PIN dal comando APDU non presenta particolari differenze con quanto già visto precedentemente.

2.3.7 Codifica di una stringa

Il comando APDU passa la stringa da codificare nel campo dati, come di consueto:

```
|CLA |INS |P1  |P2  |LC  |
 0xB0 0x50 0x00 0x00 0x0A
| CDATA                                            |LE  |
 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A 0x7F
```

Dato che la stringa può essere maggiore del buffer della carta (è anzi probabile che lo sia) è necessario prevedere una area di memoria che possa contenere la stringa completa:

```
private byte[] stringBuffer;
```

Ecco quindici il codice completo del comando, dove si fa riferimento ad un ipotetico metodo *RSAEncode* che esegue la codifica; il metodo non è stato specificato in quanto la realizzazione dipende dai metodi che il produttore prevede per il codice nativo che si occupa di ciò:

```
void sign(APDU apdu) {

 // Controlla che la carta sia stata correttamente configurata
 if ( state != (HAVE_PUBKEY | HAVE_PRIKEY | HAVE_PIN) )
  ISOException.throwIt(ISO7816.SW_COMMAND_NOT_ALLOWED);

 // Controlla se il pin e' stato verificato
 if ( ! pin.isValidated() )
  ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);

 byte buffer[] = apdu.getBuffer();

 // Controlla che la stringa da criptare non sia più lunga del massimo
 // consentito
 if (buffer[ISO7816.OFFSET_LC] > MAX_IN_STRING)
  ISOException.throwIt(ISO7816.SW_WRONG_DATA);
```

```
 // Effettua la copia della stringa in ingresso nello stringBuffer
 short bytesRead = apdu.setIncomingAndReceive();
 short echoOffset = (short)0;
 while ( bytesRead > 0 ) {
  Util.arrayCopyNonAtomic(buffer, ISO7816.OFFSET_CDATA,
      stringBuffer, echoOffset, bytesRead);
      echoOffset += bytesRead;
      bytesRead = apdu.receiveBytes(ISO7816.OFFSET_CDATA);
    }
 // Cripta il contenuto dello stringBuffer
 RSAEncode();

 // Manda in output la stringa crittografata
    apdu.setOutgoing();
 apdu.setOutgoingLength( (short) echoOffset);
    apdu.sendBytesLong( stringBuffer, (short) 0, echoOffset );
}
```

2.3.8 altro

Sono presenti anche i metodi che permettono di aggiungere certificati di esami e di frequenza, questi fanno riferimento alla classe *Dati* specificata in seguito; alla luce del codice presentato in precedenza non introducono nessuna particolare novità.

Infine ho supposto che si volesse rendere pubbico il metodo *getMedia* che fornisce la media della studente; per fare questo, come spiegato precedentemente, è necessario creare una interfaccia *LibrettoShared* che implementa *Shareable*, ecco il codice che definisce l'interfaccia:

```
package infind;

import javacard.framework.*;

public interface LibrettoShared extends Shareable {
public short getMedia();
}
```

Una ipotetica classe che volesse accedere al metodo *getMedia* di *Libretto* dovrebbe usare il seguente codice:

```
// Ricava un riferimento all'interfaccia che rappresenta
// l'applet
LibrettoShared link = (LibrettoShared)
 getAppledSharedInterfaceObject(LibrettoAID, (byte) 0);
// Esegue il metodo
media = link.getMedia();
```

dove *LibrettoAID* è appunto l'AID dell'applet *Libretto*.

2.4 la classe *Dati*

Vediamo adesso come sono rappresentati i certificati di frequenza e di conseguimento di un esame nella smartcard. Si vogliono memorizzare fino a 50 esami; per ogni record si hanno i seguenti campi:

state stato del record (scelto tra i seguenti valori: EX_INACTIVE (record indefinito), EX_FREQ (ottenuta frequenza dell'esame), EX_PASSED (esame passato) più altri lasciati liberi per altri utilizzi) [1 byte]

code codice dell'esame

date data dell'esame

vote voto ottenuto

sign Stringa con la firma digitale del professore

Il linguaggio Java, dalla versione 1.1 in poi, presenta una articolata struttura di oggetti atta a rappresentare le strutture dati complesse, chiamata *Conteiner*. Tutto ciò non è presente sulla Javacard i cui tipi base sono ulteriormente limitati dal non avere array multidimensionali. D'altra parte una smartcard ha dei requisiti particolari in quanto a memoria sia per le dimensioni (supponiamo ad esempio di volere che l'occupazione per la struttura dati sia minore di 3 Kb) e per la caratteristica di dover allocare subito tutta la RAM.

Privilegiando la possibilità di inserire la stringa di firma il più lunga possibile per favorire la sicurezza, il record potrebbe essere rappresentato così:

state 1 byte

code 2 byte

date 2 byte (il primo gennaio del 2000 è il giorno 0 a seguire gli altri fino per quasi 180 anni di range)

vote 1 byte

sign 48 byte

Per una occupazione totale di 2700 byte; a questo valore vanno aggiunte le chiavi e il buffer per la stringa da codificare e i dati accessori: anche supponendo le chiavi a 128 byte (1024 bit) ed un buffer grande per la codifica delle stringhe si rimane sotte i 4 Kb per tutta la memoria necessaria ai dati. Aggiungendo il codice (difficilmente si supererebbero i 2 Kbyte) – anche se non ho potuto provare – credo che non ci sarebbe nessun problema ad ospittare il tutto su una carta con prestazioni standard.

In virtù di tutto questo ho creato una classe *Dati* che contiene la struttura dati e i metodi per accedervi in maniera sicura e consistente. Per rendere gli accessi veloci ed efficienti ho deciso quindi allocare un unico grande array in cui muoversi esattamente come si farebbe con un puntatore. In effetti sarebbe stato possibile anche usare un oggetto *esame* e poi riferirsi ai metodi di questo per accedere per i valori; ho preferito però non seguire questa via in quanto negli esempi di codice che ho trovato si evitava quasi sempre di effettuare troppe reindirezioni sui riferimenti ai dati per non inficiare l'efficienza.

In appendice è presente il codice completo della classe che non presenta grandi particolarità: come al solito si pone particolare cura alla efficienza del codice, si usano sempre tipi primitivi e calcoli veloci; per operazioni più complicate, come ad esempio la copia di una stringa di byte in un altra si possono usare i metodi presenti nelle API: ad esempio il metodo $arrayCopyNonAtomic$ è implementato come metodo nativo, serve per copiare il contenuto di un array.

Un altro aspetto che può generare confusione è l'uso del tipo *short*, in java i tipi base sono sempre con segno, dunque *short* varia da `-0x8000` a `0x07FFF`, se si vuole invece usare solo i numeri positivi (come abbiamo fatto noi per i campi *code* e *date*) bisogna preoccuparsi di creare il codice per la conversione tenendo conto di questo.

Capitolo 3

conclusioni

E' interessante vedere come un linguaggio complesso come Java possa essere adattato allo sviluppo su una piattaforma dai requisiti cosi stringenti come una smartcard.

Il successo di Java là dove troviamo sistemi complessi e grandi quantità di codice non è ormai più in discussione: l'eleganza e la grande coerenza del modello ad oggetti lo rende particolarmente adatto alle recenti tecniche di progettazione dell'ingegneria del software; rimangono invece ancora forti dubbi sulla sua adattabilità a situazioni più estreme, come appunto la programmazione dei dispositivi embedded.

Eppure da quanto ho potuto vedere per le javacard il risultato è convincente: certo in questo caso l'aspetto che più risalta è la possibilità di creare applicazioni multipiattaforma, però mi pare rilevante la qualità del codice che viene realizzato. Grazie ad una struttura di classi molto coerente il codice rimane molto leggibile anche se è necessario tenere ben a mente le specifiche del dispositivo; tutto questo comunque dimostra che la programmazione ad oggetti può e deve essere affrontata anche su dispositivi così piccoli come una smartcard.

Capitolo 4

Appendice

4.1 Codice di Firma.java

```
package infind;

import javacard.framework.*;


public class Libretto extends Applet implements LibrettoShared
{
 // Le strutture dati che contengono la chiave pubblica e privata
 private byte[] publicKey;
 private byte[] privateKey;

 // Indica lo stato della carta
 private byte state;

  // Costanti che indicano lo stato della carta
 final static byte NO_KEYS = 0x00;
 final static byte HAVE_PIN = 0x01;
 final static byte HAVE_PUBKEY = 0x02;
 final static byte HAVE_PRIKEY = 0x04;

 // Constanti che indicano la lunghezza delle chiavi
  private static final short LENGTH_PRIVATE_KEY = 8;
  private static final short LENGTH_PUBLIC_KEY = 8;

 // Costante che indica la lunghezza massima della stringa da criptare
 private static final short MAX_IN_STRING = 256;

 // Il buffer per le stringhe da firmare
 private byte[] stringBuffer;

 // Costanti dei comandi APDU
```

```
 final static byte Libretto_CLA = (byte)0xB0;
 final static byte SET_PIN = (byte) 0x10;
 final static byte VER_PIN = (byte) 0x15;
 final static byte SET_PRIKEY = (byte)0x20;
 final static byte SET_PUBKEY = (byte)0x30;
 final static byte GET_PUBKEY = (byte)0x40;
 final static byte SIGN = (byte)0x50;
 final static byte ADD_FREQUENCY = (byte) 0xB0;
 final static byte ADD_PASSED_EXAM = (byte) 0xC0;
 final static byte GET_MEDIA =(byte) 0xD0;

 // Codici di errore per errori specifici dell'applet
 final static short SW_VERIFICATION_FAILED = 0x6300;
 final static short SW_PIN_VERIFICATION_REQUIRED = 0x6301;


 // PIN
 // notare che altre applet dello stesso package possono accedere
 // all'oggetto
 OwnerPIN pin;
 // i parametri del pin
 final static byte PIN_TRY_LIMIT = (byte)3;
 final static byte MAX_PIN_SIZE = (byte)8;

 // I dati del libretto
 Dati dati;


// ------------------------------------------------------------------------
// COSTRUTTORE: il costruttore viene chiamato solo dalla routine install, da
// qui il modificatore di accesso protected (non e' private per fare in modo
// che le classi derivate possono chiamarlo).

    protected Libretto()
    {
        publicKey = new byte[LENGTH_PUBLIC_KEY];
        privateKey = new byte[LENGTH_PRIVATE_KEY];
    state = NO_KEYS;

    // Nota: bene crearlo subito il pin, la memoria potrebbe non essere
    // sufficiente poi dopo.
    pin = new OwnerPIN(PIN_TRY_LIMIT, MAX_PIN_SIZE);

    // Viene allocata anche la memoria per lo string buffer
    stringBuffer = new byte[MAX_IN_STRING];

    // Il repository dei dati
    dati = new Dati();

    // Si registra nel JCRE
```

```
        register();
    }



// ----------------------------------------------------------------------------
// INSTALL: viene chiamato dal JCRE nel momento dell'installazione. Crea
  // l'instanza
    public static void install(byte[] bArray, short bOffset, byte bLength)
    {
        new Libretto();
    }

// ----------------------------------------------------------------------------
// PROCESS: routine principale di che viene innescata alla chiamata di un
// comando apdu. Controlla il tipo di comando e passa il controllo alla
// routine opportuna.
    public void process(APDU apdu)
    {

   // Ottiene un riferimento al buffer dei dati APDU
   byte buffer[] = apdu.getBuffer();

   // L'interfaccia (il modo classico di tenere insieme dei valori
   // costanti in Java) contiene gli offset di ogni pezzetto del comando
   // APDU.
   // Siccome anche la chiamata di select viene passata all'applet si
   // filtra questo tipo di ingresso e si scarta
   if ((buffer[ISO7816.OFFSET_CLA] == 0) && (buffer[ISO7816.OFFSET_INS] ==
     (byte) 0xA4))
      return;


   // Verifica se il codice di CLA passato e' quello atteso, altrimenti
   // genera una eccezione che significa che l'istruzione mandata non e'
   // pertinente all'applet (questo significa una errata sequenza di
   // comandi APDU).
   if (buffer[ISO7816.OFFSET_CLA] != Libretto_CLA) {
    ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
   }

   // Sulla base del tipo di comando sceglie la procedura da eseguire
   switch (buffer[ISO7816.OFFSET_INS]) {
    case SET_PIN:
     setPin(apdu);
     return;
    case VER_PIN:
     verPin(apdu);
     return;
    case SET_PRIKEY:
```

```
     setPrivateKey(apdu);
     return;
    case SET_PUBKEY:
     setPublicKey(apdu);
     return;
    case GET_PUBKEY:
     getPublicKey(apdu);
     return;
    case SIGN:
     sign(apdu);
     return;
    case ADD_FREQUENCY:
     addFrequency(apdu);
     return;
    case ADD_PASSED_EXAM:
     addPassedExam(apdu);
     return;
    case GET_MEDIA:
     getMedia(apdu);
     return;
    default:
     ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
     return;
   }
  }

// ---------------------------------------------------------------------------
// SET_PIN : memorizza il pin nella carta. Se il pin e' gia' stato
// inizializzato e' necessario prima verificare che sia stato inserito
// precedentemente.


  void setPin(APDU apdu) {

   if (((state & HAVE_PIN) == HAVE_PIN) && ( ! pin.isValidated() ))
    ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);

   byte buffer[] = apdu.getBuffer();

   // Controlla se la lunghezza del pin e' corretta
   if (buffer[ISO7816.OFFSET_LC] > MAX_PIN_SIZE)
    ISOException.throwIt(ISO7816.SW_WRONG_DATA);

   // Inizia a leggere il campo dati
    byte byteRead = (byte) (apdu.setIncomingAndReceive());

   pin.update(buffer, ISO7816.OFFSET_CDATA, byteRead);

   // Aggiorna lo stato
   state = (byte) (state | HAVE_PIN);
```

```
  }


// ---------------------------------------------------------------------------
// VER_PIN : verifica il pin e imposta la carta con il pin verficato

 void verPin(APDU apdu) {

  // Controlla che la carta sia completamente inizializzata
  if (state != (HAVE_PUBKEY | HAVE_PRIKEY | HAVE_PIN))
   ISOException.throwIt(ISO7816.SW_COMMAND_NOT_ALLOWED);

  byte buffer[] = apdu.getBuffer();

  byte byteRead = (byte) (apdu.setIncomingAndReceive());

  if ( pin.check(buffer, ISO7816.OFFSET_CDATA, byteRead) == false)
   ISOException.throwIt(SW_VERIFICATION_FAILED);
 }



// ---------------------------------------------------------------------------
// SET_PRIKEY : memorizza il valore della chiave privata nalla smartcard. Si
// presume che questo metodo venga lanciato una sola volta per tutta la vita
// della smartcard.

  void setPrivateKey(APDU apdu) {

   // Se la chiave privata e' gia' stata memorizzata impedisce che sia
   // memorizzata di nuovo
   if ((state & HAVE_PRIKEY) == HAVE_PRIKEY)
    ISOException.throwIt(ISO7816.SW_COMMAND_NOT_ALLOWED);

   byte buffer[] = apdu.getBuffer();

   // Controlla se la chiave specificata e' della lunghezza corretta
   if (buffer[ISO7816.OFFSET_LC] != LENGTH_PRIVATE_KEY)
    ISOException.throwIt(ISO7816.SW_WRONG_DATA);

   // Ricava il numero di byte letti nel comando APDU
   short bytesRead = apdu.setIncomingAndReceive();
   // Posiziona il cursore all'inizio del buffer
   short echoOffset = (short)0;
   // Scorre il buffer e copia i dati nella chiave privata
   // Tutto questo perché non possiamo sapere a priori quanti dati ci
   // vengono mandati alla volta dal CAD
   while ( bytesRead > 0 ) {
           Util.arrayCopyNonAtomic(buffer,
```

```
        ISO7816.OFFSET_CDATA, privateKey, echoOffset, bytesRead);
            echoOffset += bytesRead;
            bytesRead = apdu.receiveBytes(ISO7816.OFFSET_CDATA);
      }

   // Aggiorna lo stato della carta
   state = (byte) (state | HAVE_PRIKEY);

  }

// -------------------------------------------------------------------------
// SET_PUBKEY : Memorizza la chiave pubblica nella carta. Anche in questo caso
// la chiave può essere memorizzata soltanto una volta.

  void setPublicKey(APDU apdu) {

   // Il codice e' esattamente identico a quello per scrivere la chiave
   // privata
   if ((state & HAVE_PUBKEY) == HAVE_PUBKEY)
    ISOException.throwIt(ISO7816.SW_COMMAND_NOT_ALLOWED);

   byte buffer[] = apdu.getBuffer();

   if (buffer[ISO7816.OFFSET_LC] != LENGTH_PUBLIC_KEY)
    ISOException.throwIt(ISO7816.SW_WRONG_DATA);

   short bytesRead = apdu.setIncomingAndReceive();
   short echoOffset = (short)0;
   while ( bytesRead > 0 ) {
    Util.arrayCopyNonAtomic(buffer, ISO7816.OFFSET_CDATA,
        publicKey, echoOffset, bytesRead);
        echoOffset += bytesRead;
        bytesRead = apdu.receiveBytes(ISO7816.OFFSET_CDATA);
      }

   state = (byte) (state | HAVE_PUBKEY);

  }


// -------------------------------------------------------------------------
// GET_PUBKEY : Restituisce la chiave pubblica. Non c'e' bisogno di verificare
// il PIN nella caso in cui si voglia dare la chiave pubblica in quanto non
// c'e' bisogno di garantirne la sicurezza.

  void getPublicKey(APDU apdu) {
   // Controlla se la carta è attiva e correttamente inizializzata
   if (state != (HAVE_PUBKEY | HAVE_PRIKEY | HAVE_PIN))
    ISOException.throwIt(ISO7816.SW_COMMAND_NOT_ALLOWED);
```

```
      apdu.setOutgoing();
   apdu.setOutgoingLength( (short) LENGTH_PUBLIC_KEY);
      apdu.sendBytesLong( publicKey, (short) 0, LENGTH_PUBLIC_KEY );
  }

// ------------------------------------------------------------------------------
// SIGN: Cripta con la chiave privata la stringa ricevuta in ingresso.

  void sign(APDU apdu) {

   // Controlla che la carta sia stata correttamente configurata
   if ( state != (HAVE_PUBKEY | HAVE_PRIKEY | HAVE_PIN) )
    ISOException.throwIt(ISO7816.SW_COMMAND_NOT_ALLOWED);

   // Controlla se il pin e' stato verificato
   if ( ! pin.isValidated() )
    ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);

   byte buffer[] = apdu.getBuffer();

   // Controlla che la stringa da criptare non sia più lunga del massimo
   // consentito
   if (buffer[ISO7816.OFFSET_LC] > MAX_IN_STRING)
    ISOException.throwIt(ISO7816.SW_WRONG_DATA);


   // Effettua la copia della stringa in ingresso nello stringBuffer
   short bytesRead = apdu.setIncomingAndReceive();
   short echoOffset = (short)0;
   while ( bytesRead > 0 ) {
    Util.arrayCopyNonAtomic(buffer, ISO7816.OFFSET_CDATA,
        stringBuffer, echoOffset, bytesRead);
        echoOffset += bytesRead;
        bytesRead = apdu.receiveBytes(ISO7816.OFFSET_CDATA);
      }
   // Cripta il contenuto dello stringBuffer
   RSAEncode();

   // Manda in output la stringa crittografata
      apdu.setOutgoing();
   apdu.setOutgoingLength( (short) echoOffset);
      apdu.sendBytesLong( stringBuffer, (short) 0, echoOffset );

  }

  // Questo metodo esegue la codifica RSA
  private void RSAEncode() {
  }
```

```
// ----------------------------------------------------------------------
// ADD_FREQUENCY: aggiunge la frequenza di un esame, il campo LC e' fatto
// così:
// | Codice Esam (2 byte) | Data (2 byte) | Firma (SIGN_SIZE byte) |


  void addFrequency(APDU apdu) {
   // Controlla che la carta sia stata correttamente configurata
   if ( state != (HAVE_PUBKEY | HAVE_PRIKEY | HAVE_PIN) )
    ISOException.throwIt(ISO7816.SW_COMMAND_NOT_ALLOWED);

   byte buf[] = apdu.getBuffer();
   apdu.setIncomingAndReceive();

   // Mi aspetto di trovare nella stringa 4 byte: due byte sono il codice
   // dell'esame e due byte sono la data di frequenza
   if (buf[ISO7816.OFFSET_LC] != (4 + dati.SIGN_SIZE))
    ISOException.throwIt(ISO7816.SW_WRONG_DATA);

   // Tratto due byte come codice esame
   dati.setExam(dati.FREQ,
    (short) (buf[ISO7816.OFFSET_CDATA]*0x100 + buf[ISO7816.OFFSET_CDATA +1]),
    (short)(buf[ISO7816.OFFSET_CDATA+2]*0x100+buf[ISO7816.OFFSET_CDATA +3]),
    (byte) 0,buf ,(byte) (ISO7816.OFFSET_CDATA + 4), dati.SIGN_SIZE);

   // DEBUG: mette sulla console lo stato del libretto
   System.out.println(dati);
  }


// ----------------------------------------------------------------------
// ADD_PASSED_EXAM: aggiunge un esame seguito così:
// | Codice Esam (2byte) | Data (2byte) | Voto (1byte) Firma (SIGN_SIZE byte)|


  void addPassedExam(APDU apdu) {
   // Controlla che la carta sia stata correttamente configurata
   if ( state != (HAVE_PUBKEY | HAVE_PRIKEY | HAVE_PIN) )
    ISOException.throwIt(ISO7816.SW_COMMAND_NOT_ALLOWED);

   byte buf[] = apdu.getBuffer();
   apdu.setIncomingAndReceive();

   // Mi aspetto di trovare nella stringa 4 byte: due byte sono il codice
   // dell'esame e due byte sono la data di frequenza
   if (buf[ISO7816.OFFSET_LC] != (5 + dati.SIGN_SIZE))
    ISOException.throwIt(ISO7816.SW_WRONG_DATA);

   // Tratto due byte come codice esame
```

```
   dati.setExam(dati.PASSED,
    (short) (buf[ISO7816.OFFSET_CDATA]*0x100 + buf[ISO7816.OFFSET_CDATA +1]),
    (short)(buf[ISO7816.OFFSET_CDATA+2]*0x100+buf[ISO7816.OFFSET_CDATA +3]),
    (byte) buf[ISO7816.OFFSET_CDATA+4],
    buf ,(byte) (ISO7816.OFFSET_CDATA + 5), dati.SIGN_SIZE);

   // DEBUG: mette sulla console lo stato del libretto
   System.out.println(dati);
  }




// ---------------------------------------------------------------------
// restituisce la media esami del proprietario moltiplicata per
// 100. Questo metodo e' pubblico perche' attraverso l'interfaccia shared è
// accessibile anche da applet al di fuori di questo contesto
  public short getMedia() {
   int total = 0;
   byte count = 0;
   for (byte i=0; i < Dati.MAX_EXAM; i++)
    if (dati.getFlag(i) == Dati.PASSED) {
       total = total + dati.getVote(i);
     count++;
    }
   total = total * 100;
   return (short) (total / count);
  }

// -------------------------------------------------------------------------
// GET_MEDIA: restituisce in risposta due byte che indicano la media dello
// studente moltiplicata per 100.
  void getMedia(APDU apdu) {
   if (state != (HAVE_PUBKEY | HAVE_PRIKEY | HAVE_PIN))
    ISOException.throwIt(ISO7816.SW_COMMAND_NOT_ALLOWED);


      apdu.setOutgoing();
   apdu.setOutgoingLength( (short) 2);
   short media = getMedia();
   byte[] mediaByte = new byte[2];
   mediaByte[0] = (byte) (media / 0x100);
   mediaByte[1] = (byte) (media % 0x100);
      apdu.sendBytesLong( mediaByte, (short) 0, (short) 2 );
  }

}
```

4.2 Codice di Dati.java

```
package infind;

import javacard.framework.*;


public class Dati {


 // Come viene allocato l'esame:
 // |FLAG|CODE|DATE|VOTE|SIGN                           |
 // |1   |2   |2   |1   |16                             |
 // offset dei dati nell'array
 static short FLAG_OFFSET = 0;
  static short CODE_OFFSET = 1;
 static short DATE_OFFSET = 3;
 static short VOTE_OFFSET = 5;
 static short SIGN_OFFSET = 6;

 // Altre costanti pubbliche
 public static byte SIGN_SIZE = 4;
 public static byte MAX_EXAM = 40;

 // costante calcolata
 static short ROW_SIZE = (short) (SIGN_OFFSET + SIGN_SIZE);

 // Flag utilizzati nel campo FLAG
 public static byte EMPTY = 0;
 public static byte FREQ = 1;
 public static byte PASSED = 2;

 // L'array usato come memoria
 private static byte[] mem;


 // Costruttore: alloca la memoria -----------------------------------------
 protected Dati() {
  if (mem == null) {
   mem = new byte[MAX_EXAM * ROW_SIZE];
   // inizializza a zero i flag
   for (byte i=0; i < MAX_EXAM; i++)
   mem[i * ROW_SIZE] = (byte) 0;
  }
 }

 // Metodi per l'accesso ai dati -------------------------------------------
 public byte getFlag(byte examNum) {

  // Se il record non esiste o e' vuoto ritorna un errore
```

```
 if ((examNum >= MAX_EXAM))
  ISOException.throwIt(ISO7816.SW_RECORD_NOT_FOUND);

 return mem[examNum * ROW_SIZE + FLAG_OFFSET];

}

public short getCode(byte examNum) {

 if ((examNum >= MAX_EXAM) ||
   (mem[examNum * ROW_SIZE + FLAG_OFFSET] == EMPTY))
  ISOException.throwIt(ISO7816.SW_RECORD_NOT_FOUND);

 return bytes2Short(mem[examNum * ROW_SIZE + CODE_OFFSET],
   mem[examNum * ROW_SIZE + CODE_OFFSET + 1]);
}

public short getDate(byte examNum) {
 if ((examNum >= MAX_EXAM) ||
   (mem[examNum * ROW_SIZE + FLAG_OFFSET] == EMPTY))
  ISOException.throwIt(ISO7816.SW_RECORD_NOT_FOUND);

 return bytes2Short(mem[examNum * ROW_SIZE + DATE_OFFSET],
   mem[examNum * ROW_SIZE + DATE_OFFSET + 1]);
}

public byte getVote(byte examNum) {
 if ((examNum >= MAX_EXAM) ||
   (mem[examNum * ROW_SIZE + FLAG_OFFSET] == EMPTY))
  ISOException.throwIt(ISO7816.SW_RECORD_NOT_FOUND);

 return mem[examNum * ROW_SIZE + VOTE_OFFSET];
}

public byte[] getSign(byte examNum) {
 if ((examNum >= MAX_EXAM) ||
   (mem[examNum * ROW_SIZE + FLAG_OFFSET] == EMPTY))
  ISOException.throwIt(ISO7816.SW_RECORD_NOT_FOUND);

 byte[] sign = new byte[SIGN_SIZE];

 Util.arrayCopyNonAtomic(mem, (short) (examNum * ROW_SIZE + SIGN_OFFSET),
  sign, (short) 0, SIGN_SIZE);

 return sign;
}


// Metodi di scrittura ----------------------------------------------------
```

```
void setExam(byte flag, short code, short date, byte vote,
  byte[] sign, byte start, byte length) {

 // Cerca uno spazio libero per l'esame
 byte examNum = 0;
 boolean find = false;

 while ( (examNum < MAX_EXAM) && (!find) ) {
  if (mem[examNum * ROW_SIZE + FLAG_OFFSET] == EMPTY) {
   find = true;
  } else
   examNum++;

 }

 // Controllo che l'array passato sia della lunghezza giusta prima di
 // memorizzare qualsiasi dato
 if (length != SIGN_SIZE)
  ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

 // Memorizza i dati
 mem[examNum * ROW_SIZE + FLAG_OFFSET] = flag;
 mem[examNum * ROW_SIZE + CODE_OFFSET] = short2HByte(code);
 mem[examNum * ROW_SIZE + CODE_OFFSET + 1] = short2LByte(code);
 mem[examNum * ROW_SIZE + DATE_OFFSET] = short2HByte(date);
 mem[examNum * ROW_SIZE + DATE_OFFSET + 1] = short2LByte(date);
 mem[examNum * ROW_SIZE + VOTE_OFFSET] = vote;
 Util.arrayCopyNonAtomic(sign,start, mem, (short) (examNum * ROW_SIZE +
    SIGN_OFFSET), SIGN_SIZE);
}

// Utility per la conversione da short in byte unsigned --------------------
public static short bytes2Short(byte high, byte low) {
 return (short) ((high + 0x80) * 0xFF + low + 0x80);
}

public static byte short2HByte (short val) {
 return (byte) (val / 0xFF - 0x80);
}

public static byte short2LByte (short val) {
 return (byte) (val % 0xFF - 0x80);
}

public String toString() {
 String out = "";
 for (byte i=0; i < MAX_EXAM -2; i++) {
  if (mem[ROW_SIZE*i + FLAG_OFFSET] != EMPTY) {
   out = out +  i + "- FLAG: " + getFlag(i) + " - CODE: " + getCode(i) +
    " DATE: " + getDate(i) + " VOTE: " + getVote(i)
```

```
     + "\n  SIGN: |";
    for (int j=0; j < SIGN_SIZE; j++)
     out = out + (char) (mem[ROW_SIZE*i + SIGN_OFFSET + j]) +"|";
    out = out +"\n";
   }
  }
  return out;
 }
}
```

4.3 Testing dell'applet

Segue la sequenza dei comandi APDU inviati:

```
powerup;

echo "Demo dell'applet Libretto---------------------------------------------";

echo "";
echo "1. Seleziona l'applet";
echo "   Risposta: 90 00 = SW_NO_ERROR";

// Per selezionare una applicazione si deve usare l'istruzione con CLA 0x00 e
// INS 0xA4. Nella parte dati si mette la stringa AID dell'applicazione.
// |CLA |INS |P1  |P2  |LC  |DATI                                      |LE  |
// |    |    |    |    |    |RID                   PIX                 |    |
    0x00 0xA4 0x04 0x00 0x0a 0xa0 0x0 0x0 0x0 0x62 0x3 0x1 0xc 0x6 0x1 0x7F;


echo "";
echo "2. Chiama un comando CLA non supportato dall'applet";
echo "   Risposta: 6e 00 = SW_CLA_NOT_SUPPORTED";

// Istruzioni con CLA != 0xB0 non sono supportate da questa applet
// CLA |INS |P1  |P2  |LC  | |LE  |
   0x20 0x20 0x00 0x00 0x00  0x7F;


echo "";
echo "3. Chiama un comando con un INS non supportato dall'applet";
echo "   Risposta: 6d 00 = SW_INS_NOT_SUPPORTED";
// Chiama un comando con INS non supportato dall'applet Libretto
// L'istruzione con INS = 0x10 non e' supportato
// CLA |INS |P1  |P2  |LC  | |LE  |
   0xB0 0x10 0x00 0x00 0x00  0x7F;


echo "";
echo "4. SET_PRIKEY con chiave di lunghezza sbagliata";
echo "   Risposta: 6a 90 = SW_WRONG_DATA";
```

```
// La chiave ha infatti solo 4 caratteri, mentre dovrebbe essere 8
// CLA |INS |P1  |P2  |LC  |                     |LE  |
   0xB0 0x20 0x00 0x00 0x04 0x20 0x20 0x20 0x20  0x7F;


echo "";
echo "5. SET_PRIKEY con chiave di lunghezza corretta";
echo "   Risposta: 90 00 = SW_NO_ERROR";
// CLA |INS |P1  |P2  |LC  |                                         |LE  |
   0xB0 0x20 0x00 0x00 0x08 0x20 0x20 0x20 0x20 0x20 0x20 0x20 0x20  0x7F;


echo "";
echo "6. SET_PRIKEY quando la chiave e' gia' stata settata";
echo "   Risposta: 69 86 = SW_WRONG_COMMAND";
// CLA |INS |P1  |P2  |LC  |                                         |LE  |
   0xB0 0x20 0x00 0x00 0x08 0x20 0x20 0x20 0x20 0x20 0x20 0x20 0x20  0x7F;


echo "";
echo "7. SET_PUBKEY con chiave di lunghezza corretta";
echo "   Risposta: 90 00 = SW_NO_ERROR";
// CLA |INS |P1  |P2  |LC  |                                         |LE  |
   0xB0 0x30 0x00 0x00 0x08 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08  0x7F;


echo "";
echo "8. SET_PIN imposta il pin con la sequenza 1234";
echo "   Risposta: 90 00 = SW_NO_ERROR";
// |CLA |INS |P1  |P2  |LC  |CDATA              |LE  |
    0xB0 0x10 0x00 0x00 0x04 0x01 0x02 0x03 0x04 0x7F;


echo "";
echo "Adesso la smartcard e' pronta per funzionare.";
echo "";


echo "";
echo "9. GET_PUBKEY: chiede la chiave pubblica (non serve il pin)";
echo "   Risposta: CDATA = chiave pubblica | 90 00 = SW_NO_ERROR";
// |CLA |INS |P1  |P2  |LC  | |LE  |
    0xB0 0x40 0x00 0x00 0x00   0x7F;


echo "";
echo "11. SIGN: chiede di firmare una stringa senza aver dato il pin";
echo "   Risposta: 63 01 = SW_PIN_VERIFICATION_REQUIRED ";
// |CLA |INS |P1  |P2  |LC  |                                                   |LE  |
    0xB0 0x50 0x00 0x00 0x0A 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A 0x7F;
```

```
echo "";
echo "12. VER_PIN: inserisce un pin sbagliato";
echo "  Risposta: 63 00 = SW_VERIFICATION_FAILED";
// |CLA |INS |P1  |P2  |LC  |CDATA              |LE  |
    0xB0 0x15 0x00 0x00 0x04 0x04 0x03 0x02 0x01  0x7F;


echo "";
echo "13. VER_PIN: inserisce il pin corretto";
echo "  Risposta: 90 00 = SW_NO_ERROR";
// |CLA |INS |P1  |P2  |LC  |CDATA              |LE  |
    0xB0 0x15 0x00 0x00 0x04 0x01 0x02 0x03 0x04  0x7F;


echo "";
echo "14. SIGN: chiede la firma ora che il pin e' inserito";
echo "   Risposta: CDATA = stringa criptata | 90 00 = SW_NO_ERROR";
// |CLA |INS |P1  |P2  |LC  |                                                  |LE  |
    0xB0 0x50 0x00 0x00 0x0A 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A 0x7F;

echo "";
echo "15. ADD_FREQ: aggiunge la frequenza di un esame 256 in data 512";
echo "  90 00 = SW_NO_ERROR";
// |CLA |INS |P1  |P2  |LC  |_______________________________________|LE  |
// |    |    |    |    |    | E_CODE  | E_DATE  | E_SIGN            |    |
    0xB0 0xB0 0x00 0x00 0x08 0x01 0x00 0x02 0x00 0x7E 0x7E 0x7E 0x7E 0x7F;

echo "";
echo "16. ADD_PAS_EX: imposta come sostenuto l'esame 258 in data 522, voto 25";
echo "  90 00 = SW_NO_ERROR";
// |CLA |INS |P1  |P2  |LC  |____________________________________________|LE  |
// |    |    |    |    |    | E_CODE  | E_DATE  |VOTE| E_SIGN            |    |
    0xB0 0xC0 0x00 0x00 0x09 0x01 0x02 0x02 0x0A 0x19 0x7E 0x7E 0x7E 0x7E 0x7F;

echo "";
echo "17. ADD_PAS_EX: imposta come sostenuto l'esame 260 in data 524, voto 29";
echo "  90 00 = SW_NO_ERROR";
// |CLA |INS |P1  |P2  |LC  |____________________________________________|LE  |
// |    |    |    |    |    | E_CODE  | E_DATE  |VOTE| E_SIGN            |    |
    0xB0 0xC0 0x00 0x00 0x09 0x01 0x04 0x02 0x0C 0x1D 0x7E 0x7E 0x7E 0x7E 0x7F;

echo "17. GET_MEDIA: restituisce la media dello studente * 100";
echo "  90 00 = SW_NO_ERROR | 0a 0c (2500)";
// |CLA |INS |P1  |P2  |LC  |LE  |
    0xB0 0xD0 0x00 0x00 0x00 0x7F;
```

```
// ---------------------------------------------------------------------------

// *** SCRIPT END ***
powerdown;
```

Questo è l'output della risposta fornita da *apdutool*:

```
Java Card ApduTool (version 0.15)
Copyright (c) 2001 Sun Microsystems, Inc. All rights reserved.
Opening connection to localhost on port 9025.
Connected.
Received ATR = 0x3b 0xf0 0x11 0x00 0xff 0x00
Demo dell'applet Libretto---------------------------------------------

1. Seleziona l'applet
   Risposta: 90 00 = SW_NO_ERROR
CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 0a, a0, 00, 00, 00, 62, 03, 01, 0c, 06, 01
, Le: 00, SW1: 90, SW2: 00

2. Chiama un comando CLA non supportato dall'applet
   Risposta: 6e 00 = SW_CLA_NOT_SUPPORTED
CLA: 20, INS: 20, P1: 00, P2: 00, Lc: 00, Le: 00, SW1: 6e, SW2: 00

3. Chiama un comando con un INS non supportato dall'applet
   Risposta: 6d 00 = SW_INS_NOT_SUPPORTED
CLA: b0, INS: 10, P1: 00, P2: 00, Lc: 00, Le: 00, SW1: 6a, SW2: 80

4. SET_PRIKEY con chiave di lunghezza sbagliata
   Risposta: 6a 90 = SW_WRONG_DATA
CLA: b0, INS: 20, P1: 00, P2: 00, Lc: 04, 20, 20, 20, 20, Le: 00, SW1: 6a, SW2:
80

5. SET_PRIKEY con chiave di lunghezza corretta
   Risposta: 90 00 = SW_NO_ERROR
CLA: b0, INS: 20, P1: 00, P2: 00, Lc: 08, 20, 20, 20, 20, 20, 20, 20, 20, Le: 00
, SW1: 90, SW2: 00

6. SET_PRIKEY quando la chiave e' gia' stata settata
   Risposta: 69 86 = SW_WRONG_COMMAND
CLA: b0, INS: 20, P1: 00, P2: 00, Lc: 08, 20, 20, 20, 20, 20, 20, 20, 20, Le: 00
, SW1: 69, SW2: 86

7. SET_PUBKEY con chiave di lunghezza corretta
   Risposta: 90 00 = SW_NO_ERROR
CLA: b0, INS: 30, P1: 00, P2: 00, Lc: 08, 01, 02, 03, 04, 05, 06, 07, 08, Le: 00
, SW1: 90, SW2: 00

8. SET_PIN imposta il pin con la sequenza 1234
   Risposta: 90 00 = SW_NO_ERROR
```

```
CLA: b0, INS: 10, P1: 00, P2: 00, Lc: 04, 01, 02, 03, 04, Le: 00, SW1: 90, SW2:
00

Adesso la smartcard e' pronta per funzionare.


9. GET_PUBKEY: chiede la chiave pubblica (non serve il pin)
   Risposta: CDATA = chiave pubblica | 90 00 = SW_NO_ERROR
CLA: b0, INS: 40, P1: 00, P2: 00, Lc: 00, Le: 08, 01, 02, 03, 04, 05, 06, 07, 08
, SW1: 90, SW2: 00

11. SIGN: chiede di firmare una stringa senza aver dato il pin
   Risposta: 63 01 = SW_PIN_VERIFICATION_REQUIRED
CLA: b0, INS: 50, P1: 00, P2: 00, Lc: 0a, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0a
, Le: 00, SW1: 63, SW2: 01

12. VER_PIN: inserisce un pin sbagliato
  Risposta: 63 00 = SW_VERIFICATION_FAILED
CLA: b0, INS: 15, P1: 00, P2: 00, Lc: 04, 04, 03, 02, 01, Le: 00, SW1: 63, SW2:
00

13. VER_PIN: inserisce il pin corretto
  Risposta: 90 00 = SW_NO_ERROR
CLA: b0, INS: 15, P1: 00, P2: 00, Lc: 04, 01, 02, 03, 04, Le: 00, SW1: 90, SW2:
00

14. SIGN: chiede la firma ora che il pin e' inserito
   Risposta: CDATA = stringa criptata | 90 00 = SW_NO_ERROR
CLA: b0, INS: 50, P1: 00, P2: 00, Lc: 0a, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0a
, Le: 0a, 01, 02, 03, 04, 05, 06, 07, 08, 09, 0a, SW1: 90, SW2: 00

15. ADD_FREQ: aggiunge la frequenza di un esame 256 in data 512
  90 00 = SW_NO_ERROR
CLA: b0, INS: b0, P1: 00, P2: 00, Lc: 08, 01, 00, 02, 00, 7e, 7e, 7e, 7e, Le: 00
, SW1: 90, SW2: 00

16. ADD_PAS_EX: imposta come sostenuto l'esame 258 in data 522, voto 25
  90 00 = SW_NO_ERROR
CLA: b0, INS: c0, P1: 00, P2: 00, Lc: 09, 01, 02, 02, 0a, 19, 7e, 7e, 7e, 7e, Le
: 00, SW1: 90, SW2: 00

17. ADD_PAS_EX: imposta come sostenuto l'esame 260 in data 524, voto 29
  90 00 = SW_NO_ERROR
CLA: b0, INS: c0, P1: 00, P2: 00, Lc: 09, 01, 04, 02, 0c, 1d, 7e, 7e, 7e, 7e, Le
: 00, SW1: 90, SW2: 00
17. GET_MEDIA: restituisce la media dello studente * 100
  90 00 = SW_NO_ERROR | 0a 0c (2500)
CLA: b0, INS: d0, P1: 00, P2: 00, Lc: 00, Le: 02, 0a, 8c, SW1: 90, SW2: 00
```

Bibliografia

[1] Sun - *Java Card 2.1.1 Virtual Machine Specification*

[2] Sun - *Java Card 2.1.1 Application Programming Interface*

[3] Sun - *Java Card 2.1.1 Development kit User's Guide*

[4] D.B.Everett - *Smart Card Technology: Introduction To Smart Cards* - http://www.smartcard.co.uk

[5] Z. Chen - *How to Write a Java Card applet: A developer's guide* - Javaworld, Luglio 1999

[6] Rinaldo di Giorgio - *Get a jumpstart on the Java Card* - Javaworld, Febbraio 1998

[7] O. Fodor and V. Hassler - *Javacard and Opencard Framework: A Tutorial*

[8] O. Carre, H. Martin, JJ Vandewalle - *A semiformal model of Javacard 2.1 in UML*

[9] AIPA (Gruppo di lavoro sulla Carta di identità Elettronica) - *Il Processo di Autenticazione* - Roma, 2 Febbraio 2000

[10] Bruce Eckel - *Thinking in Java 2nd Edition* - Prentice Hall, 2000