UNIVERSITA' DEGLI STUDI DI PISA

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

SCOTT:
un'architettura modulare per il testing di soluzioni basate su smartcard.

Relatori:

Prof. Paolo Ancilotti

Prof. Giuseppe Anastasi

Dott. Tommaso Cucinotta

Candidato:

Andrea Angella

Anno accademico: **2009/2010**

Data di laurea: **8 ottobre 2009**

Ai miei genitori

e alla mia sorellina Laura

per il loro costante sostegno

e per i sacrifici fatti

affinché potessi raggiungere,

con serenità,

questo importante traguardo.

Ringraziamenti

Desidero ringraziare in modo particolare il Dott. Tommaso Cucinotta per la sua costante presenza e disponibilità durante tutto il lavoro di tesi e per le piacevoli discussioni tecniche su argomenti anche non strettamente legati all'attività accademica. Un sentito grazie ai ragazzi della mailing list di Muscle card, in particolare a Douglas E. Engert, Dr. Ludovic Rousseau e Thomas Harning Jr che hanno risposto prontamente alle mie domande sulle smart card.

Un importante aiuto è arrivato dai ragazzi dei newsgroup *it.comp.lang.c++* e *it.comp.os.linux.development* grazie ai quali ho risolto spinose problematiche di programmazione. Ringrazio anche Franco Failli per il libro "Come scrivere e discutere una tesi di laurea in ingegneria" che è stato un ottimo alleato in tutte le fasi della mia prova finale. Infine, ultimo ma non per importanza, il ringraziamento alla mia ragazza Michela che negli ultimi mesi è stata molto paziente ed ha condiviso con me fatiche, rinunce e malumori.

Sommario

Negli ultimi anni si è assistito a un incredibile aumento del numero di smartcard prodotte sul mercato; è sufficiente pensare alla quantità di SIM card presenti nel mondo per farsi un'idea del volume complessivo di vendite che nel solo anno 2008 ha raggiunto un valore superiore ai cinque miliardi di unità. Nonostante la definizione di un numero significativo di standard per regolare l'utilizzo e il funzionamento di questi dispositivi, lo sviluppo di soluzioni basate su smartcard è ancora notevolmente complesso. Una delle cause principali di questa complessità è la difficoltà nel testare le proprie applicazioni in modo automatico che è dovuta principalmente all'assenza di strumenti adatti a tale scopo.

La maggior parte degli strumenti disponibili sono proprietari e utilizzabili solamente attraverso un'interfaccia grafica che rende di fatto impossibile l'automatizzazione dei test. I pochi strumenti che lo consentono, tuttavia, sono fortemente non interoperabili per ragioni puramente commerciali. Nell'ambito di questa tesi è stato progettato ed implementato un toolkit chiamato SCOTT (Smart Card Open Test Toolkit) che ha l'obiettivo ambizioso di diventare lo strumento di riferimento per supportare il testing automatico di soluzioni basate su smartcard. Il fatto di essere open source, liberamente utilizzabile, con un'architettura modulare e completamente espandibile attraverso plugin, permetterà a SCOTT di migliorare nel tempo le proprie funzionalità e supportare un numero sempre maggiore di dispositivi e standard.

Indice dei contenuti

SOMMARIO **1**

INDICE DEI CONTENUTI **3**

INDICE DELLE FIGURE **7**

INDICE DELLE TABELLE **11**

INTRODUZIONE **12**

1. IL MONDO DELLE SMART CARD **14**

1.1 Che cos'è una smart card 14

1.2 Principali tipologie di smart card 15

1.3 Architettura interna di una smart card 17

1.4 Architettura di un sistema basato su smartcard 19

1.5 Principali soluzioni basate su smart card 21

1.6 Gli standard esistenti 23

1.6.1 La famiglia di standard ISO-7816 ..... *24*

1.6.2 Lo standard PC/SC *25*

1.7 Problematiche nello sviluppo di applicazioni 28

1.7.1 Mancanza d'interoperabilità tra smartcard *28*

1.7.2 Interfacce poco intuitive *29*

1.7.3 Mancanza di strumenti per il testing e il debugging *30*

1.8 Strumenti open source attualmente disponibili 31

1.8.1 Smartsh *32*

1.8.2 Muscle Tool e XCardii *32*

1.8.3 OpenSC Tools *33*

1.8.4 GPShell *33*

1.8.5 Limiti di questi strumenti *34*

2. INTRODUZIONE A SCOTT **35**

2.1 Che cos'è SCOTT ? 35

2.2 Requisiti di progettazione 36

2.3 Architettura software 37

2.4 Elenco dettagliato delle funzionalità 39

2.5 Modalità di utilizzo di SCOTT 42

3. GUIDA ALL'UTILIZZO DI SCOTT **43**

3.1 Il sistema di tipi 43

3.1.1 Tipi di dato primitivi 44
3.1.2 Il tipo SEnum 44
3.1.3 Il tipo SVector 45
3.1.4 Il tipo SStruct 46
3.2 La Shell SCOTT 47
3.2.1 Visualizzazione dei plugin installati 49
3.2.2 Caricamento dei plugin 49
3.2.3 Visualizzazione dei tipi disponibili 50
3.2.4 Creazione ed utilizzo delle variabili 52
3.2.5 Visualizzazione dei comandi disponibili 55
3.2.6 Esecuzione dei comandi di un plugin 57
3.2.7 Meccanismo delle variabili automatiche 58
3.2.8 Meccanismo delle variabili implicite 59
3.2.9 Visualizzazione automatica dei risultati 60
3.2.10 Visualizzazione delle impostazioni della shell 61
3.2.11 Stampa ed esportazione dei dati su file 61
3.2.12 Esecuzione automatica di comandi all'avvio 67

4. GUIDA ALL'UTILIZZO DEI PLUGIN DI SCOTT 69

4.1 Il plugin "pcsc" 69
4.1.1 Tipi definiti dal plugin "pcsc" 69
4.1.2 Il comando EstablishContext() 70
4.1.3 Il comando ListReaders() 71
4.1.4 Il comando Connect() 71
4.1.5 Il comando Status() 72
4.1.6 Il comando Transmit() 73
4.1.7 Il comando Disconnect() 74
4.1.8 Il comando ReleaseContext() 75
4.1.9 Un esempio completo di utilizzo 75
4.2 Il plugin "iso7816-4" 77
4.2.1 Tipi definiti dal plugin "iso7816-4" 80
4.2.2 Il comando Trasmit0() 80
4.2.3 Il comando Select() 81
4.2.4 Il comando GetResponse() 82
4.2.5 Il comando Verify() 83
4.2.6 Il comando ReadBinary() 83
4.2.7 Il comando UpdateBinary() 84
4.2.8 Un esempio completo di utilizzo 85
4.3 Il plugin "cryptoflex8" 88

4.3.1 Tipi definiti dal plugin "cryptoflex8" ... *88*
4.3.2 Il comando ChangeCHV() ... *91*
4.3.3 Il comando CreateFile() ... *92*
4.3.4 Il comando CreateDF() ... *92*
4.3.5 Il comando CreateEF() ... *93*
4.3.6 Il comando DeleteFile() ... *94*
4.3.7 Il comando DirNext() ... *94*
4.3.8 Il comando GetFilesInfo() ... *97*
4.3.9 Il comando GetResponse() ... *99*
4.3.10 Il comando Invalidate() ... *100*
4.3.11 Il comando ReadBinary() ... *100*
4.3.12 Il comando Rehabilitate() ... *101*
4.3.13 Il comando RetrieveCLA() ... *101*
4.3.14 Il comando Select() ... *102*
4.3.15 Il comando SelectDF() ... *102*
4.3.16 Il comando SelectEF() ... *103*
4.3.17 Il comando UnblockCHV() ... *104*
4.3.18 Il comando UpdateBinary() ... *104*
4.3.19 Il comando VerifyCHV() ... *105*
4.3.20 Il comando VerifyKey() ... *105*
4.3.21 Un esempio completo di utilizzo ... *106*

5. PROGETTAZIONE ED IMPLEMENTAZIONE DI SCOTT ... **110**

5.1 Metodologia di sviluppo e scelte tecnologiche ... 110
5.1.1 Linguaggio di programmazione ... *110*
5.1.2 Ambienti e strumenti di sviluppo ... *111*
5.1.3 Librerie di supporto ... *112*
5.1.4 Testing ... *113*
5.1.5 Documentazione del prodotto ... *114*
5.1.6 Licenza del prodotto ... *114*
5.1.7 Apparecchiatura usata ... *115*
5.2 Considerazioni preliminari e convenzioni ... 115
5.3 Architettura del sistema di tipi ... 117
5.3.1 Tipi semplici ... *118*
5.3.2 Tipi complessi ... *120*
5.4 Il meccanismo dei comandi ... 124
5.5 Il repository dei descrittori ... 127
5.6 Il sistema di risoluzione dei nomi ... 128
5.7 La risoluzione delle espressioni ... 129

5.8 La gestione delle eccezioni 130
5.9 L'interfaccia della SCOTT Core Library 131
5.10 Come ottenere il codice sorgente di SCOTT 131

6. ESTENDERE SCOTT 132

6.1 Come installare un nuovo plugin 132
6.2 Descrizione del plugin math 132
6.3 Il descrittore di plugin 133
6.3.1 Lo schema XSD 134
6.3.2 Specifica delle dipendenze 137
6.3.3 Definizione dei tipi di dato 137
6.3.4 Definizione dei comandi 139
6.3.5 Il descrittore del plugin math 140
6.4 Scrivere il codice del nuovo plugin 141
6.4.1 La classe Plugin 141
6.4.2 SCOTT Plugin Creator 144
6.4.3 Analisi dei file autogenerati 144
6.4.4 Implementazione del corpo dei comandi 148
6.5 Utilizzare il nuovo plugin 150

7. CONCLUSIONI E SVILUPPI FUTURI 151

APPENDICE A - LA SMARTCARD CRYPTOFLEX 8K 154

TABELLA DEGLI ACRONIMI 157

BIBLIOGRAFIA 159

Indice delle figure

Figura 1 - Un esempio di smartcard. 14
Figura 2 - Classificazione delle smart card in base alle potenzialità 15
Figura 3 - Classificazione delle smart card in base all’interfacciamento 16
Figura 4 - Lettore di Contact Smartcard. 17
Figura 5 - Lettore di Contactless Smartcard. 17
Figura 6 - Architettura interna di una smart card 18
Figura 7 - Componenti hardware di un semplice sistema basato su smart card.. 19
Figura 8 - Architettura logica di un sistema basato su smart card 20
Figura 9 - Architettura definita dallo standard PC/SC 2.0 26
Figura 10 - Architettura software di SCOTT 38
Figura 11 - Tipi di dato in SCOTT 43
Figura 12 - La Shell SCOTT all'avvio 48
Figura 13 - Elenco dei comandi disponibili 48
Figura 14 - Maggiori informazioni sui comandi di shell 49
Figura 15 - Esempio di utilizzo del comando 'plugins' 49
Figura 16 - Caricamento di un plugin 50
Figura 17 - Caricamento di un plugin con dipendenze 50
Figura 18 - Comando 'types' 51
Figura 19 - Fully Qualified Type Name 51
Figura 20 - Dettagli di un tipo 52
Figura 21 - Sintassi del comando ‘var’ 52
Figura 22 - Creazione di variabili di tipo semplice 53
Figura 23 - Comando ‘vars’ 53
Figura 24 - Comando ‘show’ 54
Figura 25 - Creare variabili vettoriali e accesso ai membri di un vettore 54
Figura 26 - Creazione di strutture ed espressioni complesse 55
Figura 27 - Comando 'commands' 55
Figura 29 - Ottenere i dettagli di un comando di plugin 56
Figura 28 - Fully Qualified Command Name 56

Figura 30 - Sintassi per eseguire un comando di plugin 57
Figura 31 - Esempi di esecuzione comandi 58
Figura 32 - Uso del meccanismo delle variabili automatiche 59
Figura 33 - Meccanismo delle variabili automatiche ignorato 59
Figura 34 - Esempio di utilizzo delle variabili implicite 60
Figura 35 - Visualizzazione dei risultati con dumpvars a false 60
Figura 36 - Visualizzazione dei risultati con dumpvars a true.............................. 61
Figura 37 - Comando ‘settings’ 61
Figura 39 - Uso banale di ‘write’ 62
Figura 38 - Sintassi del comando ‘write’ 62
Figura 40 - Stampa di tipi primitivi 63
Figura 41 - Stampa di tipi complessi 63
Figura 42 - Sessione di esportazione dati 64
Figura 43 - Risultato dell'esportazione dati (file out.txt) 64
Figura 44 - File di configurazione della shell 67
Figura 45 - Avvio della shell con file di configurazione 68
Figura 46 - Comando pcsc.EstablishContext().. 71
Figura 47 - Comando pcsc.ListReaders() 71
Figura 48 - Comando pcsc.Connect() 72
Figura 49 - Comando pcsc.Status()... 73
Figura 50 - Comando pcsc.Transmit() 74
Figura 51 - Comando pcsc.Disconnect()... 74
Figura 52 - Comando pcsc.ReleaseContext() 75
Figura 53 - Riepilogo comandi pcsc.. 76
Figura 54 - Esecuzione del riepilogo di comandi pcsc.. 76
Figura 55 - Messaggio di input secondo il protocollo T=0 78
Figura 56 - Messaggio di Output secondo il protocollo T=0 78
Figura 57- Utilizzo del comando iso7816-4.Transmit0() 81
Figura 58 - Esempio di uso del comando iso7816-4.Select() 81
Figura 59 - Esempio di uso del comando iso7816-4.GetResponse() 82
Figura 60 - Esempio di uso del comando iso7816-4.Verify() 83
Figura 61 - Esempio di uso del comando iso7816-4.ReadBinary() 84

Figura 62 - Esempio di uso del comando iso7816-4.UpdateBinary() 84
Figura 63 - Riepilogo comandi iso7816-4 86
Figura 64 - Esecuzione del riepilogo comandi iso7816-4 87
Figura 65 - Comando cryptoflex8.ChangeCHV() 91
Figura 66 - Comando cryptoflex8.CreateFile() 92
Figura 67 - Comando cryptoflex8.CreateDF() 93
Figura 68 - Comando cryptoflex8.CreateEF() 93
Figura 69 - Comando cryptoflex8.DeleteFile() 94
Figura 70 - Comando cryptoflex8.Invalidate() 100
Figura 71 - Comando Rehabilitate() 101
Figura 72 - Comando cryptoflex8.RetrieveCLA() 102
Figura 73 - Comando cryptoflex8.SelectDF() 103
Figura 74 - Comando cryptoflex8.SelectEF() 103
Figura 75 - Comando cryptoflex8.UnblockCHV() 104
Figura 76 - Comando cryptoflex8.VerifyKey() 105
Figura 77 - Esecuzione della batteria di test per SCOTT 114
Figura 78 - Lettore “Todos Argos Mini II” e smart card “Cryptoflex 8K” 115
Figura 79 - Creazione di smart pointer senza utilizzare alias 116
Figura 80 - Macro CREATE_ALIAS 117
Figura 81 - Macro CREATE_MAP_ALIAS 117
Figura 82 - Creazione di smart pointer utilizzando alias 117
Figura 83 - Gerarchia delle classi per i tipi semplici 119
Figura 84 - Creazione di tipi semplici via codice 119
Figura 85 - Gerarchia delle classi per i tipi complessi 120
Figura 86 - Gerarchia dei Type Descriptor 121
Figura 87 - Gerarchia dei TypeDescriptorEntry 121
Figura 88 - Creazione e utilizzo di una variabile di tipo SEnum 123
Figura 89 - Creazione e utilizzo di una variabile di tipo SVector 123
Figura 90 - Creazione e utilizzo di una variabile di tipo SStruct 124
Figura 91 - La classe Command 125
Figura 92 - La classe CommandDescriptor 125
Figura 93 - Gerarchia dei Parameter 125

Figura 94 - Creazione e utilizzo di un comando via codice 127
Figura 95 - Classe DescriptorsRepository.. 128
Figura 96 - Classi per la risoluzione dei nomi 129
Figura 97 - Classe ExpressionResolutor.. 130
Figura 98 - Gerarchia delle eccezioni di SCOTT 130
Figura 99 - Classe Scott 131
Figura 100 - Posizione dei quadranti in ‘math’ 133
Figura 101 - Struttura base del descrittore di plugin. 136
Figura 102 - Sezione <Dependencies> nel descrittore di plugin........................ 137
Figura 103 - Sezione <Enums> nel descrittore di plugin 138
Figura 104 - Sezione <Vectors> nel descrittore di plugin 138
Figura 105 - Sezione <Structs> nel descrittore di plugin 139
Figura 106 - Sezione <Commands> nel descrittore di plugin 139
Figura 107 - Classe Plugin.. 142
Figura 108 - Utilizzo di SCOTT Plugin Creator. 144
Figura 109 - Utilizzo del plugin 'math'.. 150

Indice delle Tabelle

Tabella 1 - Numero di smart card (in milioni di unità) vendute nel 2008 a livello mondiale. (fonte Eurosmart) ... 22
Tabella 2 - Tipi di dato primitivi ... 44
Tabella 3 - Esempi di SVector con elementi primitivi ... 45
Tabella 4 - Esempi di SVector con elementi complessi ... 46
Tabella 5 - Definizione di un tipo SStruct ... 46
Tabella 6 - Modalità di valorizzazione di un tipo SStruct ... 47
Tabella 7 - Sequenze di escape supportate dal comando write ... 64
Tabella 8 - Specificatori di formato per SByte ... 65
Tabella 9 - Specificatori di formato per SNumber ... 65
Tabella 10 - Specificatori di formato per SChar ... 65
Tabella 11 - Specificatori di formato per SString ... 66
Tabella 12 - Specificatori di formato per SEnum ... 66
Tabella 13 - Specificatori di formato per SStruct ... 66
Tabella 14 - Specificatori di formato per SVector ... 66
Tabella 15 - Principali comandi definiti dallo standard ISO7816-4 ... 79
Tabella 16 - Classificazione delle Stutus Word ... 80
Tabella 17 - Membri della struttura FileInfo ... 89
Tabella 18 - Membri della struttura DirInfo ... 90
Tabella 19 - Membri della struttura DirSelectInfo ... 91
Tabella 20 - Identificatori dei file riservati ... 154
Tabella 21 - Comandi della scheda Cryptoflex 8K ... 155
Tabella 22 - Access Condition della scheda Cryptoflex 8K ... 156

Introduzione

Nell'ambito di questa tesi è stato progettato ed implementato un toolkit di nome SCOTT (Smart Card Open Test Toolkit) per supportare il testing di soluzioni basate su smart card. L'idea di realizzare questo progetto deriva essenzialmente dall'esperienza del mio relatore Tommaso Cucinotta che, in quanto esperto di sicurezza e sviluppo su smartcard, sentiva la necessità di uno strumento che permettesse di semplificare ed automatizzare le attività di testing. Di seguito è riportato brevemente il contenuto di ciascun capitolo per orientare la lettura e fornire un quadro di riferimento riguardo agli argomenti trattati.

Il Capitolo 1 è introduttivo al mondo delle smartcard. Si discute delle varie tipologie di smartcard e dell'architettura dei sistemi basati su di esse. E' presente una rapida elencazione delle principali soluzioni basate su smartcard. Infine si presentano due dei principali standard esistenti e cioè il PC/SC e l'ISO7816. Il capitolo termina con un'analisi delle principali problematiche nello sviluppo di soluzioni basate su smartcard anticipando le motivazioni che hanno portato alla creazione di un toolkit come SCOTT.

A partire dal Capitolo 2 si entra nel merito del progetto SCOTT. Il capitolo vuole spiegare ad alto livello le motivazioni che hanno portato a SCOTT elencando i requisiti di progettazione principali su cui il progetto ha fondato le sue radici. Dopo la presentazione dell'architettura generale si riporta un elenco dettagliato delle funzionalità del prodotto che saranno discusse in maniera più approfondita in tutti i paragrafi successivi. Il capitolo termina analizzando i differenti modi in cui si possono utilizzare gli strumenti offerti da SCOTT.

Il Capitolo 3 rappresenta la vera e propria guida dell'utente di SCOTT. Dopo aver presentato in maniera dettagliata il sistema dei tipi, si entra nel merito di come utilizzare la shell. Per ogni comando o funzionalità è riportata la sintassi e una descrizione, accompagnata da uno o più esempi di utilizzo.

Il Capitolo 4 rappresenta una guida per utilizzare i tre plugin che sono stati realizzati in questa tesi. Il primo plugin, di nome “pcsc”, permette di gestire la comunicazione a basso livello con le smartcard in accordo allo standard PC/SC. Il secondo plugin, chiamato “iso7816-4”, implementa parzialmente alcuni comandi definiti nella parte 4 dello standard ISO7816. Infine il plugin “cryptoflex8” implementa molti dei comandi specifici della scheda Cryptoflex 8K fornendo un’interfaccia di alto livello estremamente semplice da utilizzare. Per ogni comando è riportata la sintassi e una descrizione, accompagnata da uno o più esempi di utilizzo.

Il Capitolo 5 scende all’interno dell’architettura software del progetto toccando aspetti tecnici relativi al come sono state realizzate le varie funzionalità offerte. All’inizio si descrive la metodologia di sviluppo seguita e sono motivate le varie scelte tecnologiche che sono state prese; poi, dopo una presentazione delle convenzioni stabilite per la scrittura del codice, sono mostrate le principali classi che implementano tutte le funzionalità ampiamente descritte, a livello utente, nel capitolo 3. Per le classi più importanti sono mostrati alcuni esempi di utilizzo tramite codice. Tali conoscenze risultano indispensabili per chi desidera entrare nel merito della implementazione di plugin per SCOTT.

Il Capitolo 6 spiega com’è possibile scrivere un plugin per estendere le funzionalità di SCOTT. Partendo da come s’installa un nuovo plugin, è descritta nel dettaglio la struttura del descrittore di plugin per arrivare a spiegare come implementare il corpo dei comandi del nuovo plugin. Infine è mostrato l’utilizzo dello strumento “SCOTT Plugin Creator” per facilitare la fase di scrittura del codice ed aumentare la produttività durante lo sviluppo.

Il Capitolo 7 termina spiegando i risultati del lavoro svolto sia da un punto di vista tecnico che personale. Inoltre sono discusse le possibili evoluzioni del progetto nel prossimo futuro.

1. Il mondo delle Smart Card

1.1 Che cos'è una smart card

La smart card è un dispositivo hardware, di dimensioni estremamente ridotte, capace di memorizzare ed elaborare informazioni mediante un circuito integrato incapsulato in un supporto di plastica. (Figura 1).

Figura 1 - Un esempio di smartcard.

Una smart card:

- Può partecipare in una transazione elettronica automatica.
- È usata principalmente per aggiungere sicurezza.
- Non può essere facilmente rubata o copiata.
- È in grado di memorizzare dati in maniera sicura.
- Può essere in grado di eseguire un insieme di algoritmi crittografici.

Grazie a queste caratteristiche una smart card si propone in primo luogo come strumento informatico d'identificazione sicura e certificata degli individui e in secondo luogo come dispositivo di elaborazione a supporto della crittografia.

La smart card è stata inventata e brevettata nei primi anni settanta. Le prime applicazioni risalgono agli inizi degli anni ottanta con l'introduzione delle carte telefoniche prepagate e delle carte bancarie di credito/debito ad alta sicurezza

soprattutto in Francia e Germania. Il vero boom nelle vendite di questi dispositivi si è avuto con l'introduzione delle SIM card nei sistemi di telefonia mobile GSM e recentemente UMTS.

Negli ultimi anni le nuove tecniche di miniaturizzazione, mediante le quali è stato possibile produrre microchip sempre più piccoli a costi sempre più bassi, hanno consentito di realizzare smart card più potenti dotate di coprocessore crittografico e di buone capacità di memoria.

1.2 Principali tipologie di smart card

La Figura 2 mostra una classificazione delle smart card fatta sulla base delle potenzialità e delle caratteristiche del microchip. Esistono smart card a sola memoria (**Memory Card**) e smart card a microprocessore (**Microprocessor Card**).

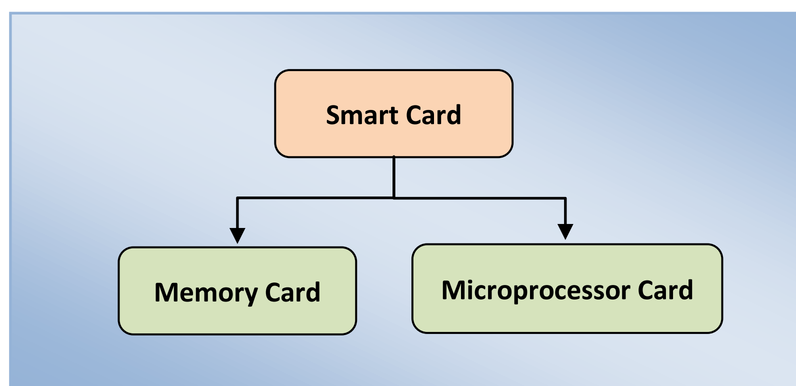


Figura 2 - Classificazione delle smart card in base alle potenzialità

Le **memory card** sono smart card che offrono solamente un supporto per la memorizzazione e la lettura dei dati. Tecnologicamente sono dispositivi molto semplici ed economici, offrono qualche kilobyte di memoria e sono utilizzati principalmente per realizzare applicazioni dove non è richiesta una "forte" protezione dei dati.

Le **microprocessor card** sono smart card che possiedono un microprocessore in grado di elaborare le informazioni contenute al loro interno. Su queste schede è

installato un sistema operativo che offre un certo numero di funzionalità. In particolare sono presenti funzioni per la lettura e scrittura in memoria, funzioni per impostare i permessi di accesso, funzioni crittografiche, ecc.

La necessità di implementare servizi più complessi, le maggiori richieste di memoria e di sicurezza e la necessità di elaborazione direttamente sulla scheda hanno contribuito nel tempo alla sempre più rapida diffusione delle microprocessor card rispetto alle memory card.

Un'altra classificazione delle smart card può invece essere fatta sulla base del tipo d'interfaccia di collegamento. Dalla Figura 3 si possono distinguere smart card con contattiera (**Contact Smartcard**), smart card con antenna (**Contactless Smartcard**) e smart card con antenna e contattiera (**Dual Interface Smartcard**).

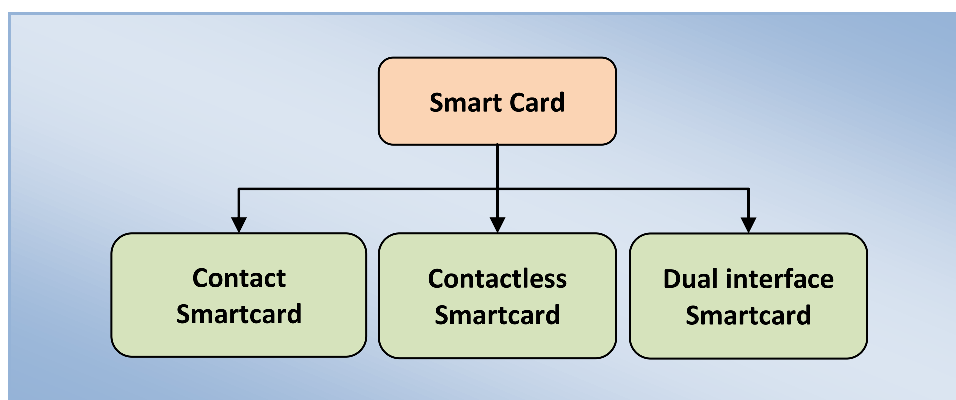


Figura 3 - Classificazione delle smart card in base all'interfacciamento

Le **Contact Smartcard** sono le più diffuse. Tramite una contattiera ricevono l'alimentazione e dialogano con l'esterno solamente dopo che sono state inserite in un apposito terminale chiamato lettore di smart card (Figura 4).

Le **Contactless Smartcard**, invece, comunicano con l'esterno utilizzando un'antenna quando sono in prossimità di un opportuno dispositivo di lettura/scrittura (Figura 5). Sono preferite in contesti in cui sono necessarie transazioni semplici e veloci, come sistemi di controllo dell'accesso in edifici ed aree riservate.

Le **Dual Interface Smartcard** sono schede ibride che integrano entrambe le tecnologie contact e contactless. Queste schede tuttavia non sono molto diffuse soprattutto perché la loro fabbricazione richiede dei processi complessi.

A partire dai prossimi paragrafi quando si utilizzerà il termine generico smartcard, si farà riferimento alle microprocessor contact smart card.

Figura 4 - Lettore di Contact Smartcard.

Figura 5 - Lettore di Contactless Smartcard.

1.3 Architettura interna di una smart card

Una smart card dal punto di vista architetturale è sostanzialmente un microcontrollore realizzato su un singolo chip.

Le CPU che dominano il mercato delle smart card sono principalmente a 8 bit e sono versioni specialmente adattate dei famosissimi microprocessori Motorola 6805 ed Intel 8051 (Rankl 2007). Gli adattamenti riguardano principalmente caratteristiche elettriche, in modo da ottenere alte performance con bassissimi consumi di potenza, e fisiche per prevenire il più possibile attacchi informatici. Per ragioni di forza e robustezza il chip non può superare in dimensioni i 25 mm^2.

La Figura 6 mostra i principali componenti presenti all'interno di una smart card.

La RAM è utilizzata dal microprocessore per conservare i risultati parziali della computazione e tipicamente contiene da 100 a 8K byte. La ROM contiene il sistema operativo e i dati relativi al microchip (codice seriale, produttore, ecc.) e

può raggiungere dimensioni massime di 200 Kbyte. La EEPROM consente la lettura e la scrittura di dati che persistono anche in assenza di alimentazione e le sue dimensioni variano dagli 8 Kbyte ai 64 Kbyte. Ultimamente sono state realizzate delle smart card basate su memoria flash in sostituzione della EEPROM in quanto offrono prestazioni e durata superiore. Ogni smart card è dotata inoltre di una porta di I/O seriale per comunicare con l'esterno.

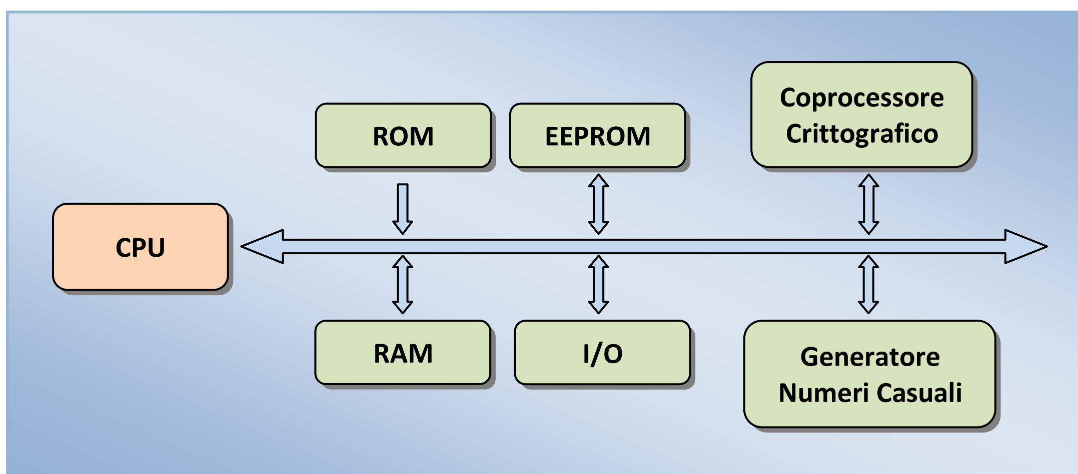


Figura 6 - Architettura interna di una smart card

Le smart card più sofisticate permettono di eseguire al proprio interno diverse funzionalità crittografiche sfruttando appositi moduli hardware come un generatore di numeri casuali e un coprocessore crittografico. Gli algoritmi principali implementati riguardano la crittografia a chiave simmetrica (DES, TDES, AES) e la crittografia a chiave asimmetrica (RSA).

Recentemente si stanno diffondendo smartcard multi-applicazione che permettono di ospitare al loro interno i dati relativi ad applicazioni e servizi di tipo diverso. In particolare esistono due implementazioni importanti nel panorama internazionale: **JavaCard** e **SmartCard.NET**.

JavaCard è un sottoinsieme del famoso linguaggio Java per PC in grado di essere utilizzato per costruire applicazioni eseguibili all'interno di una smartcard. Un'applicazione di questo tipo prende il nome di Applet e viene eseguita

all’interno della Java Card Virtual Machine (JCVM) che rappresenta un sottoinsieme della macchina virtuale Java.

SmartCard.NET è la risposta di Microsoft a JavaCard per portare le funzionalità della piattaforma .NET all’interno delle smartcard. Le schede compatibili con SmartCard.NET offrono una macchina virtuale in grado di eseguire codice scritto nei linguaggi .NET (C# e VB.NET) in grado di utilizzare molte delle funzionalità presenti nella libreria standard del framework.

I principali vantaggi offerti dalle tecnologie JavaCard e SmartCard.NET sono un notevole incremento di flessibilità nello sviluppo e sicuramente un aumento dell’interoperabilità tra le applicazioni. Per maggiori informazioni consultare (Java Card Technology s.d.) e (.NET Smart Card Framework s.d.).

1.4 Architettura di un sistema basato su smartcard

Dal punto di vista hardware l’architettura più semplice di un sistema basato su smart card è costituita da un lettore di smart card collegato ad un personal computer, generalmente tramite porta seriale o porta USB (Figura 7).

Il lettore fornisce l’alimentazione e il segnale di clock alla smart card e gestisce il canale di I/O da e verso la scheda.

Figura 7 - Componenti hardware di un semplice sistema basato su smart card

Dal punto di vista logico una soluzione basata su smart card consiste in un'applicazione software che gira sul computer e che interagisce con la smart card invocando i vari comandi che essa rende disponibili.

Per fare questo l'applicazione può sfruttare delle opportune API offerte dal sistema operativo del PC oppure sfruttare un middleware che fornisce un'interfaccia di programmazione più semplice e flessibile. Alla base di tutto è presente il driver del lettore che è l'unico componente in grado di interagire direttamente con la scheda.

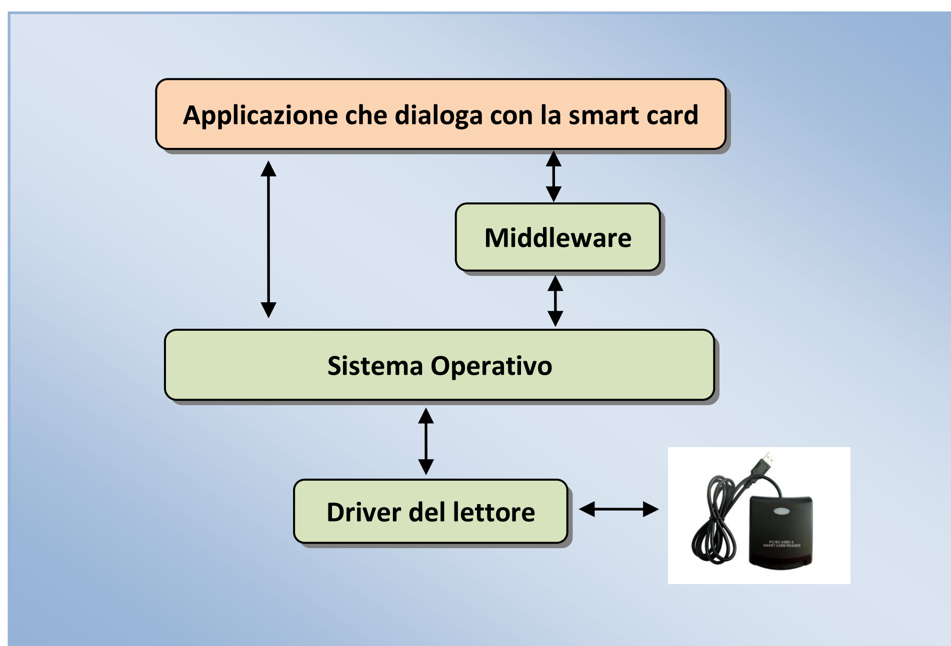


Figura 8 - Architettura logica di un sistema basato su smart card

Le applicazioni possono essere classificate in **chiuse** o **aperte**.

Nel caso di applicazioni chiuse l'intero sistema è nella mani di una singola entità. Al contrario un sistema aperto coinvolge altre entità che sono integrate fra loro ma che non appartengono all'operatore del sistema stesso. Probabilmente il miglior esempio di un tale sistema sono le smart card usate come carte di credito: il fornitore è una banca, l'operatore di sistema è una compagnia internazionale di carte di credito e chi accetta le carte, sono i commercianti.

1.5 Principali soluzioni basate su smart card

Le migliori applicazioni per cui conviene utilizzare smart card sono quelle che hanno bisogno di un numero di dispositivi di memoria poco costosi che possono memorizzare in modo sicuro dati individuali e personali e che devono eseguire attività riguardanti la sicurezza come l'autenticazione, la cifratura e/o la firma.

Le principali applicazioni coinvolgono i seguenti settori:

- Telecomunicazioni
- Finanza
- Governo e sanità
- Trasporti
- TV satellitare
- Controllo fisico degli accessi

Il mercato delle smart card è in forte e continua crescita. Basti pensare che nel solo anno 2008 sono state vendute più di cinque miliardi di schede (Tabella 1). Per maggiori informazioni sulle statistiche di vendita consultare (Eurosmart - The Voice of the Smart Security Industry s.d.).

Nell'ambito delle telecomunicazioni le smart card sono utilizzate per realizzare le SIM card presenti all'interno dei telefoni cellulari. Una SIM card memorizza un identificatore del cliente e permette in maniera sicura l'identificazione dell'utente stesso all'interno della rete cellulare grazie all'utilizzo di opportuni algoritmi crittografici. Inoltre la SIM card permette la memorizzazione d'informazioni personali dell'utente come la rubrica telefonica e gli SMS.

Le banche utilizzano le smart card per realizzare carte di credito ad alta sicurezza in sostituzione delle carte magnetiche che non offrono un'adeguata protezione dalle frodi informatiche.

Settore	Memory Card (Mu)	Microprocessor Card (Mu)
Telecomunicazioni	380	3200
Finanza	30	650
Governo e sanità	250	140
Trasporti	160	30
Pay TV	-	100
Altro	80	65
Totale	900	4185
TOTALE	5085	

Tabella 1 - Numero di smart card (in milioni di unità) vendute nel 2008 a livello mondiale. (fonte Eurosmart)

Nell'ambito governativo le smart card sono utilizzate principalmente come strumento d'identificazione e di memorizzazione d'informazioni personali. E' importante citare la **Carta Nazionale dei servizi** (**CNS**) e la **Carta d'identità elettronica** (**CIE**). La prima è un documento informatico, rilasciato da una Pubblica amministrazione, con la finalità di identificare in rete il titolare della carta e garantirgli l'accesso a diversi servizi. La seconda invece è il documento d'identificazione destinato a sostituire la carta d'identità cartacea sul territorio italiano. Per maggiori informazioni fare riferimento a (CNS, Carta Nazionale dei Servizi s.d.).

Nei trasporti pubblici le smart card sono utilizzate in sostituzione dei biglietti e diventano a tutti gli effetti titoli di viaggio. Tipicamente sono utilizzate come biglietti prepagati o in sostituzione degli abbonamenti.

Nel settore delle TV satellitari le smart card sono utilizzate principalmente come strumento per controllare l'accesso ai contenuti digitali nel modo più sicuro possibile.

Per quanto riguarda il controllo fisico degli accessi, le smart card permettono di realizzare procedure di autenticazione molto sicure. La sicurezza è basata sostanzialmente sul possesso della smart card e sulla conoscenza di un PIN. Per

innalzare ulteriormente il livello di sicurezza il PIN può essere sostituito o affiancato da un'informazione biometrica come l'impronta digitale, la conformazione dell'iride, ecc. L'utilizzo della tecnologia biometrica offre il grande vantaggio che l'accesso alle credenziali utente sulla smart card può avvenire solo in presenza del titolare, il quale addirittura non è tenuto a ricordare alcun PIN. Alcune smart card dell'ultima generazione permettono di far eseguire direttamente al microprocessore l'algoritmo di corrispondenza specifico per l'impronta digitale. Con tal estensione, l'impronta originale del titolare non è mai estratta dalla memoria della smart card e pertanto assicura un elevatissimo livello di sicurezza.

1.6 Gli standard esistenti

Fino a pochi anni fa, le smart card e i lettori disponibili sul mercato erano molto differenti tra loro e rendevano estremamente complesso lo sviluppo di applicazioni basate su smart card. A causa delle notevoli differenze tra le molteplici implementazioni native, solo un numero ristretto di programmatori altamente specializzati era in grado di scrivere applicazioni che funzionavano. Queste applicazioni però funzionavano solamente con un insieme estremamente ristretto di smart card e/o lettori.

La crescente domanda di smart card e lo sviluppo tecnologico degli ultimi anni ha portato alla definizione di un insieme di specifiche standard che consentisse l'interoperabilità tra smart card e lettori di diversa fabbricazione.

Nei seguenti paragrafi saranno discussi due dei principali standard attualmente esistenti.

1.6.1 La famiglia di standard ISO-7816

La famiglia di standard ISO-7816 definisce le caratteristiche fisiche, elettriche ed operative delle smart card a microprocessore con contatti elettrici.

Lo standard è strutturato in dieci parti:

- Parte 1: Caratteristiche Fisiche
- Parte 2: Dimensioni e posizione dei contatti elettrici
- Parte 3: Segnali elettrici e protocolli di trasmissione
- Parte 4: Comandi per l'intercambiabilità
- Parte 5: Procedure di registrazione e sistemi di numerazione per identificativi d'applicazione
- Parte 6: Definizione dei dati
- Parte 7: Comandi per query strutturate
- Parte 8: Comandi riguardanti la sicurezza
- Parte 9: Comandi addizionali e attributi di sicurezza
- Parte 10: Segnali elettrici e "Answer to Reset"

Dal punto di vista dello sviluppatore di soluzioni basate su smart card le parti più importanti dello standard sono la parte 4, la parte 7 e la parte 8.

La parte 4 descrive i comandi, i messaggi e le risposte trasmessi tra la smart card e il terminale d'interfacciamento, la struttura logica dei file e dei dati memorizzati nella memoria della smart card e i meccanismi di protezione dell'accesso agli stessi. Definisce inoltre i comandi crittografici per cifrare la comunicazione tra la smart card e il terminale (Secure Messaging).

La parte 7 definisce i concetti di un database su smart card con relativo SCQL (Structured Card Query Language) e i relativi comandi.

La parte 8 specifica i protocolli di sicurezza utilizzabili, i comandi atti ad eseguire algoritmi crittografici sia in chiave privata sia in chiave pubblica e a maneggiare i relativi certificati, e altri comandi per la trasmissione sicura dei dati.

1.6.2 Lo standard PC/SC

Nel maggio del 1996 è stato creato un consorzio di compagnie con lo scopo di definire un’architettura standard per interfacciare la smart card con un PC.

Il gruppo ha identificato tre parti da standardizzare:

- L’interfaccia tra il lettore e il PC
- Una API di alto livello per accedere alle funzionalità della smart card
- Un meccanismo per permettere a molte applicazioni di condividere risorse, come schede o lettori

I membri del consorzio hanno anche definito un’architettura e delle specifiche che permettessero ai produttori di carte e lettori di sviluppare in modo indipendente i loro prodotti con la certezza che questi potranno funzionare insieme. Nel dicembre del 1997 nasce lo standard PC/SC (PC/Smart Card) che nel tempo è stato aggiornato ed ha raggiunto la versione 2.0.

La Figura 9 mostra l’architettura definita dallo standard PC/SC che è suddivisa in nove parti che saranno ora brevemente descritte.

La parte 1 offre un’anteprima dell’architettura del sistema e dei componenti.

La parte 2 entra nel merito dei requisiti per assicurare l’interoperabilità tra la smart card (ICC, Integrated Chip Card) ed il lettore (IFD, Interface Device) come le caratteristiche fisiche e i protocolli di comunicazione.

La parte 3 descrive l’interfaccia che il driver di un lettore (chiamato IFD Handler) deve implementare affinché i livelli più alti dello stack PC/SC possano comunicare in modo indipendente con esso, senza conoscere il connettore sottostante (USB o seriale) e i protocolli utilizzati.

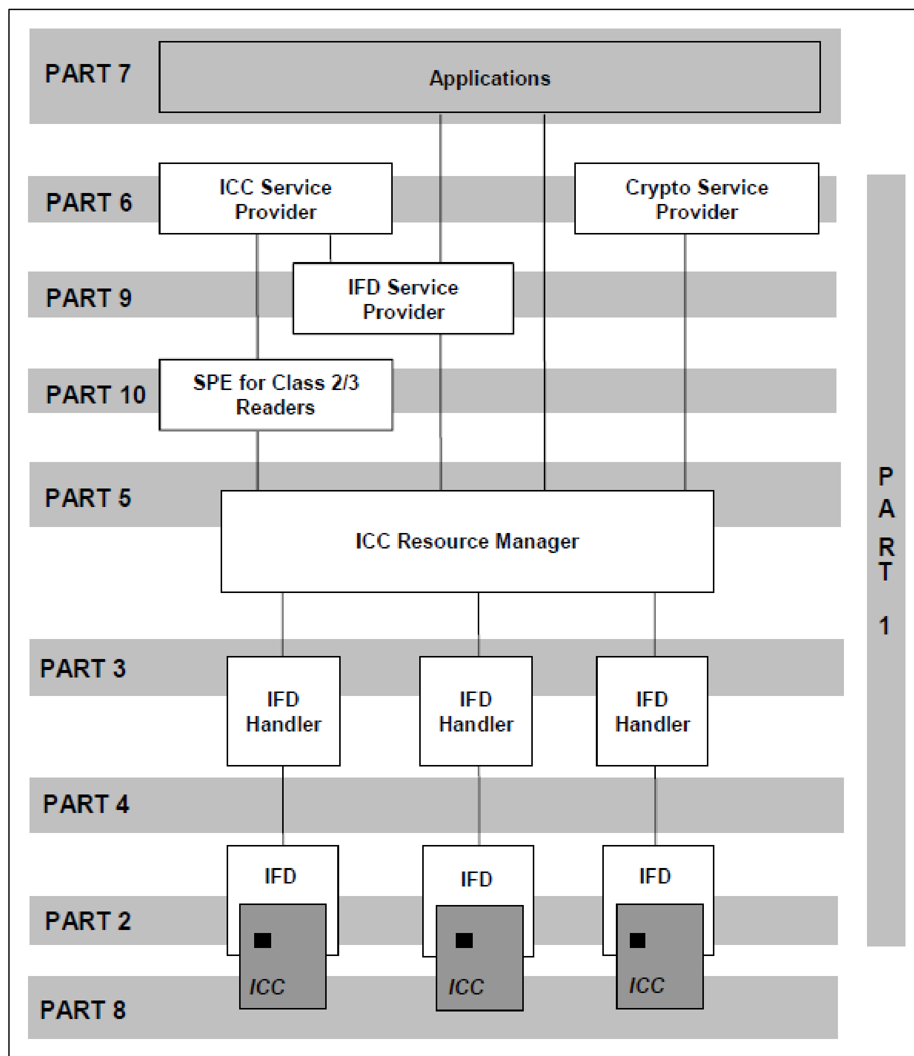


Figura 9 - Architettura definita dallo standard PC/SC 2.0

La parte 4 presenta i vari tipi d'interfacce verso il lettore (PS2, USB, seriale, PCMCIA, ecc) e fornisce un certo numero di raccomandazioni che i lettori dovrebbero rispettare.

La parte 5 descrive le interfacce e le funzionalità supportate dal Resource Manager che è l'elemento centrale dell'architettura PC/SC.

La parte 6 ha lo scopo di nascondere la complessità dell'architettura sottostante proponendo delle API di alto livello per l'esecuzione di specifiche operazioni.

La parte 7 presenta i metodi per lo sviluppo di applicazioni e descrive come utilizzare gli altri componenti.

La parte 8 descrive le funzionalità raccomandate per il supporto alla crittografia.

La parte 9 descrive come gestire i lettori con caratteristiche addizionali come sensori biometrici, schermi, ecc.

Le applicazioni basate su smart card sono costruite sopra uno o più Service Provider e sul Resource Manager. I provider incapsulano le funzionalità per una specifica smart card e le rendono accessibili attraverso interfacce di alto livello. Il Resource Manager gestisce le risorse relative alle schede ed ai lettori all'interno del sistema. Esso permette di comunicare con i lettori e con le smart card tramite una semplice interfaccia sulla quale sono implementati i servizi dei livelli più alti.

Microsoft come membro del consorzio è stata la prima azienda a fornire un'implementazione delle specifiche PC/SC. Queste funzionalità sono integrate in tutti i sistemi operativi Windows e sono implementate all'interno di una libreria di sistema chiamata winscard.dll.

Per fornire un supporto alle smart card anche all'interno degli ambienti Unix è stato creato il movimento M.U.S.C.L.E (Movement for the Use of Smart Card in A Linux Environment) con il progetto pcsc-lite, avviato da David Corcoran, che implementa le SCard API di Microsoft. Oggi, pcsc-lite funziona su un grande numero di piattaforme come Linux, Solaris, Mac OS X, FreeBSD, ecc. Apple ha deciso di utilizzare pcsc-lite come implementazione di riferimento per il suo sistema Mac OS X. Tecnicamente, il progetto pcsc-lite è costituito da un demone (pcscd) e da una libreria condivisa che implementa le varie funzionalità.

Dal 2003, il progetto pcsc-lite gioca un importante ruolo nel mondo delle smart card, in particolare grazie al lavoro di Ludovic Rousseau.

Per maggiori informazioni sullo standard PC/SC e le sue implementazioni fare riferimento a (PC/SC Workgroup s.d.), (Smart Card Resource Manager API s.d.) e (pcsc-lite SCard API s.d.).

1.7 Problematiche nello sviluppo di applicazioni

All'interno di questo paragrafo sono presentati i principali problemi che affliggono lo sviluppo di soluzioni basate su smart card.

1.7.1 Mancanza d'interoperabilità tra smartcard

Il mondo delle smartcard ha una complessità intrinseca dovuta all'esistenza di una moltitudine di tipi di carte e lettori. Le prime tre parti dello standard ISO-7816 sono seguite dalla totalità dei produttori di smart card poiché definiscono le caratteristiche fisiche ed elettriche della smart card ed in particolare del microchip. Le altre, che descrivono principalmente le strutture dati, i comandi e i protocolli di comunicazione, sono viste solitamente come indicazioni generali e spesso molti costruttori adottano soluzioni parziali e/o alternative. Lo standard ISO7816-4, ad esempio, definisce una serie di comandi per accedere a file e chiavi ma non definisce nulla su come queste risorse possano essere create sulla scheda. Questi aspetti costringono i produttori ad implementare comandi proprietari (e quindi non interoperabili) per colmare queste mancanze.

Il vero problema, tuttavia, è che i produttori stessi vedono l'interoperabilità come un nemico perché hanno tutto l'interesse di inserire comandi proprietari (o una variazione di quelli standard) allo scopo di aggiungere funzionalità che rendono i propri dispositivi unici rispetto alla concorrenza.

Tutto questo costringe lo sviluppatore a consultare sempre il manuale specifico del sistema operativo usato dalla smart card, per ottenere informazioni sulla codifica dei comandi, rendendo di fatto la propria applicazione non interoperabile con altre schede.

Questa situazione scoraggia un'ampia diffusione della tecnologia delle smartcard perché gli sviluppatori non hanno un modo diretto di scrivere applicazioni

interoperabili e indipendenti dalle schede e sono costretti ad imparare diverse estensioni proprietarie per ogni tipo di carta che intendono supportare.

Lo standard JavaCard ha fatto un nuovo passo avanti verso l'interoperabilità tra dispositivi di produttori differenti grazie all'indipendenza della piattaforma favorita dall'utilizzo del linguaggio Java. La possibilità di caricare ed eseguire codice direttamente sulla smartcard, inoltre, se da un lato offre una grande libertà nello sviluppo di soluzioni, dall'altro diventa una sorgente di errori ed incompatibilità.

Il mondo delle smartcard si affida ancora ad un approccio "proprietario" in cui ogni produttore deliberatamente devia dagli standard per rendere più appetibili i propri prodotti. Se il produttore in futuro non produce più una carta che è stata utilizzata in un'applicazione, lo sviluppatore deve sperare che la versione successiva sia interoperabile con la precedente perché in alternativa sarà costretto a riscrivere tutto il software.

Per maggiori informazioni sul tema dell'interoperabilità consultare (Tesi di dottorato di Tommaso Cucinotta: Issues in authentication by means of smart card devices. 2004) e (Computer Science Software Engineering (CSSE) Journal, Vol. 20, No. 6 - An open middleware for smart-cards 2005).

1.7.2 Interfacce poco intuitive

Per scrivere applicazioni basate su smartcard, si ricorre sostanzialmente all'utilizzo del linguaggio C/C++ in quanto le funzionalità del Resource Manager sono esposte attraverso una API in C. Questa interfaccia è poco intuitiva e costringe lo sviluppatore a lavorare ad un basso livello di astrazione.

I produttori di schede spesso offrono opportune librerie per migliorare e semplificare lo sviluppo ma sono spesso a pagamento e non interoperabili tra schede di produttori differenti.

1.7.3 Mancanza di strumenti per il testing e il debugging

Il software di applicazioni embedded, che include il software di applicazioni basate su smart card, è soggetto a stringenti requisiti nell'ambito della qualità. Due dei più importanti attributi di qualità sono la robustezza e l'assenza di difetti ed errori. Per soddisfare questi requisiti è necessario realizzare attente procedure di testing e debugging delle proprie applicazioni.

Esistono diversi tipi di test che possono essere utilizzati a seconda della specifica fase in cui ci si trova durante il ciclo di vita della propria applicazione. I test unitari sono utilizzati per testare moduli software individuali. I test d'integrazione invece permettono di testare l'interazione di un gruppo di moduli. Infine il test di sistema ha lo scopo di testare l'intero sistema comprendente tutti i moduli che sono stati realizzati. Un test-case definisce uno specifico test in termini di scopo, prerequisiti, inputs, procedura di test e risultati attesi. Le suite di test sono sempre composte da un numero considerevole di test case.

Gli scenari di test comunemente utilizzati sono arrangiati gerarchicamente in ordine di precedenza. Il primo livello consiste nel test dei comandi con lo scopo di testare in modo individuale i vari comandi di una specifica smart card utilizzando un test per ogni comando. Il livello successivo consiste nel testare tipici casi d'uso che consistono di molti comandi differenti e sequenze di comandi. Quest'approccio bottom-up al testing, che è comunemente utilizzato nell'industria, può essere usato per testare sistematicamente le applicazioni basate su smart card iniziando con comandi individuali e finendo con l'intero sistema. I test per le smart card devono essere progettati in modo che possano essere eseguiti automaticamente senza interventi manuali.

E' importante inoltre sottolineare che nell'ambito delle smartcard la sicurezza della soluzione è fondamentale, ragion per cui è necessario testare completamente non solo i test-case comuni e di successo, ma anche quelli negativi, ad esempio quelli in cui ci s'impersonifica in un ipotetico attaccante che cerca di clonare una chiave privata presente nella smartcard. In alcuni contesti,

dove esistono stringenti vincoli di qualità, si utilizza spesso un approccio top-down codificando i casi di test direttamente in fase di specifica di sistema, per poi utilizzarli alla fine dell'implementazione per validarla.

Un completo insieme di test per un sistema operativo di una smart card tipicamente comprende circa 50000 test case con un totale di 300000 comandi invocati. I test quindi richiedono la scrittura di molto codice che accede alle smart card con tutti i problemi presentati in precedenza. Sarebbe molto utile possedere degli strumenti che semplificano le attività d'interazione con le smartcard in modo da scrivere test ed effettuare il debug delle proprie applicazioni in tempi più rapidi.

La maggior parte dei produttori fornisce degli strumenti grafici che permettono di accedere a tutte le funzionalità delle loro schede in maniera estremamente semplice. Tuttavia questi strumenti, essendo dotati d'interfaccia grafica, hanno la limitazione di non essere utilizzabili per realizzare procedure di test automatiche. Tutto ciò di fatto rende questi strumenti adatti solamente nella fase iniziale di apprendimento delle funzionalità di una scheda ma sono praticamente inadatti per supportare il testing automatico. Alcuni produttori quindi affiancano allo strumento grafico un interprete a riga di comando in modo che quest'ultimo possa essere integrato all'interno di script di test automatici. Questi strumenti tuttavia sono prodotti commerciali spesso costosi e soprattutto proprietari e non interoperabili. Per questo motivo, tra gli sviluppatori di soluzioni basate su smartcard, è forte l'esigenza di avere a disposizione strumenti open source che possano aumentare notevolmente la loro produttività.

1.8 Strumenti open source attualmente disponibili

In questo paragrafo s'introducono alcuni strumenti disponibili liberamente che sono stati creati per venire incontro alle problematiche di testing e debugging di applicazioni basate su smartcard.

1.8.1 Smartsh

Smartsh è una shell realizzata da Tommaso Cucinotta come strumento di utilità per gestire il testing e il debugging di un'applicazione sviluppata nell'ambito della sua tesi di laurea in Ingegneria Informatica. La shell è in grado di inviare comandi generici secondo il protocollo T=0, comandi definiti nello standard ISO7816-4 e in particolare è in grado di invocare comandi crittografici presenti nelle schede Cyberflex Access 16K prodotte da Schlumberger.

Per maggiori informazioni fare riferimento a (Tesi di Laurea di Tommaso Cucinotta: Progettazione e realizzazione di un sistema per firma digitale e autenticazione basato su smartcard. 1999). Per scaricare l'ultima versione della shell fare riferimento a (Smart Sign Project 2004).

1.8.2 Muscle Tool e XCardii

Muscle Tool è uno strumento a linea di comando che fornisce un'interfaccia utente per eseguire diverse operazioni sulle smartcard. E' costruito sopra un'interfaccia portabile chiamata Muscle API che implementa un insieme di funzioni di base tramite l'utilizzo della libreria standard PC/SC. Grazie a Muscle Tool è possibile eseguire comandi di alto livello per la manipolazione di oggetti, PIN e chiavi a bordo delle schede supportate.

XCardii è un'applicazione grafica in grado di gestire in modo visuale una MuscleCard (una scheda JavaCard contenente l'applet di MuscleCard o una scheda Cryptoflex).

Per maggiori informazioni consultare (Muscle Card - Movement for the use of the smart cards in a linux environment s.d.) e (Graphical program to manage a MuscleCard smartcard s.d.).

1.8.3 OpenSC Tools

OpenSC include un insieme di strumenti a linea di comando per esplorare smarcard, testare, automatizzare e debuggare applicazioni.

Di seguito sono riportati alcuni tra gli strumenti disponibili:

- **opensc-tool**: è uno strumento che permette di visualizzare i lettori presenti nel sistema e identificare le schede inserite.
- **opensc-explorer**: permette di visualizzare il file system di una scheda e compiere operazioni di lettura e scrittura di file.
- **cryptoflex-tool**: permette di eseguire comandi per le schede Cryptoflex
- **cardos-info**: permette di ottenere informazioni su schede Siemens CardOS/M4
- **pkcs15-tool**: permette di manipolare pin, certificati e chiavi secondo quanto definito nello standard PKCS #15.

Per maggiori informazioni consultare (OpenSC Tools s.d.).

1.8.4 GPShell

GlobalPlatform è uno standard per la gestione del contenuto di smartcard che si occupa in particolare dell’installazione e della rimozione di applicazioni. La GPShell è un interprete a riga di comando che offre un modo semplice ed automatizzabile di eseguire comandi in conformità a quanto definito nello standard.

Per ottenere maggiori informazioni sullo standard consultare (Global Platform s.d.) mentre per scaricare il codice sorgente della shell fare riferimento a (GPShell s.d.).

1.8.5 Limiti di questi strumenti

Tutti gli strumenti mostrati in precedenza se da una parte hanno liberato il testing e il debugging di applicazioni basate su smartcard dall'utilizzo di strumenti proprietari dall'altra parte presentano una forte limitazione di utilizzo. Ciascuno di essi infatti può essere utilizzato solamente su particolari categorie di dispositivi ed offre un numero limitato di funzionalità.

Il mondo open source, quindi, ha bisogno di uno strumento che possa diventare la scelta di riferimento per il testing e il debugging di applicazioni basate su smartcard; uno strumento in grado di evolvere ed espandersi nel tempo e che non abbia alcuna limitazione nel numero di funzionalità e dispositivi che possono essere supportati. Da queste motivazioni nasce il progetto SCOTT che sarà descritto nel dettaglio nei capitoli successivi.

2. Introduzione a SCOTT

2.1 Che cos’è SCOTT ?

Il processo di sviluppo di software per smart card continua ad essere particolarmente complesso per la mancanza di appropriati strumenti di supporto allo sviluppo e al debug, se non quelli proprietari messi a disposizione dai singoli fornitori di soluzioni, che comunque spesso sono artificiosamente non interoperabili per ragioni puramente commerciali.

La presenza e la continua nascita di strumenti open source che semplificano l’interazione con le smartcard, alcuni dei quali sono stati presentati nel paragrafo 1.7, testimonia il fatto che è molto sentita tra gli sviluppatori la necessità di avere strumenti per supportare il testing automatico di soluzioni basate su smartcard. Il problema principale di questi strumenti liberi è quello di supportare solamente un numero ristretto di schede e funzionalità. Tutto questo porta allo sviluppo di una moltitudine di applicativi diversi per risolvere sostanzialmente lo stesso problema in scenari differenti. La soluzione ideale è quindi quella di avere un unico strumento che possa essere utilizzato in tutti gli scenari possibili e che sia in grado di inglobare le funzionalità degli strumenti attualmente disponibili e renderle fruibili attraverso un’interfaccia uniforme.

Per capire cos’è SCOTT (Smart Card Open Test Toolkit) conviene analizzare le parole contenute nell’acronimo che rappresenta:

- “**Smart Card**” indica che il progetto riguarda il mondo delle smart card.
- “**Toolkit**” si riferisce al fatto che SCOTT non è un singolo strumento ma un insieme di strumenti e librerie per lo sviluppatore.
- “**Test**” indica che lo scopo del toolkit è supportare il testing di soluzioni basate su smart card.
- “**Open**” invece vuole sottolineare l’indipendenza da specifici produttori di schede perché il software è completamente espandibile e rilasciato come Open Source. L’evoluzione di SCOTT è quindi lasciata alla comunità degli

sviluppatori con lo scopo di favorire l'interoperabilità e la diffusione di strumenti condivisi da tutti.

SCOTT nasce quindi come un toolkit di riferimento per supportare il testing di soluzioni basate su smartcard con la caratteristica di essere liberamente disponibile, altamente modulare ed espandibile. Tutto ciò significa che SCOTT praticamente non ha alcuna limitazione nel numero di funzionalità e dispositivi che potranno essere supportati nel tempo.

2.2 Requisiti di progettazione

SCOTT è stato progettato considerando i seguenti requisiti:

- Deve permettere di interagire con diversi dispositivi di smart card
- Deve essere completamente espandibile tramite plugin
- Le funzionalità principali devono essere isolate e indipendenti dall'interfaccia utente in modo da rendere possibile lo sviluppo d'interfacce differenti senza inutili duplicazioni
- Deve integrare almeno un interprete a riga di comando (Shell) con caratteristiche tali da essere utilizzabile in procedure di test automatiche nel modo più semplice possibile
- Deve offrire un'interfaccia uniforme all'esecuzione di comandi
- Deve essere liberamente utilizzabile per favorire l'interoperabilità
- Deve essere utilizzabile almeno sulle piattaforme Windows e Linux

Alla luce di queste caratteristiche SCOTT è uno strumento unico nel suo genere.

Esistono strumenti, spesso proprietari e a pagamento, che semplificano l'interazione con le smart card ma che sono molto limitati perché supportano un numero ridotto di dispositivi e non sono stati progettati per essere espandibili ed integrabili in procedure di test automatiche.

La principale caratteristica che rende unico SCOTT è la sua totale espandibilità che si spera porti in futuro allo sviluppo di un numero considerevole di plugin.

2.3 Architettura software

La Figura 10 mostra l'architettura software di SCOTT.

La parte centrale dell'architettura, chiamata "SCOTT Core Library", contiene il cuore del sistema ed è isolata all'interno di una libreria dinamica (scott.dll su Windows e scott.so su Linux).

La libreria core di SCOTT:

- Definisce un'interfaccia per conoscere e caricare i plugin installati
- Permette di invocare i comandi offerti dai plugin e ricevere i risultati
- Definisce un ricco sistema di tipi con cui realizzare le interfacce dei plugin
- Definisce una funzione per la formattazione dei dati
- Offre diversi servizi come la risoluzione di espressioni e la risoluzione dei nomi per accedere in modo univoco a dati e comandi esposti dai plugin
- Definisce un'interfaccia che ogni plugin deve implementare in modo da rendere uniforme l'accesso ad essi
- Integra il parser del descrittore di plugin e ne gestisce le dipendenze

La libreria core da sola non offre alcun comando nell'ambito delle smart card e delega completamente ai plugin il compito di implementare le varie funzionalità. SCOTT è stato realizzato con l'obiettivo primario di essere completamente espandile, quindi chiunque può realizzare con relativa semplicità un plugin arricchendo il toolkit con nuove funzionalità. Un'altra caratteristica che aumenta notevolmente la semplicità di estendere SCOTT è la possibilità di creare un plugin sfruttando le funzionalità offerte da altri realizzando quindi una rete di dipendenze tra plugin.

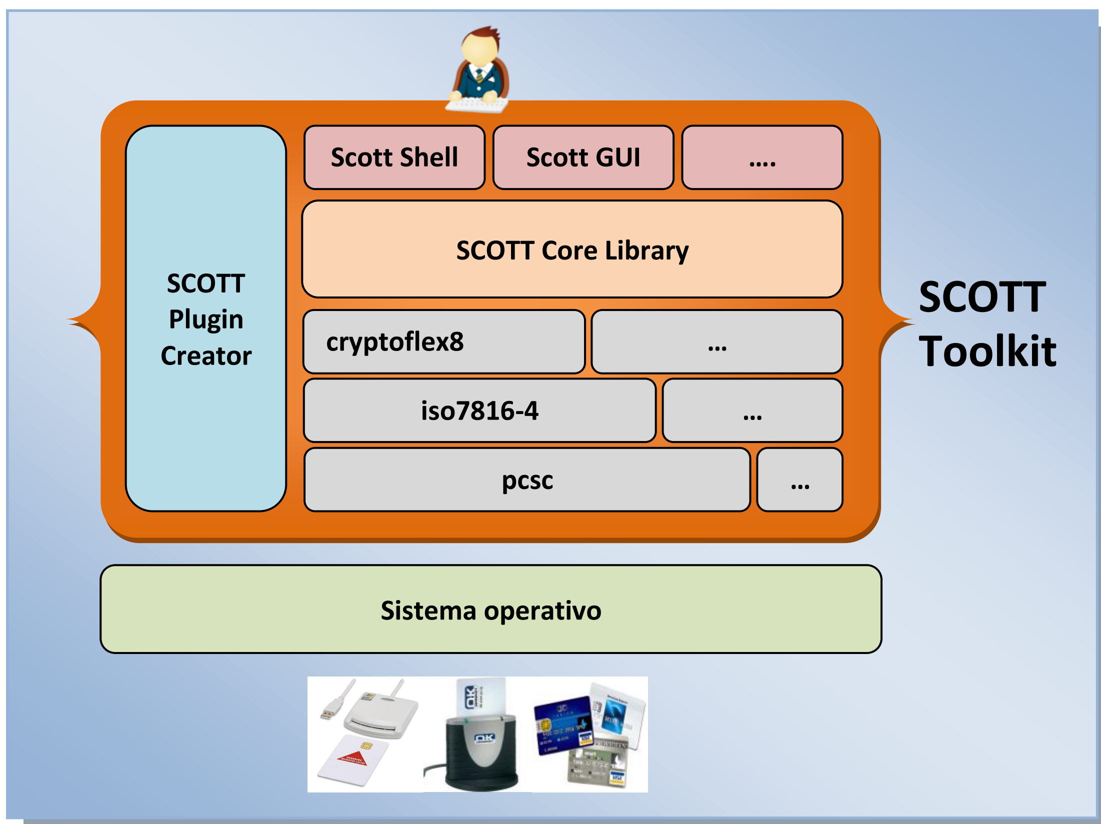


Figura 10 - Architettura software di SCOTT

Il plugin "**pcsc"** implementa i comandi principali definiti dallo standard PC/SC che permettono di instaurare una comunicazione di basso livello con qualsiasi smart card collegata al sistema.

Il plugin "**iso7816-4**" sfrutta le funzionalità offerte dal plugin pcsc per implementare alcuni dei comandi definiti dalla parte 4 dello standard ISO-7816.

Il plugin "**cryptoflex8**" dipende dai precedenti due plugin e aggiunge altri comandi specifici della scheda Cryptoflex 8K.

Si spera che in futuro siano disponibili un numero considerevole di plugin che implementano i comandi definiti in standard industriali e comandi offerti dalle varie schede disponibili sul mercato. Dal punto di vista tecnico un plugin è semplicemente una libreria dinamica spesso accoppiata con un file xml che permette di definire, in modo semplice, l'interfaccia di accesso. Questo file xml

prende il nome di *descrittore di plugin*. Per maggiori informazioni sui plugin e sulla struttura del descrittore di plugin fare riferimento al capitolo 6.

Affinché lo sviluppatore possa accedere alle funzionalità di SCOTT e quindi a tutti i comandi offerti dai plugin disponibili, è necessario realizzare delle opportune interfacce utente o viste. Siccome le funzionalità core di SCOTT sono isolate all'interno di una libreria dinamica, è possibile teoricamente realizzare un numero qualsiasi di viste indipendenti. Nella pratica si rendono necessari solamente un interprete a riga di comando (Shell) ed una interfaccia grafica.

La shell, tramite una interfaccia testuale, permette di invocare i comandi definiti nei vari plugin installati attraverso la creazione di opportune variabili che possono anche essere esportate su file.

Nell'ambito di questa tesi, per non appesantire ulteriormente il lavoro, non è stata creata nessuna vista grafica sulle funzionalità offerte da SCOTT. Il compito di realizzarne una, si lascia a successivi studenti e alla comunità.

Lo strumento "SCOTT Plugin Creator", trasversale all'infrastruttura di SCOTT, offre un validissimo aiuto per la fase di creazione di un nuovo plugin, permettendo di generare gran parte del codice di un plugin a partire dal suo descrittore, lasciando quindi allo sviluppatore solamente il compito di implementare il corpo dei comandi del nuovo plugin.

2.4 Elenco dettagliato delle funzionalità

In questo paragrafo sono elencate in maniera estremamente dettagliata tutte le funzionalità offerte da SCOTT e dai plugin implementati. Consultare i capitoli 3 e 4 per ottenere maggiori informazioni ed esempi di utilizzo.

Il seguente elenco mostra tutte le funzionalità offerte dal toolkit SCOTT raggruppate in base a quale modulo software le rende accessibili:

1. SCOTT Shell
   1.1. Visualizzazione dell'elenco dei comandi di shell disponibili.
   1.2. Visualizzazione della sintassi e delle informazioni su uno specifico comando di shell.
   1.3. Visualizzazione dell'elenco dei plugin installati.
   1.4. Caricamento di un plugin.
   1.5. Visualizzazione dell'elenco dei comandi di plugin disponibili.
   1.6. Visualizzazione dei tipi di dato complessi disponibili.
   1.7. Visualizzazione della sintassi e delle informazioni relative ad uno specifico comando di un plugin che è stato caricato.
   1.8. Visualizzazione delle informazioni relative ad uno specifico tipo complesso definito da un plugin che è stato caricato.
   1.9. Visualizzazione dell'elenco delle variabili di shell definite.
   1.10. Visualizzazione del tipo e del valore di una variabile di shell.
   1.11. Creazione e inizializzazione di una variabile di shell.
   1.12. Meccanismo delle variabili automatiche.
   1.13. Meccanismo delle variabili implicite.
   1.14. Meccanismo della visualizzazione automatica dei risultati.
   1.15. Stampa su video di variabili con formattazione personalizzata.
   1.16. Esportazione su file di variabili con formattazione personalizzata.
   1.17. Visualizzazione delle impostazioni della shell.
   1.18. Esecuzione di un comando di plugin.
   1.19. Esecuzione automatica di comandi all'avvio tramite file di configurazione.
2. Plugin "pcsc"
   2.1. Comando EstablishContext()
   2.2. Comando ListReaders()
   2.3. Comando Connect()
   2.4. Comando Status()

2.5. Comando Transmit()

2.6. Comando Disconnect()

2.7. Comando ReleaseContext()

3. Plugin “iso7816-4”

3.1. Comando Transmit0()

3.2. Comando Select()

3.3. Comando GetResponse()

3.4. Comando Verify()

3.5. Comando ReadBinary()

3.6. Comando WriteBinary()

4. Plugin “cryptoflex8”

4.1. Comando ChangeCHV()

4.2. Comando CreateDF()

4.3. Comando CreateEF()

4.4. Comando CreateFile()

4.5. Comando DeleteFile()

4.6. Comando DirNext()

4.7. Comando GetFilesInfo()

4.8. Comando GetResponse()

4.9. Comando Invalidate()

4.10. Comando ReadBinary()

4.11. Comando Rehabilitate()

4.12. Comando RetrieveCLA()

4.13. Comando Select()

4.14. Comando SelectDF()

4.15. Comando SelectEF()

4.16. Comando UnblockCHV()

4.17. Comando UpdateBinary()

4.18. Comando VerifyCHV()

4.19. Comando VerifyKey()

2.5 Modalità di utilizzo di SCOTT

SCOTT può essere utilizzato in due modi differenti:

- Interattivo
- Automatico

L'utilizzo interattivo di SCOTT può essere fatto sia attraverso la Shell sia attraverso l'interfaccia grafica e consiste nell'invocare manualmente i comandi offerti dai vari plugin analizzandone "a vista" i risultati. L'uso interattivo è molto utile nella fase di apprendimento di SCOTT e per conoscere le funzionalità di un nuovo plugin e quindi per prendere confidenza con una particolare smart card.

La possibilità di eseguire SCOTT in maniera automatica, requisito fondamentale di progettazione, avviene attraverso la scrittura di opportuni script basati sulla SCOTT Shell, a supporto del testing di soluzioni basate su smart card.

E' importante sottolineare che SCOTT non è stato realizzato per essere utilizzato in produzione all'interno di software basati su smart card. SCOTT, è invece un toolkit di supporto allo sviluppatore in particolare durante la fase di test delle proprie soluzioni.

3. Guida all'utilizzo di SCOTT

3.1 Il sistema di tipi

SCOTT definisce un insieme di tipi di dato che possono essere utilizzati per creare delle variabili. I tipi di dato possono essere classificati in *primitivi* (o *semplici*) e *complessi* come mostrato in Figura 11. I nomi dei tipi iniziano sempre con la lettera S maiuscola che è una abbreviazione di SCOTT.

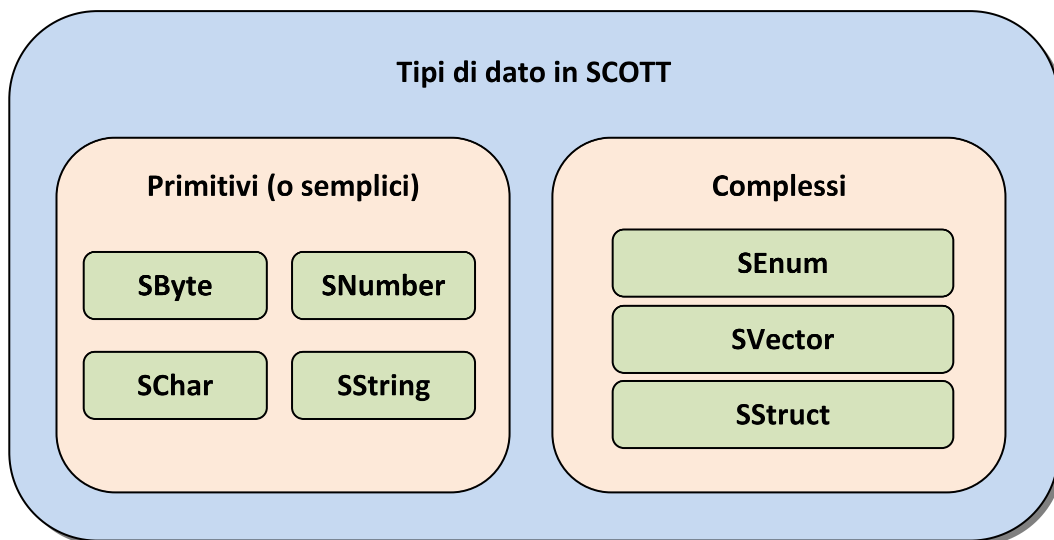


Figura 11 - Tipi di dato in SCOTT

I tipi di dato primitivi sono ben definiti e sempre disponibili all'interno di Scott.

I tipi di dato complessi invece, affinché possano essere utilizzati, devono essere accompagnati da un descrittore di tipo (Type Descriptor). Il descrittore di tipo permette di definire la struttura di un tipo di dato complesso. Sarà compito dei produttori di Plugin scrivere correttamente i descrittori per i tipi complessi a loro necessari. Per maggiori informazioni sul come definire i descrittori di tipo fare riferimento al capitolo 6.

3.1.1 Tipi di dato primitivi

I tipi di dato primitivi in SCOTT sono SByte, SNumber, SChar e SString.

Una variabile di tipo **SByte** permette la memorizzazione di un byte d'informazione. E' possibile specificare il valore utilizzando la notazione decimale o la notazione esadecimale (tramite il prefisso 0x).

Una variabile di tipo **SNumber** permette la memorizzazione di un numero senza segno su 32 bit. E' possibile specificare il valore utilizzando la notazione decimale o la notazione esadecimale (tramite il prefisso 0x).

Una variabile di tipo **SChar** permette la memorizzazione di un carattere. E' possibile specificare il valore racchiudendo il carattere tra apici singoli.

Una variabile di tipo **SString** permette la memorizzazione di una stringa di caratteri. E' possibile specificare il valore racchiudendo la sequenza di caratteri tra apici doppi.

La Tabella 2 mostra, per ciascun tipo di dato primitivo, il range di valori ammissibili e una serie di esempi di valori.

Tipo di dato primitivo	Range di valori	Esempi di valori
SByte	Da 0 a 255	0, 10, 255, 0x12, 0xFF
SNumber	Da 0 a 4294967295	0, 10000, 0xFA3, 0xF56D, 0xFFFFFFFF
SChar	-	'A', '0'
SString	-	"", "Hello SCOTT"

Tabella 2 - Tipi di dato primitivi

3.1.2 Il tipo SEnum

Una variabile di tipo SEnum permette la memorizzazione di un valore enumerato. Il descrittore di un tipo SEnum deve definire l'insieme dei valori

ammissibili sotto forma di costanti denominate cui corrisponde un valore numerico.

Un esempio di tipo SEnum è il tipo di nome Protocol definito all'interno del plugin pcsc. Una variabile di tipo Protocol può assumere uno dei valori possibili tra quelli definiti come SCARD_PROTOCOL_T0 o SCARD_PROTOCOL_T1.

3.1.3 Il tipo SVector

Una variabile di tipo SVector permette la memorizzazione di un numero variabile di elementi dello stesso tipo.

Un SVector può essere rappresentato separando tra virgole i valori dei vari elementi e racchiudendo tutto all'interno di una coppia di parentesi graffe. La Tabella 3 mostra alcuni esempi di vettori di tipi primitivi. E' importante notare che il tipo SVector di SByte è un po' speciale in quanto è possibile utilizzare come valore una lunga stringa esadecimale.

Tipo di dato	Esempi di valori
SVector di SByte	{}, { 0, 10, 3}, {0xF1, 0xF4}, {10, 0xFF, 255}, 0xFA3429DC100B
SVector di SNumber	{}, {10, 100, 1000}, { 0xFE4A, 0, 0x78F432DA, 20}
SVector di SChar	{}, { 'A', 'B', 'C' }
SVector di SString	{}, { "Uno", "Due", "Tre" }

Tabella 3 - Esempi di SVector con elementi primitivi

Il tipo degli elementi di un vettore, tuttavia, non è limitato ad essere un tipo semplice. E' quindi possibile realizzare vettori di enumeratori, vettori di vettori e vettori di strutture. La Tabella 4 mostra alcuni esempi di vettori i cui elementi sono a loro volta tipi complessi.

Questa ricca possibilità di composizione, rende il sistema di tipi di Scott estremamente flessibile e praticamente adatto a tutte le situazioni.

Tipo di dato	Esempi di valori
SVector di SEnum (Color)	{}, { red, yellow, green, red}
SVector di SVector di SNumber	{}, { {1,2,3}, { 0xFE }, {4,5}, {} }
SVector di SStruct	{}, { { 1, ‘A’, “Ciao” }, { s = “Ciao”, n = 2} }

Tabella 4 - Esempi di SVector con elementi complessi

3.1.4 Il tipo SStruct

Una variabile di tipo SStruct permette di aggregare un gruppo di variabili tra loro correlate. Oltre al nome del nuovo tipo, il descrittore di un tipo SStruct deve specificare nome e tipo di ciascuno dei suoi membri. Come nel caso degli SVector non c’è una limitazione sul tipo di dato dei membri e quest’aspetto offre una grande possibilità di personalizzazione dei dati.

Supponiamo di avere una struttura definita come in Tabella 5.

Nome del membro	Tipo del membro
B	SByte
Num	SNumber
Ch	SChar
Str	SString
Color	SEnum
VNum	SVector di SNumber

Tabella 5 - Definizione di un tipo SStruct

La Tabella 6 mostra alcuni esempi di valori ammissibili per la struttura appena definita. Com’è possibile notare, la sintassi per valorizzare una struttura è molto simile a quella per valorizzare un tipo SVector, tuttavia esistono differenti modalità per specificare i valori dei membri di un tipo struttura.

La **prima modalità (posizionale)** consiste nello specificare i valori dei membri nell’ordine in cui sono definiti nel descrittore di tipo. Questa modalità permette

di avere una stringa di valorizzazione molto breve, ma costringe l'utente a ricordare l'ordine di definizione dei membri nella struttura.

La **seconda modalità (associativa)** consiste nel far precedere al valore da assegnare a ciascun membro il rispettivo nome seguito dal simbolo di uguaglianza. Questa seconda modalità permette di specificare i membri della struttura anche senza rispettare l'ordine di definizione. Lo svantaggio di un tale approccio è un appesantimento della stringa di valorizzazione che può diventare talvolta anche notevolmente lunga e poco leggibile soprattutto in presenza di molti elementi complessi annidati.

La terza **modalità (mista posizionale e associativa)** permette di utilizzare contemporaneamente le modalità posizionale e associativa. E' possibile specificare in modo posizionale i primi membri della struttura e in modo associativo i rimanenti.

Modalità di valorizzazione	Esempi di valori
Posizionale	{ 0, 1000, 'C', "Ciao", red, {1,2,3} }
Associativa	{ B=0, Num=1000, Ch='C', Str="Ciao", Color=yellow, VNum={1,2,3} } { Num=1000, B=0, Ch='C', Str="Ciao", Color=yellow, VNum={1,2,3} }
Mista posizionale e associativa	{ 0, 1000, Str="Ciao", Ch='C', Color=yellow, VNum={1,2,3} }

Tabella 6 - Modalità di valorizzazione di un tipo SStruct

3.2 La Shell SCOTT

La Shell SCOTT (***scottsh***) rappresenta l'interfaccia principale di accesso a tutte le funzionalità offerte dal toolkit. La Shell offre un ambiente interattivo per eseguire i comandi implementati all'interno dei vari plugin disponibili.

Le principali funzionalità offerte sono:

- Visualizzazione e caricamento dei plugin
- Creazione ed utilizzo di variabili
- Esecuzione dei comandi di un plugin
- Esportazione dei dati

La Figura 12 mostra come si presenta la Shell quando viene eseguita.

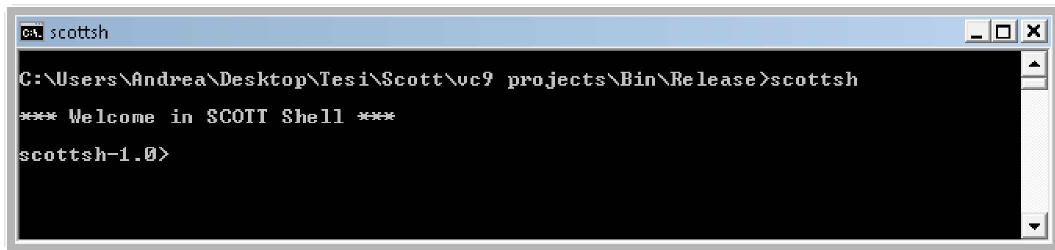


Figura 12 - La Shell SCOTT all'avvio

Tramite il comando *help* (Figura 13) è possibile ottenere l'elenco di tutti i comandi disponibili con la relativa sintassi di utilizzo.

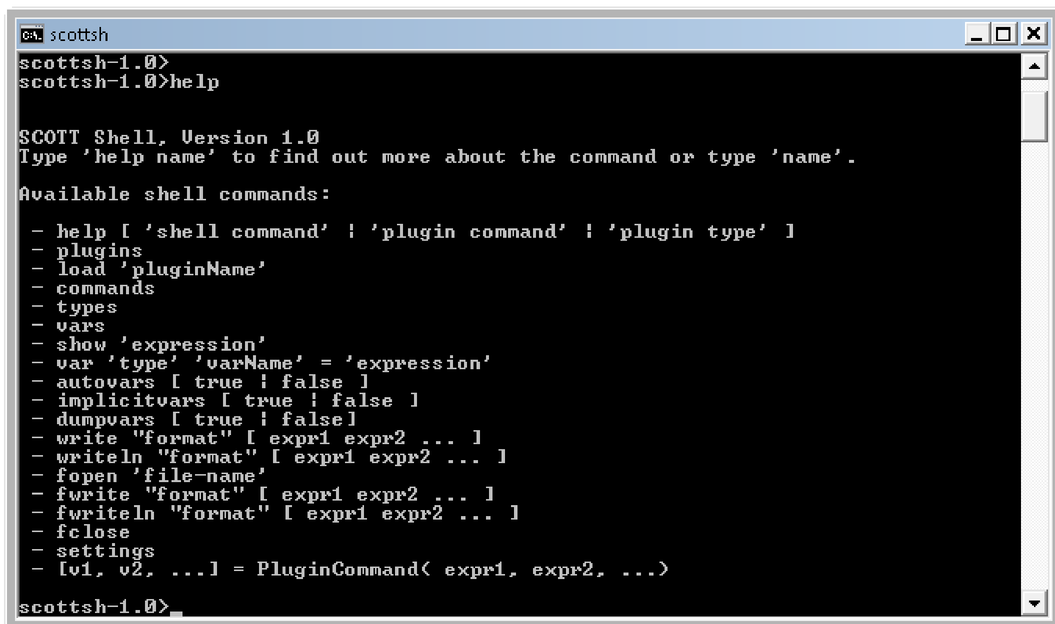


Figura 13 - Elenco dei comandi disponibili

Per ottenere maggiori informazioni su un particolare comando di shell è possibile utilizzare di nuovo il comando *help* seguito dal nome del comando d'interesse (Figura 14).

I paragrafi seguenti descrivono, uno per uno e in maniera dettagliata, tutti i comandi offerti dalla Shell.

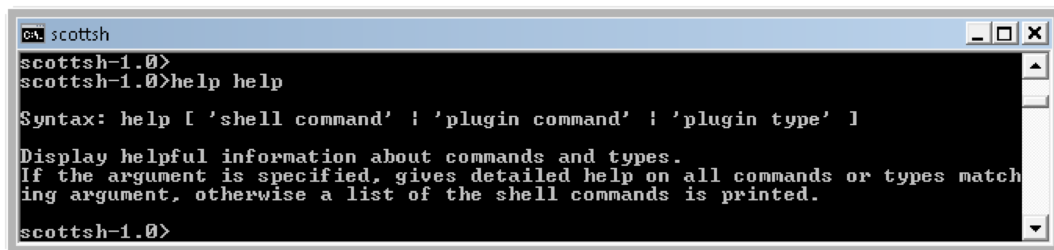


Figura 14 - Maggiori informazioni sui comandi di shell

3.2.1 Visualizzazione dei plugin installati

Il comando ***plugins*** permette di visualizzare l'elenco dei plugin installati sulla macchina. La Figura 15 mostra l'utilizzo di questo semplice comando.

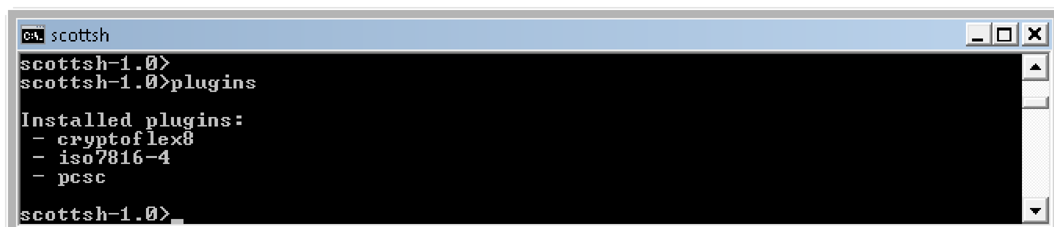


Figura 15 - Esempio di utilizzo del comando 'plugins'

3.2.2 Caricamento dei plugin

Affinché sia possibile utilizzare tipi e comandi definiti all'interno di un particolare plugin, è indispensabile, prima di tutto, procedere al caricamento del plugin stesso.

Il comando ***load*** permette di caricare un plugin il cui nome deve essere specificato come unico argomento del comando.

La Figura 16 mostra l'utilizzo del comando *load* per caricare il plugin pcsc.

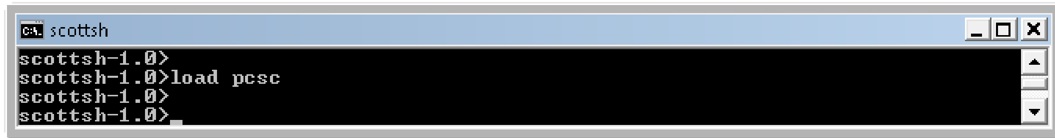


Figura 16 - Caricamento di un plugin

Nel caso in cui il plugin richiesto non fosse disponibile, sarà mostrato un opportuno messaggio di errore. E' importante sottolineare che una successiva chiamata al comando *load*, specificando un plugin già caricato, non comporta alcuna operazione. Se un plugin dopo essere stato caricato nella shell viene aggiornato, è necessario chiudere la Shell e rieseguire il comando *load* per rendere visibili i nuovi aggiornamenti.

Se è caricato un plugin che dipende da altri plugin, questi ultimi saranno caricati automaticamente. La Figura 17 mostra il caricamento del plugin iso78164 che dipende dal plugin *pcsc*.

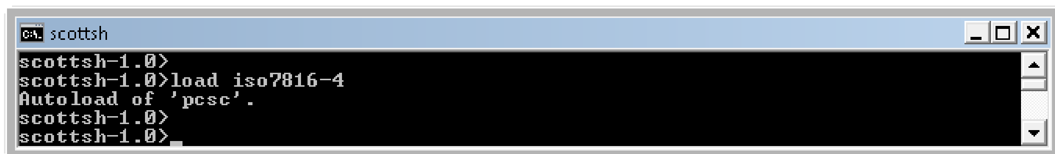


Figura 17 - Caricamento di un plugin con dipendenze

Per maggiori informazioni relative alla fase di caricamento di un plugin consultare il capitolo 6.

3.2.3 Visualizzazione dei tipi disponibili

Il comando ***types*** permette di visualizzare tutti i tipi definiti nei plugin che sono stati caricati tramite il comando *load* e che sono quindi direttamente utilizzabili.

La Figura 18 mostra un esempio di utilizzo del comando *types* dopo che è stato caricato il plugin *iso7816-4*.

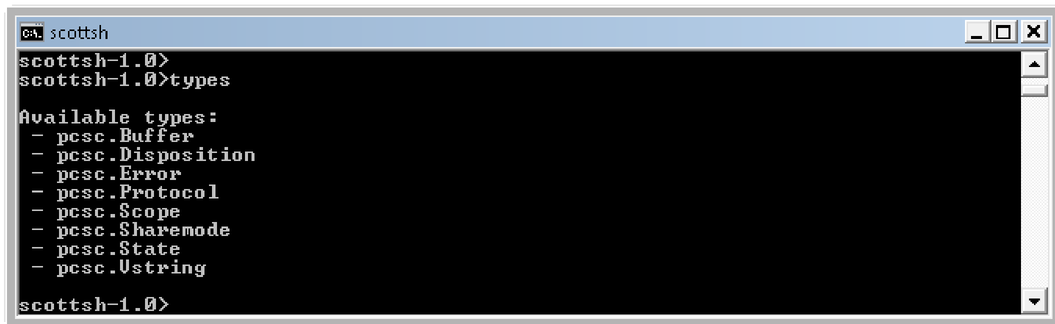


Figura 18 - Comando 'types'

Per identificare in maniera univoca un tipo di dato si deve utilizzare quello che è chiamato **fully qualified type name**. Il nome effettivo di un tipo è quindi costituito dal nome del plugin in cui è definito seguito da un punto e dal nome del tipo (Figura 19).

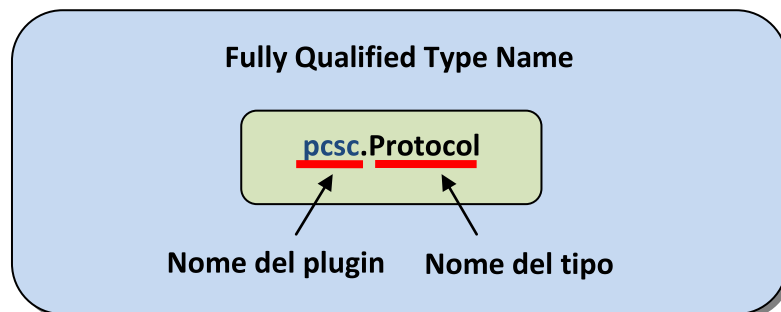


Figura 19 - Fully Qualified Type Name

Per ottenere maggiori informazioni relative ad un particolare tipo di dato è possibile utilizzare il comando *help*. L'esempio in Figura 20 mostra come ottenere i dettagli relativi al tipo *pcsc.Protocol*.

Nel riferirsi ad un tipo, tuttavia, la Shell non impone l'utilizzo della sintassi fully qualified ma permette anche di specificare solamente il nome del tipo. Sarà compito della Shell determinare il plugin in cui quel particolare tipo è stato definito (nell'esempio in Figura 20 si poteva semplicemente scrivere *Protocol* invece di *pcsc.Protocol* e ottenere gli stessi identici risultati). La Shell procede la

sua ricerca partendo dall'ultimo plugin caricato fino ad arrivare al primo; quindi se sono presenti due tipi di dato con lo stesso nome in plugin differenti, la Shell prediligerà il tipo di dato definito nel plugin caricato più recentemente. In quest'ultimo caso tuttavia è raccomandabile l'utilizzo della sintassi completa in modo da evitare qualsiasi ambiguità e rendere più leggibili i propri script.

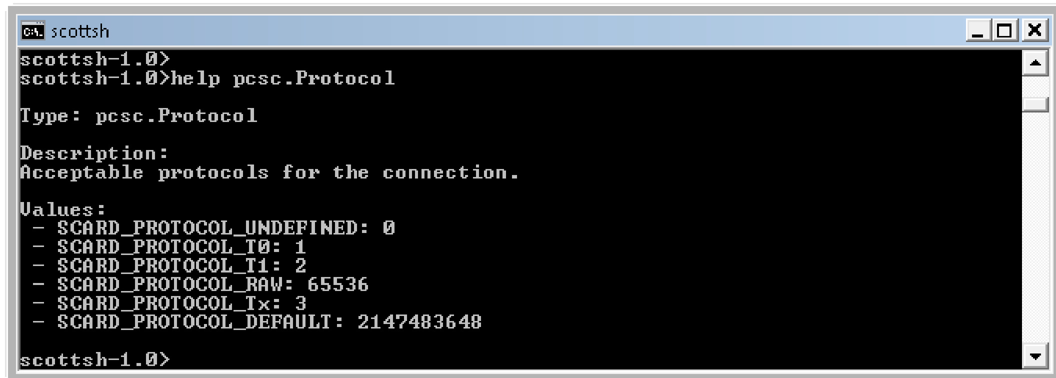


Figura 20 - Dettagli di un tipo

3.2.4 Creazione ed utilizzo delle variabili

Tramite il comando ***var*** la Shell permette di creare variabili. Per creare una variabile è necessario specificare il tipo, il nome ed una espressione d'inizializzazione (Figura 21). Le variabili in Scott sono fortemente tipizzate ed immutabili, cioè non è possibile cambiarne il valore dopo la creazione. Per questo motivo l'espressione d'inizializzazione è obbligatoria.

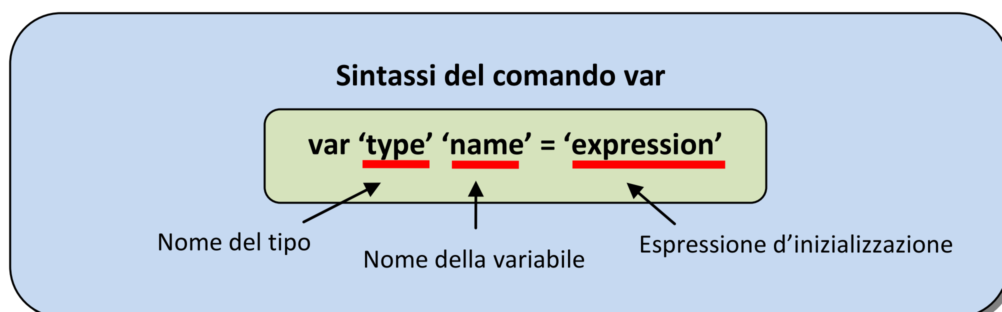


Figura 21 - Sintassi del comando 'var'

Il parametro ‘**type’** può essere il nome di un tipo primitivo (sbyte, snumber, schar, sstring) o il nome di un tipo complesso individuato tramite fully qualified type name o semplicemente con il nome del tipo lasciando alla shell il compito di inferire il plugin in cui è definito (come spiegato nel paragrafo 3.2.3).

Il parametro ‘**name’** rappresenta il nome che si vuole attribuire alla variabile. Tramite questo nome sarà possibile utilizzare all’interno della shell la variabile appena creata.

Il parametro ‘**expression’** infine permette di indicare il valore che si vuole assegnare alla nuova variabile scrivendolo direttamente nella riga del comando oppure prelevandolo da una variabile di shell precedentemente creata o ottenuta come risultato dell’esecuzione di un comando di plugin.

La Figura 22 mostra la creazione di alcune variabili di tipo primitivo. E’ possibile notare come la variabile m sia inizializzata prelevando il valore dalla variabile n precedentemente creata.

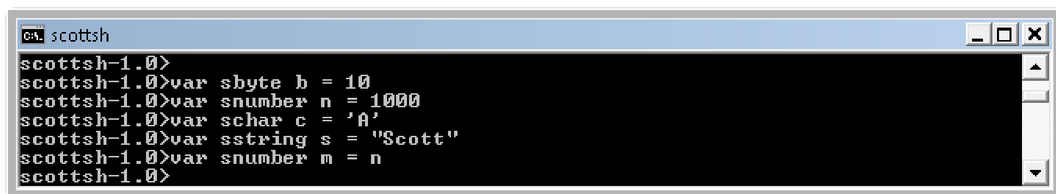


Figura 22 - Creazione di variabili di tipo semplice

Tramite il comando ***vars*** (Figura 23) è possibile visualizzare tutte le variabili di shell che sono state create.

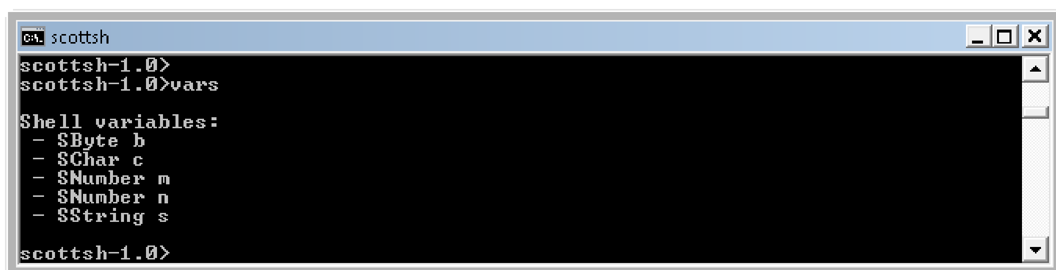


Figura 23 - Comando ‘vars’

Il comando *show* (Figura 24) permette di visualizzare il valore e il tipo di una particolare variabile.

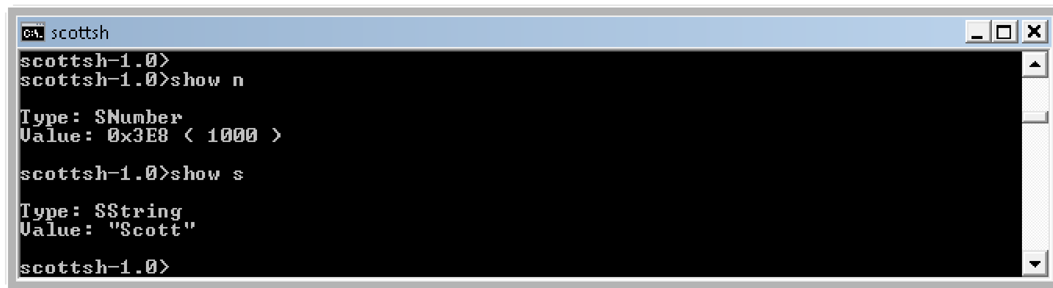


Figura 24 - Comando 'show'

La Figura 25 mostra come creare variabili di tipo *VNumber* (*SVector* di *SNumber*). Per accedere direttamente ad un elemento di un vettore si devono utilizzare le parentesi quadre e specificare la posizione dell'elemento d'interesse. Inoltre è anche possibile inizializzare un vettore utilizzando nella lista d'inizializzazione delle variabili create precedentemente ed individuate tramite una opportuna espressione.

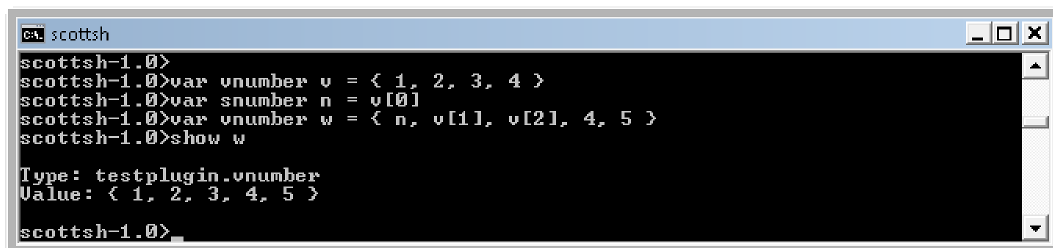


Figura 25 - Creare variabili vettoriali e accesso ai membri di un vettore

Consideriamo ora di avere a disposizione una SStruct di nome *point* che contiene due SNumber x e y. Consideriamo inoltre un tipo di nome *vpoint* che rappresenta un vettore di point. L'esempio in Figura 26 mostra come creare variabili di tipo SStruct e come scrivere espressioni complesse per accedere ai membri di una struttura.

```
scottsh
scottsh-1.0>
scottsh-1.0>var point p1 = { x=1, y=2 }
scottsh-1.0>var snumber x = p1.x
scottsh-1.0>var snumber y = p1.y
scottsh-1.0>var point p2 = { 3, 4 }
scottsh-1.0>var vpoint v = { {5,6}, {7,8} }
scottsh-1.0>var vpoint w = { {x,y}, p2, v[0], {v[1].x, v[1].y} }
scottsh-1.0>show w

Type: testplugin.vpoint
Value: { { x = 1, y = 2 }, { x = 3, y = 4 }, { x = 5, y = 6 }, { x = 7, y = 8 }
}

scottsh-1.0>
```

Figura 26 - Creazione di strutture ed espressioni complesse

3.2.5 Visualizzazione dei comandi disponibili

Il comando ***commands*** permette di visualizzare tutti i comandi definiti nei plugin che sono stati caricati tramite il comando load e che sono quindi direttamente utilizzabili.

La Figura 27 mostra un esempio di utilizzo del comando *commands* dopo che è stato caricato il plugin iso7816-4.

```
scottsh
scottsh-1.0>
scottsh-1.0>commands

Available commands:
 - iso7816-4.GetResponse()
 - iso7816-4.ReadBinary()
 - iso7816-4.Select()
 - iso7816-4.Transmit0()
 - iso7816-4.UpdateBinary()
 - iso7816-4.Verify()
 - pcsc.Connect()
 - pcsc.Disconnect()
 - pcsc.EstablishContext()
 - pcsc.ListReaders()
 - pcsc.ReleaseContext()
 - pcsc.Status()
 - pcsc.Transmit()

scottsh-1.0>
```

Figura 27 - Comando 'commands'

Per identificare in maniera univoca un comando di plugin si deve utilizzare quello che è chiamato **fully qualified command name**. Il nome effettivo di un comando di plugin è quindi costituito dal nome del plugin in cui è definito seguito da un punto e dal nome del comando (Figura 28).

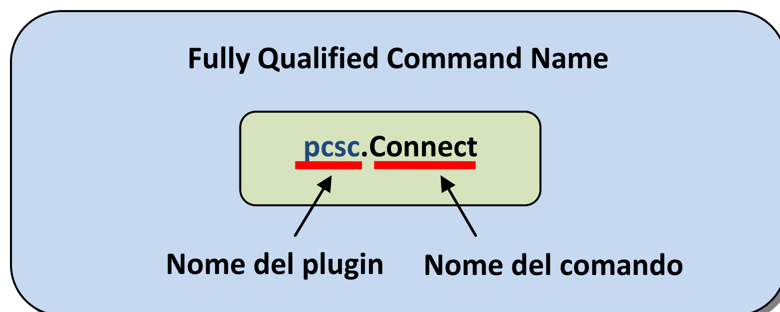


Figura 28 - Fully Qualified Command Name

Per ottenere maggiori informazioni relative ad un particolare comando di plugin è possibile utilizzare il comando *help*. L'esempio in Figura 29 mostra come ottenere i dettagli relativi al comando *pcsc.ListReaders*.

```
scottsh
scottsh-1.0>
scottsh-1.0>help pcsc.ListReaders

Command: pcsc.ListReaders

Description:
This command provides the list of readers.

Inputs:
 - SNumber context : Handle to the resource manager context.

Outputs:
 - pcsc.Error res : The command result.
 - pcsc.Vstring readers : The vector of reader names

Usage example:
[ res, readers ] = ListReaders( context )

scottsh-1.0>
```

Figura 29 - Ottenere i dettagli di un comando di plugin

La descrizione dettagliata di un comando comprende il nome, la descrizione, i parametri d'input e i parametri di output con relativa descrizione ed infine un esempio di utilizzo del comando stesso.

Nel riferirsi ad un comando, tuttavia, la Shell non impone l'utilizzo della sintassi fully qualified ma permette anche di specificare solamente il nome del comando. Sarà compito della Shell determinare il plugin in cui quel particolare comando è stato definito (nell'esempio in Figura 29 si poteva semplicemente scrivere *ListReaders* invece di *pcsc.ListReaders* e ottenere gli stessi identici risultati). La Shell procede la sua ricerca partendo dall'ultimo plugin caricato fino ad arrivare al primo quindi se sono presenti due comandi con lo stesso nome in plugin

differenti, la Shell prediligerà il comando definito nel plugin caricato più recentemente. In quest'ultimo caso tuttavia è raccomandabile l'utilizzo della sintassi completa in modo da evitare qualsiasi ambiguità e rendere più leggibili i propri script.

3.2.6 Esecuzione dei comandi di un plugin

La sintassi per eseguire un comando di plugin è mostrata in Figura 30.

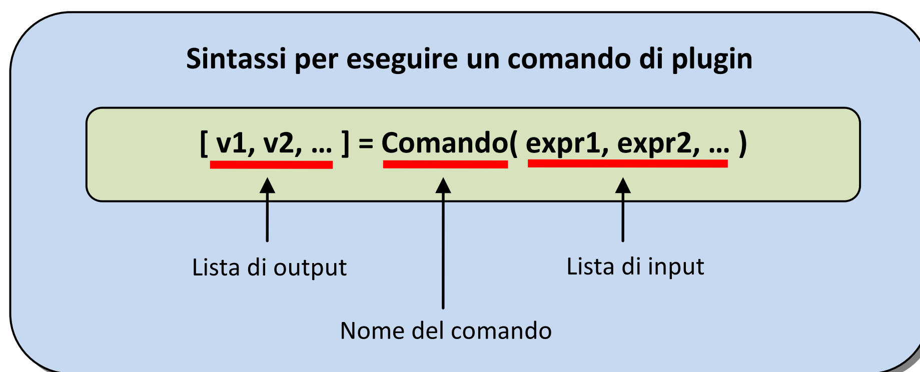


Figura 30 - Sintassi per eseguire un comando di plugin

Per eseguire un comando è necessario far seguire al nome del comando una coppia di parentesi tonde al cui interno si devono specificare i parametri di input mediante la cosiddetta lista di input. La lista di input consiste in un insieme di espressioni separate tra loro da una virgola. Ogni singola espressione rappresenta un parametro da passare al comando. La lista di input è equivalente ad un tipo SStruct i cui membri corrispondono uno ad uno ai parametri di input del comando. Per questo motivo è possibile specificare i parametri del comando anche senza rispettarne l'ordine di definizione utilizzando una sintassi del tutto analoga a quella delle strutture in modalità associativa.

La lista di output è facoltativa e permette di specificare i nomi delle variabili di shell che si desidera creare a partire dai risultati ottenuti dall'esecuzione del comando. Se la lista di output non è specificata, oppure è lasciata vuota tutti i risultati ottenuti dall'esecuzione del comando saranno semplicemente ignorati

(Questa affermazione è vera se *autovars* è impostata a false, vedi il paragrafo 3.2.7 per maggiori informazioni).

Se la lista di output contiene almeno un elemento, allora saranno create delle variabili di shell il cui valore d'inizializzazione corrisponde al valore delle variabili risultato dell'operazione. Le variabili inserite nella lista di output devono rispettare l'ordine di definizione delle variabili di output nel comando stesso. Nel caso in cui sia presente una sola variabile di output, è possibile omettere le parentesi quadrate.

La Figura 31 mostra un esempio di esecuzione di alcuni comandi del plugin pcsc. Questo esempio vuole solamente mostrare alcune sintassi utilizzabili per eseguire comandi senza entrare nel merito del significato dello specifico comando. Per maggiori informazioni relative ai comandi presenti nei vari plugin fare riferimento al capitolo 4.

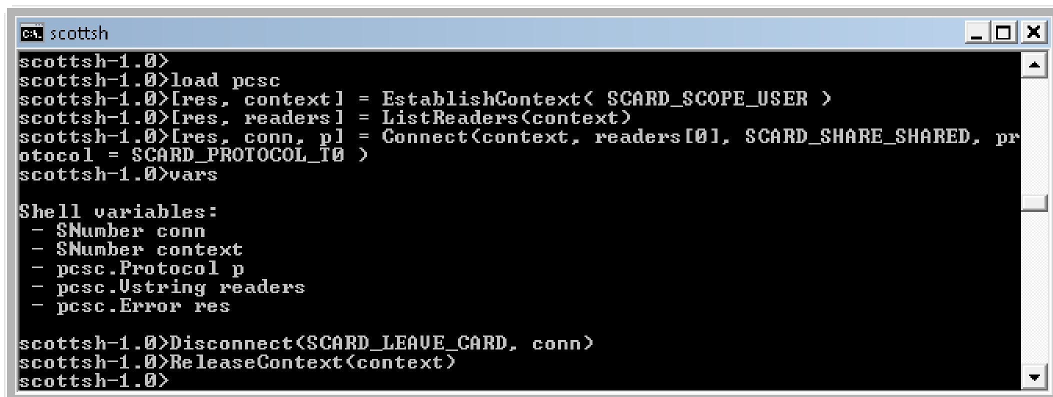


Figura 31 - Esempi di esecuzione comandi

3.2.7 Meccanismo delle variabili automatiche

Tramite il comando ***autovars*** è possibile attivare e disattivare il meccanismo delle variabili automatiche. Quando si esegue un comando senza specificare la lista di output e questo meccanismo è attivo, la shell automaticamente crea le variabili di output utilizzando come nome quello specificato nel descrittore del comando stesso. La Figura 32 mostra un esempio d'invocazione del comando *pcsc.EstablishContext()* per meglio comprendere il funzionamento di questo

meccanismo. L’esecuzione del comando *vars* permette di notare che sono state create automaticamente le due variabili di output context e res.

Figura 32 - Uso del meccanismo delle variabili automatiche

E’ importante dire che la lista di output ha la priorità rispetto al meccanismo delle variabili automatiche. Nel caso in cui è specificata una lista di output, nessuna variabile di shell sarà creata automaticamente ma saranno create solamente quelle specificate direttamente nella lista stessa (Figura 33).

Figura 33 - Meccanismo delle variabili automatiche ignorato

3.2.8 Meccanismo delle variabili implicite

Il comando ***implicitvars*** è complementare al comando *autovars* e permette di attivare e disattivare il meccanismo delle variabili implicite. Questo meccanismo offre all’utente la possibilità di non specificare alcuni o tutti i parametri nella lista di input di un comando. La shell passerà automaticamente al comando stesso tutte le variabili il cui nome corrisponde al nome di un parametro di input.

Il meccanismo delle variabili implicite unito a quello delle variabili automatiche consente a tutti gli effetti di eseguire un comando senza specificare la lista di

input e la lista di output. L'esempio in Figura 34 mostra l'utilizzo del comando *pcsc.Connect()* sfruttando questi due meccanismi combinati.

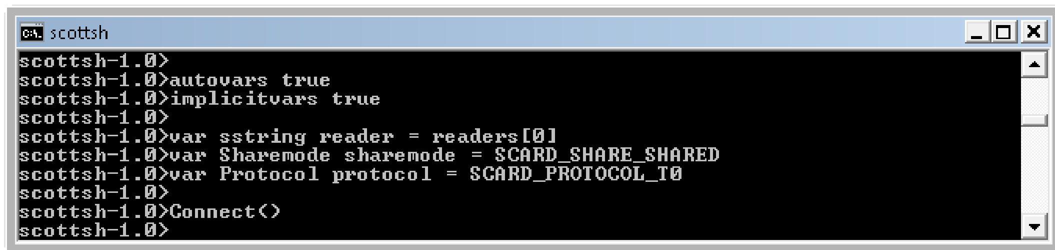


Figura 34 - Esempio di utilizzo delle variabili implicite

3.2.9 Visualizzazione automatica dei risultati

Quando si esegue un comando, i risultati sono memorizzati in opportune variabili di shell. Si è quasi sempre interessati al valore di questi risultati e per poterli visualizzare si deve ricorrere al comando *show* per un numero di volte pari al numero di risultati cui si è interessati (Figura 35).

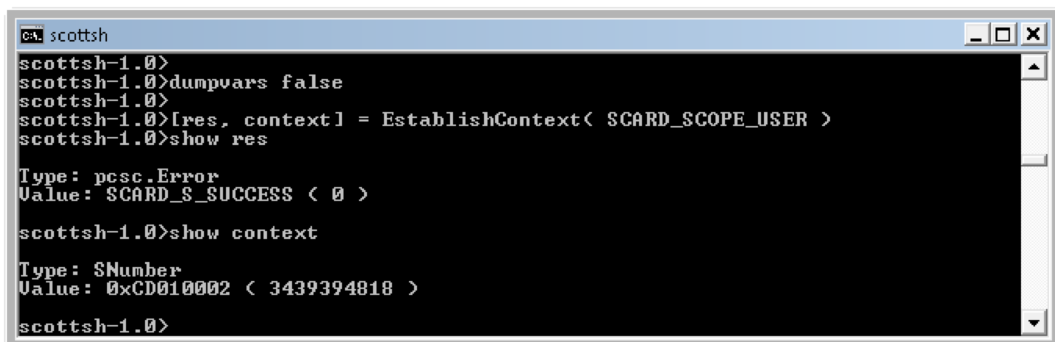


Figura 35 - Visualizzazione dei risultati con dumpvars a false

Attivando la visualizzazione automatica dei risultati tramite il comando ***dumpvars***, la shell visualizza automaticamente il valore delle nuove variabili che sono create dopo l'esecuzione di un comando (Figura 36).

Figura 36 - Visualizzazione dei risultati con dumpvars a true

Questo meccanismo è utile principalmente durante un utilizzo interattivo della shell e permette di risparmiare all'utente l'incombenza di chiamare il comando show diverse volte dopo ogni invocazione di comando.

3.2.10 Visualizzazione delle impostazioni della shell

Tramite il comando ***settings*** (Figura 37) è possibile controllare le impostazioni correnti della shell e in particolare l'attivazione o la disattivazione del meccanismo delle variabili automatiche, del meccanismo delle variabili implicite e del meccanismo di visualizzazione automatica dei risultati.

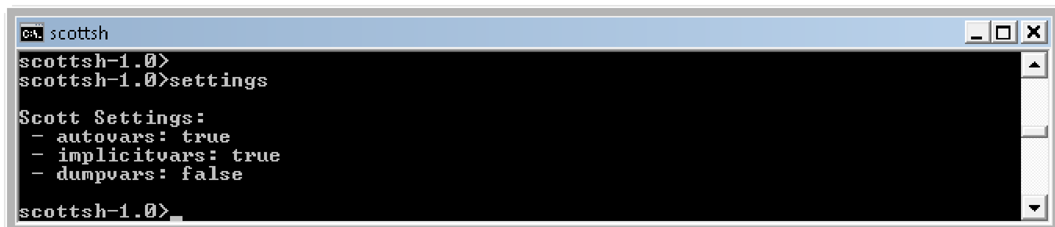


Figura 37 - Comando 'settings'

3.2.11 Stampa ed esportazione dei dati su file

Per visualizzare il valore di una variabile di shell è stato già presentato il comando *show*. Esiste un comando più avanzato per visualizzare il valore di una o più variabili di shell in modo formattato e completamente personalizzato. Il comando che permette di fare questo si chiama ***write*** e la Figura 38 mostra la sintassi di utilizzo.

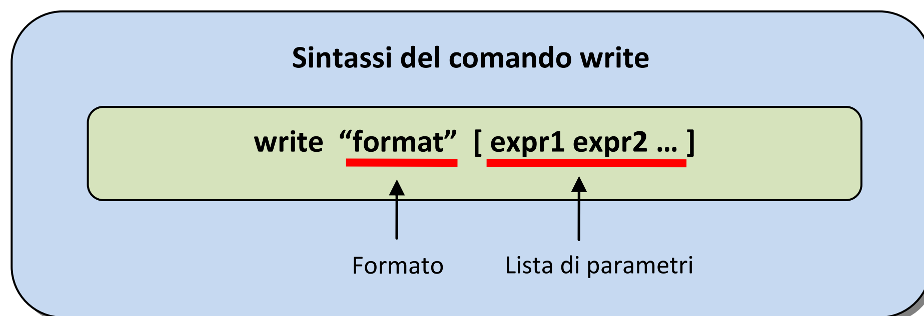


Figura 38 - Sintassi del comando 'write'

La Figura 39 mostra l'utilizzo più semplice del comando *write* che comporta la scrittura a video del testo scritto nella stringa di formato.

```
scottsh
scottsh-1.0>
scottsh-1.0>write "Scott"
Scott
scottsh-1.0>
```

Figura 39 - Uso banale di 'write'

La stringa di formattazione è costituita quasi interamente dal testo da stampare ma dove deve essere formattata una variabile si deve inserire, tra parentesi graffe, il suo indice nella lista dei parametri. All'interno delle parentesi graffe si può inoltre inserire uno specificatore di formato, preceduto da una virgola, che indica come si vuole che l'elemento sia formattato. Ad esempio è possibile specificare che un numero sia stampato in esadecimale (specificatore x) piuttosto che decimale (specificatore d). Questa sintassi prende spunto da quella del comando *`String.Format()`* presente nella libreria di base del framework .NET (String.Format in MSDN s.d.). La Figura 40 mostra come stampare il valore di alcune variabili primitive.

Nella lista di argomenti è possibile specificare qualsiasi tipo di espressione. Questa possibilità offre una grande flessibilità nel visualizzare i dati. Supponiamo di avere a disposizione una SStruct di nome *point* che contiene due SNumber x e

y e un tipo di nome *vpoint* che rappresenta un vettore di point. L'esempio in Figura 41 mostra l'utilizzo del comando *write* con queste variabili complesse.

```
scottsh-1.0>
scottsh-1.0>var snumber n = 10
scottsh-1.0>var schar c = 'A'
scottsh-1.0>var sstring s = "Scott"
scottsh-1.0>
scottsh-1.0>write "Numero: {0}" n
Numero: 10
scottsh-1.0>write "Carattere: {0}" c
Carattere: 'A'
scottsh-1.0>write "Stringa: {0}" s
Stringa: "Scott"
scottsh-1.0>write "Numero esadecimale: {0,x} o {0,nx}" n
Numero esadecimale: 0xA o A
scottsh-1.0>write "n={0,x}, c={1}, s={2}" n c s
n=0xA, c='A', s="Scott"
scottsh-1.0>
scottsh-1.0>
```

Figura 40 - Stampa di tipi primitivi

```
scottsh-1.0>
scottsh-1.0>var point p = {1,2}
scottsh-1.0>var point q = {3,4}
scottsh-1.0>var vpoint v = {p,q}
scottsh-1.0>
scottsh-1.0>write "p = {0}" p
p = { x = 1, y = 2 }
scottsh-1.0>write "q = {0}" q
q = { x = 3, y = 4 }
scottsh-1.0>write "v = {0}" v
v = { { x = 1, y = 2 }, { x = 3, y = 4 } }
scottsh-1.0>
scottsh-1.0>write "p.x={0}, p.y={1}\nv\{1\}.x={2}" p.x p.y v[1].x
p.x=1, p.y=2
v{1}.x=3
scottsh-1.0>
scottsh-1.0>
```

Figura 41 - Stampa di tipi complessi

Dall'esempio è possibile notare l'utilizzo di sequenze di escape per riuscire a stampare le parentesi graffe e per inserire un "ritorno a capo". La Tabella 7 mostra le sequenze di escape che sono supportate dal comando.

Esiste un comando analogo a *write* per esportare su file le variabili di shell. Questo comando si chiama ***fwrite*** ed è perfettamente equivalente a *write* dal punto di vista sintattico. Prima di utilizzare *fwrite* si deve chiamare il comando *fopen* che permette di specificare il file in cui si vuole scrivere. Il comando *fopen* sostanzialmente permette di aprire una sessione di esportazione dati. Per terminare la sessione di esportazione si deve utilizzare il comando *fclose*. Un esempio di sessione è mostrato in Figura 42.

Sequenza di escape	Descrizione
\{	{
\}	}
\n	Nuova linea
\r	Ritorno del carrello
\t	Tabulazione
****	\

Tabella 7 - Sequenze di escape supportate dal comando write

```
scottsh-1.0>
scottsh-1.0>var snumber n = 10
scottsh-1.0>var point p = {1,2}
scottsh-1.0>var point q = {3,4}
scottsh-1.0>var vpoint v = {p, q}
scottsh-1.0>
scottsh-1.0>fopen out.txt
scottsh-1.0>fwrite "Ecco i risultati esportati\n\n"
scottsh-1.0>fwrite "n = {0}\n" n
scottsh-1.0>fwrite "p = {0}\n" p
scottsh-1.0>fwrite "q = {0}\n" q
scottsh-1.0>fwrite "v\{0\} = {0}\n" v[0]
scottsh-1.0>fwrite "v\{1\} = \{ {0,x}, {1,x} \}\n" v[1].x v[1].y
scottsh-1.0>fclose
scottsh-1.0>
```

Figura 42 - Sessione di esportazione dati

La Figura 43 mostra il contenuto del file "out.txt" in cui sono stati scritti i dati esportati. I comandi per l'esportazione dei dati rivestono un ruolo di estrema importanza per integrare la shell Scott all'interno di script di test automatici. Lo sviluppatore dei test creerà una sessione di Scott inviando opportuni comandi, salverà i risultati su file e poi verificherà che i dati esportati siano quelli attesi. Quest'operazione di verifica potrà essere realizzata senza dover obbligatoriamente chiudere la sessione corrente della shell.

```
Ecco i risultati esportati

n = 10
p = { x = 1, y = 2 }
q = { x = 3, y = 4 }
v{0} = { x = 1, y = 2 }
v{1} = { 0x3, 0x4 }
```

Figura 43 - Risultato dell'esportazione dati (file out.txt)

Esistono due varianti di *write* e *fwrite* che sono ***writeln*** e ***fwriteln***. Questi due ulteriori comandi sono perfettamente equivalenti ai precedenti con la sola differenza che il testo stampato o aggiunto al file è seguito da un ritorno a capo senza che sia necessario specificarlo nella stringa di formato.

In Tabella 8, Tabella 9, Tabella 10, Tabella 11, Tabella 12, Tabella 13 e Tabella 14 si riportano tutti gli specificatori di formato disponibili.

Formato	Descrizione	Esempio
Nessuno	Visualizza il valore in decimale.	10
d	Visualizza il valore in decimale.	10
x	Visualizza il valore in esadecimale, con prefisso, utilizzando il minor numero possibile di caratteri.	0xA
nx	Visualizza il valore in esadecimale, senza prefisso, utilizzando il minor numero di caratteri possibili.	A
<num>x	Visualizza il valore in esadecimale, con prefisso, utilizzando per la parte numerica il numero di caratteri specificati da <num>. Lo '0' è il carattere di riempimento.	0x000A
<num>nx	Visualizza il valore in esadecimale, senza prefisso, utilizzando per la parte numerica il numero di caratteri specificati da <num>. Lo '0' è il carattere di riempimento.	000A
a	Interpreta il byte come una carattere in codifica ASCII. Un carattere non stampabile è visualizzato con un punto.	.

Tabella 8 - Specificatori di formato per SByte

Formato	Descrizione	Esempio
Nessuno	Visualizza il valore in decimale.	123
d, x, nx, <num>x, <num>nx	Vedi Tabella **8**.	-

Tabella 9 - Specificatori di formato per SNumber

Formato	Descrizione	Esempio
Nessuno	Visualizza il carattere racchiuso tra apici singoli.	'Z'
n	Visualizza il carattere senza racchiuderlo tra apici.	Z

Tabella 10 - Specificatori di formato per SChar

Formato	Descrizione	Esempio
Nessuno	Visualizza la stringa racchiusa tra apici doppi.	“Scott”
n	Visualizza la stringa senza racchiuderla tra apici.	Scott

Tabella 11 - Specificatori di formato per SString

Formato	Descrizione	Esempio
Nessuno	Visualizza la costante di denominazione dell’enumeratore.	SCARD_PROTOCOLO_T0
d, x, nx, <num>x, <num>nx	Visualizza il valore dell’enumeratore. Vedi Tabella **8**.	-

Tabella 12 - Specificatori di formato per SEnum

Formato	Descrizione	Esempio
Nessuno	Visualizza la struttura su una singola linea in modalità associativa. Ogni elemento è visualizzato senza una precisa formattazione.	{x=10, y=20}
pp	Visualizza la struttura in modalità “pretty print”. Ogni elemento è visualizzato su una riga diversa e per i numeri è mostrata sia la rappresentazione esadecimale sia decimale.	-

Tabella 13 - Specificatori di formato per SStruct

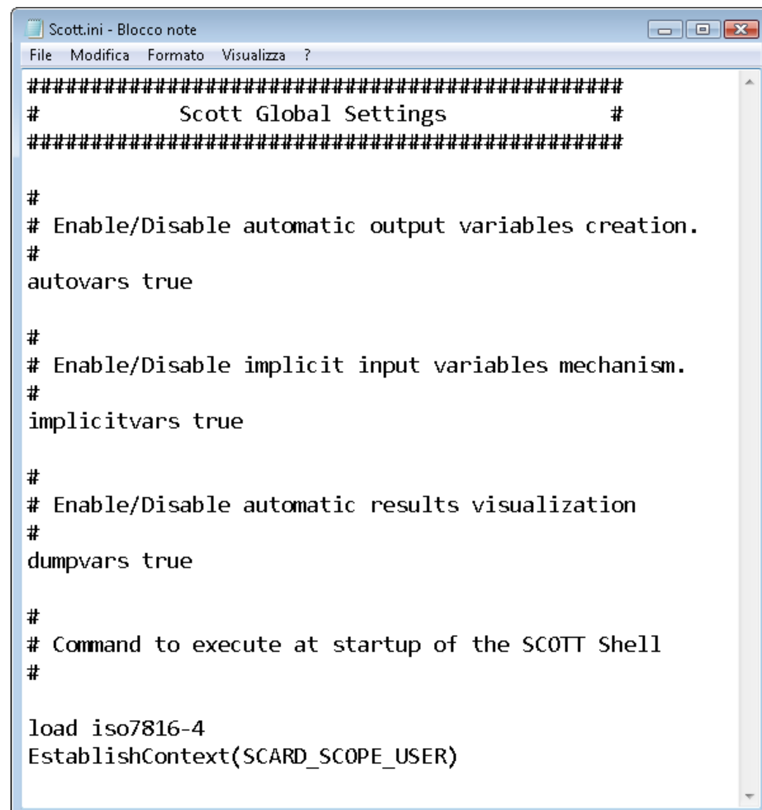
Formato	Descrizione	Esempio
Nessuno	Visualizza il vettore su una singola linea. Ogni elemento è visualizzato senza una precisa formattazione.	{18,52,86,120}
x	Applicabile solo su vettori di SByte, visualizza la rappresentazione esadecimale del vettore con prefisso. Ogni byte è visualizzato sempre con due caratteri.	0x12345678
nx	Applicabile solo su vettori di SByte, visualizza la rappresentazione esadecimale del vettore senza prefisso. Ogni byte è visualizzato con due caratteri.	12345678
<num>x	Applicabile solo su vettori di SByte visualizza con il formato “x” solamente i primi <num> byte.	0x1234
<num>nx	Applicabile solo su vettori di SByte visualizza con il formato “nx” solamente i primi <num> byte.	1234
a	Applicabile solo su vettori di SByte, interpreta ogni byte come un carattere ASCII e visualizza la stringa corrispondente. Un carattere non stampabile è visualizzato con un punto.	.4Vx
pp	Visualizza il vettore in modalità “pretty print”. Ogni elemento è visualizzato su una riga diversa e per i numeri è mostrata sia la rappresentazione esadecimale sia decimale.	-

Tabella 14 - Specificatori di formato per SVector

3.2.12 Esecuzione automatica di comandi all’avvio

E’ possibile scrivere un file di configurazione i cui comandi saranno eseguiti all’avvio della shell. In ambiente Windows questo file si chiama **Scott.ini** e deve essere creato nella cartella d’installazione di Scott. In ambiente Linux questo file di chiama Scottrc e deve essere creato all’interno della cartella *etc* sotto la cartella d’installazione di Scott.

La Figura 44 mostra un esempio di file di configurazione. All’interno di questo file è possibile inserire qualsiasi comando supportato dalla shell. Mediante il carattere # è inoltre possibile inserire dei commenti che saranno ignorati durante l’esecuzione. Provando ad eseguire la shell in presenza di questo file di configurazione si ottengono i risultati mostrati in Figura 45. Nel caso in cui non sia presente un file di configurazione le impostazioni autovars, implicitvars e dumpvars assumono un valore true per default.




```
##############################################
#           Scott Global Settings            #
##############################################

#
# Enable/Disable automatic output variables creation.
#
autovars true

#
# Enable/Disable implicit input variables mechanism.
#
implicitvars true

#
# Enable/Disable automatic results visualization
#
dumpvars true

#
# Command to execute at startup of the SCOTT Shell
#

load iso7816-4
EstablishContext(SCARD_SCOPE_USER)
```

Figura 44 - File di configurazione della shell

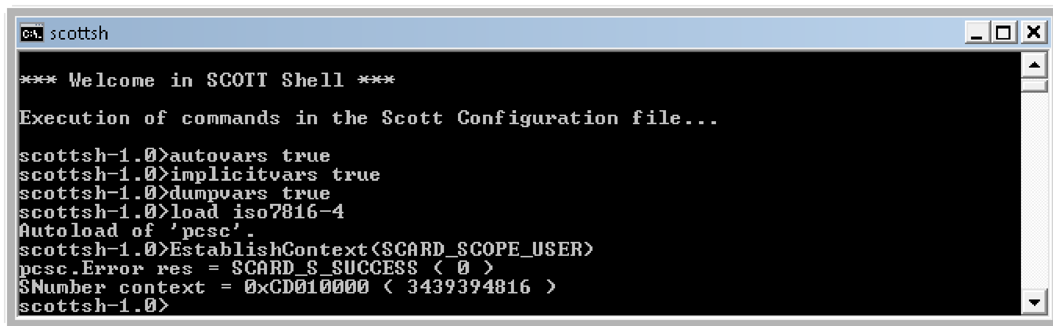


Figura 45 - Avvio della shell con file di configurazione

Una volta compresi tutti i comandi di shell è il momento di entrare nel merito dei plugin che offrono le funzionalità necessarie per l’interfacciamento con le smartcard.

4. Guida all’utilizzo dei plugin di SCOTT

4.1 Il plugin “pcsc”

Il plugin **pcsc** offre una serie di comandi di base per accedere alle smartcard. Questo plugin sostanzialmente rappresenta un wrapper semplificato alle API PC/SC presentate nel paragrafo 1.6.2.

Tutti i comandi del plugin restituiscono almeno una istanza del tipo enumerato *pcsc.Error* la quale permette di verificare se l’operazione è andata a buon fine (SCARD_S_SUCCESS) oppure se si è in presenza di un errore. Per conoscere tutti i possibili errori che possono essere generati digitare il comando “help pcsc.Error” o fare riferimento allo standard PC/SC.

Nei prossimi paragrafi assumeremo che autovars, implicitvars e dumpvars siano impostati a true e che il plugin pcsc sia stato opportunamente caricato.

4.1.1 Tipi definiti dal plugin “pcsc”

Il plugin *pcsc* definisce i seguenti tipi di dato:

- **Buffer**: un vettore di SByte.
- **Vstring**: un vettore di SString.
- **Error**: un enumeratore che definisce tutti i possibili codici di errore che possono essere restituiti dai vari comandi del plugin. Per un elenco completo dei valori utilizzare il comando “help pcsc.Error”.
- **Disposition**: un enumeratore che è utilizzato per specificare cosa fare con la carte nel lettore quando viene chiusa la connessione. I possibili valori sono :
    - SCARD_LEAVE_CARD, SCARD_RESET_CARD
    - SCARD_UNPOWER_CARD, SCARD_EJECT_CARD.

- **Protocol**: un enumeratore che permette di specificare i protocolli utilizzati nella comunicazione. I possibili valori sono:
    - SCARD_PROTOCOL_UNDEFINED, SCARD_PROTOCOL_DEFAULT
    - SCARD_PROTOCOL_T0, SCARD_PROTOCOL_T1,
    - SCARD_PROTOCOL_RAW, SCARD_PROTOCOL_Tx
- **Scope**: un enumeratore che permette di specificare lo scope del contesto del Resource Manager. I possibili valori sono:
    - SCARD_SCOPE_USER, SCARD_SCOPE_SYSTEM
- **Sharemode**: un enumeratore che permette di specificare se altre applicazioni possono connettersi alla carta o meno. I possibili valori sono:
    - SCARD_SHARE_EXCLUSIVE
    - SCARD_SHARE_SHARED
    - SCARD_SHARE_DIRECT
- **State**: un enumeratore per rappresentare i possibili stati della smartcard all'interno del lettore. I possibili valori sono:
    - SCARD_UNKNOWN, SCARD_ABSENT
    - SCARD_PRESENT, SCARD_SWALLOWED
    - SCARD_POWERED, SCARD_NEGOTIABLE
    - SCARD_SPECIFIC

I valori assegnabili ai tipi *Error*, *Disposition*, Protocol, *Scope*, *Sharemode* e *State* corrispondono esattamente alle costanti definite dallo standard PC/SC.

4.1.2 Il comando EstablishContext()

Il comando ***EstablishContext()*** permette di ottenere il riferimento al resource manager. Il comando possiede un unico argomento di input di tipo *pcsc.Scope* per specificare lo scope con cui accedere al resource manager. Oltre al codice di errore il comando restituisce un *SNumber* denominato context che rappresenta l'handler al contesto del resource manager. Questo valore dovrà spesso essere

fornito come input agli altri comandi *pcsc*. In Figura 46 è mostrato un esempio di utilizzo.

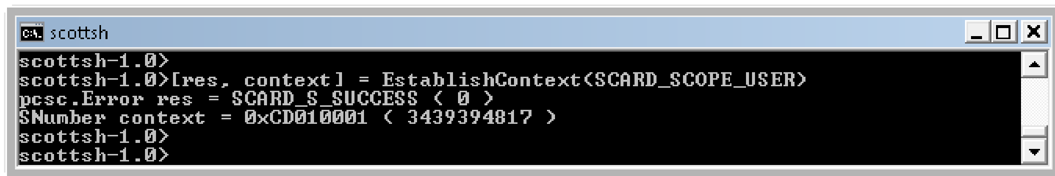


Figura 46 - Comando pcsc.EstablishContext()

Per ottenere maggiori informazioni digitare "help pcsc.EstablishContext".

4.1.3 Il comando ListReaders()

Il comando ***ListReaders()*** restituisce l'elenco dei lettori di smartcard sotto forma di un vettore di stringhe (tipo *Vstring*). L'unico argomento di input è il context ottenuto dalla precedente chiamata al comando *EstablishContext()*. La Figura 47 mostra un esempio di utilizzo.

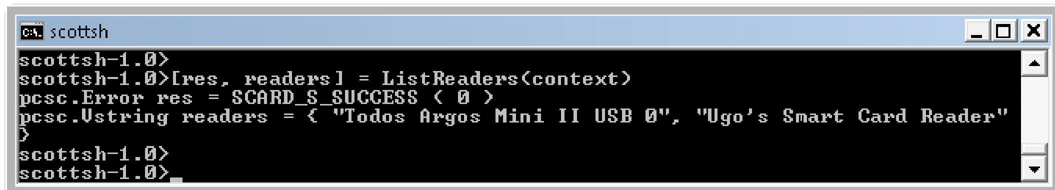


Figura 47 - Comando pcsc.ListReaders()

Per ottenere maggiori informazioni digitare "help pcsc.ListReaders".

4.1.4 Il comando Connect()

Il comando ***Connect()*** permette di aprire una connessione con una smartcard inserita in uno specifico lettore scelto tra quelli forniti dal comando *ListReaders()*.

Per eseguire il comando è necessario fornire i seguenti input:

- **context**: handle che identifica il contesto del resource manager ottenuto da una precedente chiamata al metodo *EstablishContext()*.
- **reader**: il nome del lettore di smartcard sotto forma di SString. Conviene prelevare questo valore a partire dal vettore di stringhe risultato della chiamata al comando *ListReaders()*.
- **sharemode**: specifica se la modalità di connessione alla scheda è esclusiva o meno tramite una istanza del tipo *pcsc.Sharemode*.
- **protocol**: specifica i protocolli da utilizzare per la comunicazione con la scheda tramite una istanza del tipo *pcsc.Protocol*.

Il comando, oltre al codice di errore, restituisce un handler alla connessione appena stabilita, che dovrà essere utilizzato in seguito come input per altri comandi *pcsc*, e restituirà il protocollo scelto per la comunicazione. La Figura 48 mostra un esempio di utilizzo.

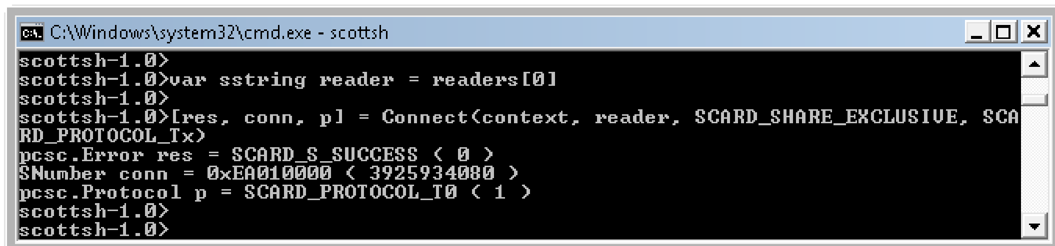


Figura 48 - Comando pcsc.Connect()

Per ottenere maggiori informazioni digitare “help pcsc.Connect”.

4.1.5 Il comando Status()

Il comando ***Status()*** fornisce lo stato corrente di una smartcard in un lettore. Si può chiamare questo comando quante volte si desidera dopo una chiamata a *Connect()* avvenuta con successo. L’input è l’handler di connessione ottenuto con la precedente chiamata al comando *Connect()*.

Le informazioni di stato restituite sono:

- **reader**: il nome del lettore correntemente connesso memorizzato in una istanza di tipo SString.
- **state**: lo stato corrente della smartcard nel lettore.
- **protocol**: protocollo utilizzato per la comunicazione.
- **atr**: ATR della scheda sotto forma di vettore di SByte (Buffer).

La Figura 49 mostra un esempio di utilizzo di questo comando.

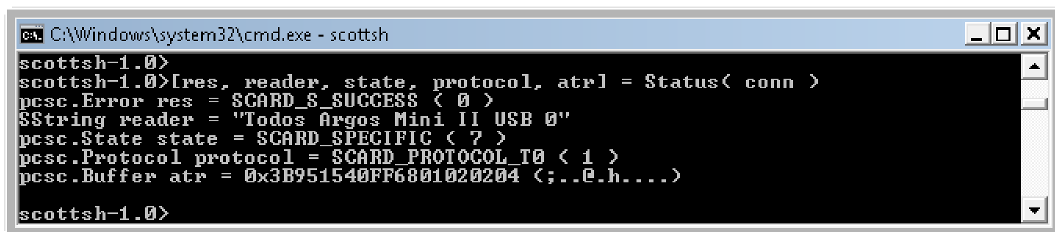


Figura 49 - Comando pcsc.Status()

Per ottenere maggiori informazioni digitare “help pcsc.Status”.

4.1.6 Il comando Transmit()

Il comando ***Transmit()*** è il comando principale del plugin pcsc e permette di inviare una sequenza di byte alla smartcard ed ottenere una risposta dalla stessa.

Il comando richiede i seguenti input:

- **data**: dati da inviare alla smartcard sotto forma di vettore di SByte.
- **connection**: handler alla connessione ottenuta precedentemente come risultato del comando Connect().
- **protocol**: protocollo in uso.

Il comando, oltre al codice di errore (*res*), restituisce un vettore di byte (*receiveBuffer*) in cui è contenuta la risposta inviata dalla scheda. Gli ultimi due byte di questo vettore contengono generalmente un codice che permette di

sapere se l'operazione è andata a buon fine, in tal caso il codice sarà pari al valore 0x9000.

La Figura 50 mostra un esempio di utilizzo del comando *Transmit()* per effettuare l'autenticazione alla scheda Cryptoflex8 e leggere il codice seriale della scheda.

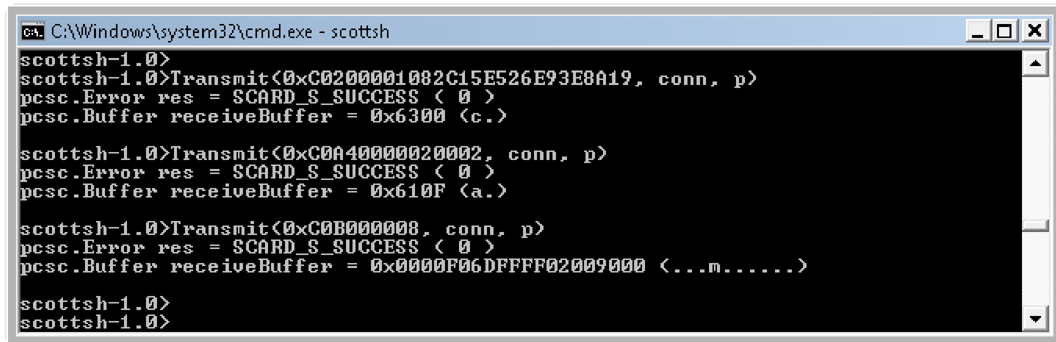


Figura 50 - Comando pcsc.Transmit()

Per ottenere maggiori informazioni digitare "help pcsc.Transmit".

4.1.7 Il comando Disconnect()

Il comando ***Disconnect()*** permette di chiudere una connessione precedentemente aperta tramite il comando *Connect()*. In input è necessario specificare l'handler alla connessione e un enumeratore di tipo *pcsc.Disposition* che specifica cosa fare con la carta all'interno del lettore.

La Figura 51 mostra una esecuzione del comando *Disconnect()*.

Figura 51 - Comando pcsc.Disconnect()

Per ottenere maggiori informazioni digitare "help pcsc.Disconnect".

4.1.8 Il comando ReleaseContext()

Il comando ***ReleaseContext()*** chiude il contesto del resource manager stabilito con la chiamata al metodo *EstablishContext()*. L'unico parametro da fornire al comando è il riferimento al context.

La Figura 52 mostra l'esecuzione del comando *ReleaseContext()*.

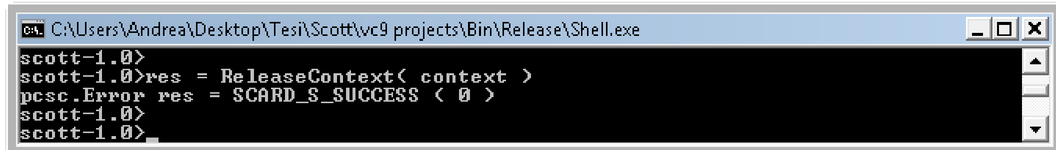


Figura 52 - Comando pcsc.ReleaseContext()

Per ottenere maggiori informazioni digitare "help pcsc.ReleaseContext".

4.1.9 Un esempio completo di utilizzo

Siccome le variabili context e connection devono quasi sempre essere fornite come input ai comandi del plugin pcsc, il meccanismo delle variabili implicite permette di evitare il passaggio di queste variabili ai comandi, aumentando la leggibilità degli script di shell e permettendo all'utente di concentrarsi sui parametri più significativi.

Il plugin *pcsc* inoltre, come è buona norma fare, è stato progettato in modo tale che il nome delle variabili di output di un comando che devono essere fornite come input di altri hanno lo stesso nome del parametro di input corrispondente. Per questo motivo, attivando anche il meccanismo delle variabili automatiche è possibile eseguire i comandi del plugin pcsc ignorando completamente gli handler che vengono creati. Tutto questo permette di semplificare notevolmente l'utilizzo delle API PC/SC per comunicare con le smartcard.

La Figura 53 mostra un esempio di script che sfrutta questo meccanismo per eseguire tutti i comandi pcsc. L'esecuzione di questo script con l'impostazione *autodump* attiva è mostrata nella Figura 54.

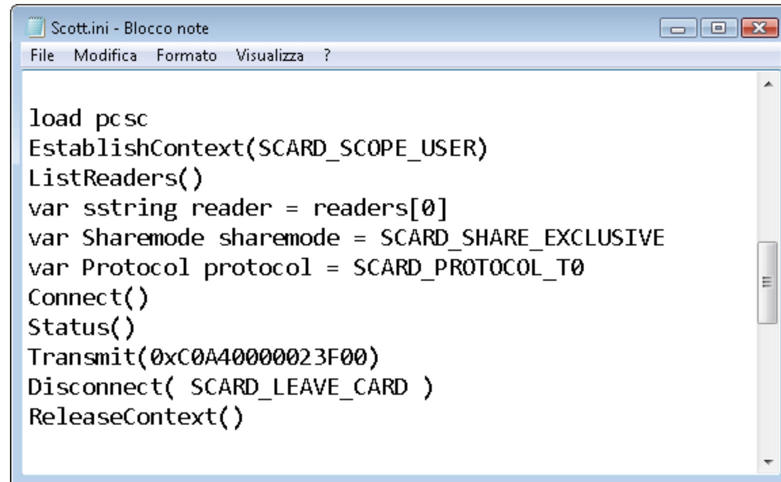


Figura 53 - Riepilogo comandi pcsc

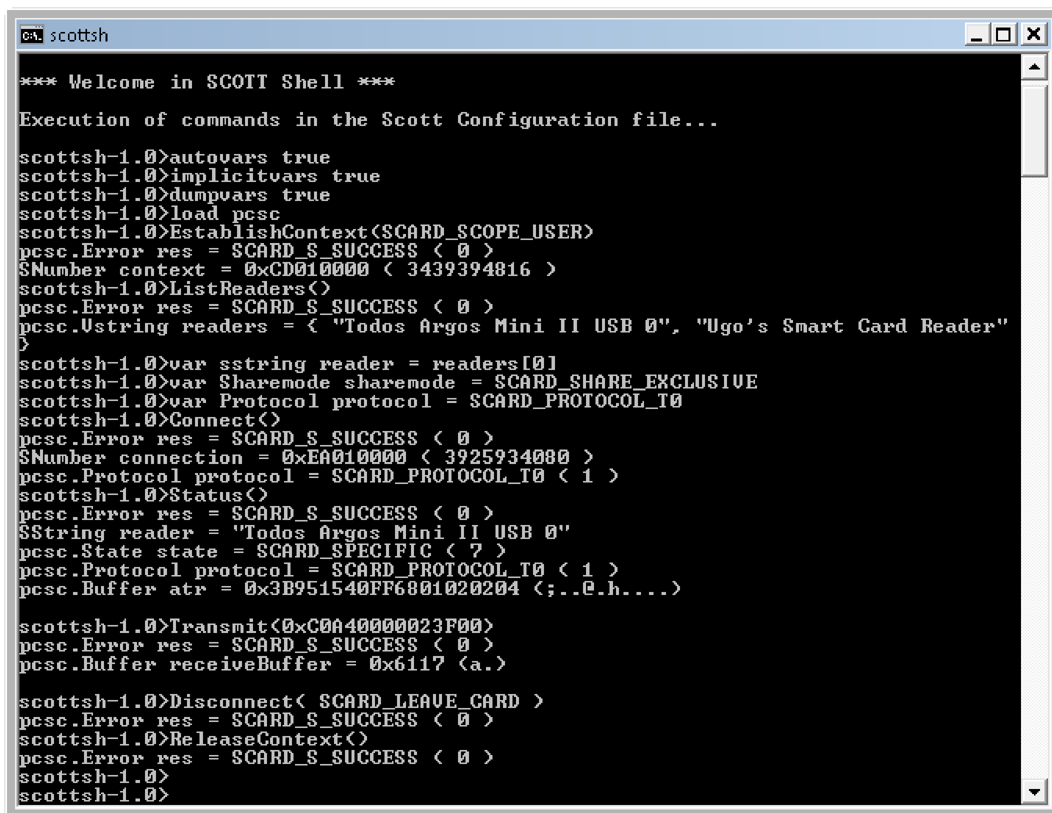


Figura 54 - Esecuzione del riepilogo di comandi pcsc

4.2 Il plugin “iso7816-4”

La maggioranza delle applicazioni che utilizzano smart card sono basate su file. Un’applicazione di questo tipo normalmente prende la forma di un insieme di file localizzati in una o più directory. Ogni file e directory è identificato tramite un ID numerico a sedici bit.

Nello standard si utilizza il termine **Dedicated File** (DF) per indicare una directory mentre si utilizza il termine **Elementary File** (EF) per indicare un file. La directory root del file system è chiamata **Master File** (MF) ed ha un identificatore riservato pari a 0x3F00. Lo standard definisce differenti tipologie di file; quelli più utilizzati sono i ***Transparent File***, che contengono una sequenza di byte non strutturata e sono utilizzati principalmente per memorizzare chiavi e certificati.

Lo standard ISO7816-4 presenta un insieme di comandi necessari per la manipolazione dei file e definisce il formato dei messaggi che serve per gestire l’input e l’output verso la scheda attraverso il protocollo standard T=0.

Un comando è inviato alla scheda sotto forma di un pacchetto che prende il nome di APDU (Application Protocol Data Unit).

Il lettore invia alla smartcard la classe del comando (CLA), l’identificatore del comando (COM) e tre parametri di input (P1, P2 e P3) e si mette in attesa di una risposta. La carta risponde inviando l’identificatore del comando come acknowledgment che è pronta a ricevere dati. Il lettore invia il vettore di input della lunghezza specificata in P3. Quando la scheda ha ricevuto correttamente questa quantità di dati, invia una conferma sotto forma di status word (SW1 e SW2). La Figura 55 mostra l’invio di un messaggio di input secondo il protocollo standard T=0.

Input Command-Response Message								
Reader	**Cla**	**Com**	**P1**	**P2**	**P3**		**Data**	
Card						Ack		SW1SW2
Length	1 B	1 B	1 B	1 B	1 B	1 B	*lgth* B	2 B

Figura 55 - Messaggio di input secondo il protocollo T=0

La Figura 56 mostra l'invio di un messaggio di output secondo il protocollo T=0. In questo caso è la scheda a restituire un insieme di byte come risultato.

Output Command-Response Message								
Reader	**Cla**	**Com**	**P1**	**P2**	**P3**			
Card						Ack	Data	SW1SW2
Length	1 B	1 B	1 B	1 B	1 B	1 B	*lgth* B	2 B

Figura 56 - Messaggio di Output secondo il protocollo T=0

La Tabella 15 mostra i principali comandi definiti dallo standard ISO7816-4. Nell'ambito di questa tesi sono stati implementati solamente i comandi necessari all'autenticazione tramite PIN e alla manipolazione dei transparent file.

Tutti i comandi definiti in questo plugin hanno come parametro di input il byte CLA in quanto lo standard non definisce un unico valore possibile per questo byte. Grazie al meccanismo delle variabili implicite è possibile creare una variabile di tipo SByte e di come CLA per evitare di passare ogni volta questo parametro quando s'invocano i comandi del plugin. Nei successivi paragrafi considereremo che questa variabile esista e sia definita con il valore richiesto dalla specifica smartcard che si sta utilizzando.

Tutti i comandi definiti in questo plugin restituiscono sempre almeno i seguenti tre parametri di output:

- **sw1** e **sw2**: le status word che insieme forniscono un codice per indicare una specifica condizione di errore. La Tabella 15 mostra le principali classi di valori che possono assumere questi due byte.

- **desc**: una descrizione sotto forma di stringa dell'errore; è praticamente una interpretazione delle status word secondo quanto riportato dallo standard.

Comando	INS	Implementato nel plugin
ERASE BINARY	0x0E	NO
VERIFY	0x20	SI
MANAGE CHANNEL	0x70	NO
EXTERNAL AUTHENTICATE	0x82	NO
GET CHALLENGE	0x84	NO
INTERNAL AUTHENTICATE	0x88	NO
SELECT FILE	0xA4	SI
READ BINARY	0xB0	SI
READ RECORD	0xB2	NO
GET RESPONSE	0xC0	SI
ENVELOPE	0xC2	NO
GET DATA	0xCA	NO
WRITE BINARY	0xD0	SI
WRITE RECORD	0xD2	NO
UPDATE BINARY	0xD6	NO
PUT DATA	0xDA	NO
UPDATE RECORD	0xDC	NO
APPEND RECORD	0xE2	NO

Tabella 15 - Principali comandi definiti dallo standard ISO7816-4

Per maggiori informazioni sullo standard ISO7816-4 fare riferimento alla bibliografia (International Standard Organization. ISO/IEC 7816-4: Identification cards - Integrated circuit cards - Part 4: Organization, security and commands for interchange 2005).

SW1 – SW2	Significato
9000	Operazione avvenuta con successo.
61XX	SW2 indica il numero di byte di risposta disponibili e prelevabili attraverso il comando *GetResponse()*.
62XX	Lo stato della memoria non volatile non è cambiato.
63XX	Lo stato della memoria non volatile è cambiato.
64XX	Lo stato della memoria non volatile non è cambiato.
65XX	Lo stato della memoria non volatile è cambiato.
66XX	Riservato per questioni riguardanti la sicurezza.
6700	Errore di lunghezza.
68XX	Funzioni in CLA non supportate.
69XX	Comando non permesso.
6AXX	Errore nei parametri P1-P2.
6B00	Errore nei parametri P1-P2.
6CXX	Errore nel parametro P3, SW2 indica la lunghezza esatta.
6D00	Codice d’istruzione non supportato o invalido.
6E00	Classe non supportata.
6F00	Nessuna diagnosi precisa.

Tabella 16 - Classificazione delle Status Word

4.2.1 Tipi definiti dal plugin “iso7816-4”

Il plugin *iso7816-4* non definisce alcun tipo aggiuntivo ma utilizza i tipi di dato definiti all’interno del plugin *pcsc*.

4.2.2 Il comando Trasmit0()

Il comando ***Transmit0()*** permette di inviare un comando utilizzando il protocollo standard T=0.

Il comando richiede i seguenti input:

- **CLA**: Classe dell'istruzione.
- **INS**: Codice dell'istruzione
- **P1**: Parametro 1 dell'istruzione.
- **P2**: Parametro 2 dell'istruzione.
- **P3**: Parametro 3 dell'istruzione.
- **Data**: Vettore di byte da inviare.

Il comando, oltre alle status word, restituisce un vettore di byte che rappresenta i dati risultato del comando stesso.

La Figura 57 mostra un esempio di utilizzo del comando *Transmit0()* per selezionare il master file in una scheda conforme allo standard.

```
scottsh
scottsh-1.0>
scottsh-1.0>Transmit0(CLA=0xC0, INS=0xA4, P1=0, P2=0, P3=2, data=0x3F00)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0x17 ( 23 )
scottsh-1.0>Transmit0(0xC0, 0xA4, 0, 0, 2, 0x3F00)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0x17 ( 23 )
scottsh-1.0>
```

Figura 57- Utilizzo del comando iso7816-4.Transmit0()

Per ottenere maggiori informazioni digitare "help iso7816-4.Transmit0".

4.2.3 Il comando Select()

Il comando ***Select()*** permette di selezionare un file o una directory con uno specifico identificatore. La Figura 58 mostra un esempio di utilizzo del comando per effettuare la selezione del master file.

```
scottsh
scottsh-1.0>
scottsh-1.0>var sbyte CLA = 0xC0
scottsh-1.0>
scottsh-1.0>Select(0x3F00)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0x17 ( 23 )
SString desc = "sw2 bytes of the response data"
scottsh-1.0>
```

Figura 58 - Esempio di uso del comando iso7816-4.Select()

Per ottenere maggiori informazioni digitare “help iso7816-4.Select”.

4.2.4 Il comando GetResponse()

Tramite il comando ***GetResponse()*** è possibile ottenere maggiori informazioni relative ai risultati del comando eseguito immediatamente prima. Ad esempio nel caso di una *Select()*, il comando *GetResponse()* permette di ottenere una stringa di byte che contiene delle informazioni dettagliate sul file selezionato come ad esempio dimensioni, access conditions, ecc. Il formato dei dati restituiti è generalmente proprietario e potrà essere compreso solamente consultando il manuale specifico della scheda. Il comando richiede in input il numero di byte d’informazione da prelevare. Questo numero è generalmente indicato nella status word 2 ottenuta dal comando precedente.

La Figura 59 mostra un esempio di utilizzo del comando *GetResponse()* dopo l’invocazione di *Select()* su un file riservato della scheda Cryptoflex 8K che contiene le informazioni relative al PIN (File ID = 0000).

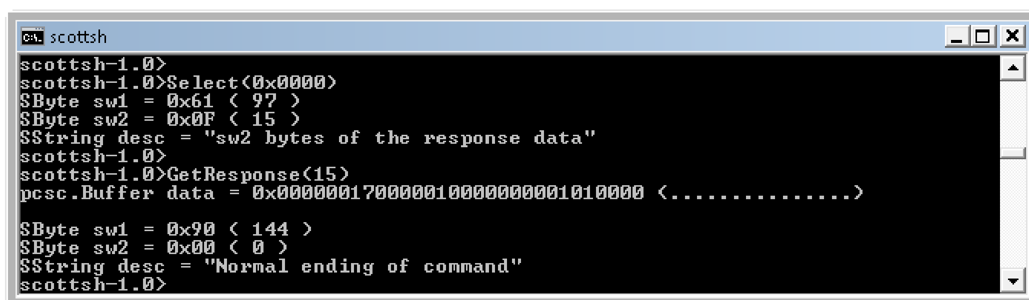


Figura 59 - Esempio di uso del comando iso7816-4.GetResponse()

Per ottenere maggiori informazioni digitare “help iso7816-4.GetResponse”.

4.2.5 Il comando Verify()

Il comando ***Verify()*** è utilizzato per effettuare l'autenticazione basata su PIN. Come input è necessario specificare il tipo di chiave, la lunghezza della chiave e la chiave stessa. Nel caso in cui è restituito il codice 0x9000 significa che l'operazione di autenticazione è andata a buon fine.

La Figura 60 mostra un esempio di utilizzo del comando *Verify()* per ottenere l'accesso alla scheda Cryptoflex 8K fornendo il PIN numero 1.

Figura 60 - Esempio di uso del comando iso7816-4.Verify()

Per ottenere maggiori informazioni digitare "help iso7816-4.Verify".

4.2.6 Il comando ReadBinary()

Il comando ***ReadBinary()*** permette di leggere una parte o tutto il contenuto di un transparent file dopo che questo è stato correttamente selezionato mediante il comando *Select()*. Il comando richiede la posizione iniziale da cui prelevare il contenuto (offset) e il numero di byte da leggere (length).

La Figura 61 mostra un esempio di lettura del contenuto del file con ID=0x0002 presente in una scheda Cryptoflex 8K.

```
scottsh
scottsh-1.0>
scottsh-1.0>Select(0x0002)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0x0F ( 15 )
SString desc = "sw2 bytes of the response data"
scottsh-1.0>
scottsh-1.0>ReadBinary(0, 8)
pcsc.Buffer data = 0x0000F06DFFFF0200 (...m....)

SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x00 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>
```

Figura 61 - Esempio di uso del comando iso7816-4.ReadBinary()

Per ottenere maggiori informazioni digitare “help iso7816-4.ReadBinary”.

4.2.7 Il comando UpdateBinary()

Il comando ***UpdateBinary()*** permette di scrivere una parte o tutto il contenuto di un transparent file dopo che questo è stato correttamente selezionato tramite il comando *Select()*. Il comando richiede la posizione iniziale da cui iniziare a scrivere (offset), il numero di byte da scrivere (length) e i dati stessi sotto forma di vettore di byte.

La Figura 62 mostra un esempio di scrittura di 10 byte su un file con ID=0x1111.

```
scottsh
scottsh-1.0>
scottsh-1.0>Select(0x1111)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0x0F ( 15 )
SString desc = "sw2 bytes of the response data"
scottsh-1.0>
scottsh-1.0>UpdateBinary(0, 10, 0x00112233445566778899)
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x00 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>
```

Figura 62 - Esempio di uso del comando iso7816-4.UpdateBinary()

Per ottenere maggiori informazioni digitare “help iso7816-4.UpdateBinary”.

4.2.8 Un esempio completo di utilizzo

La Figura 63 mostra un esempio di script che utilizza tutti i comandi definiti dal plugin "*iso7816-4*" e alcuni del plugin "*pcsc*" per svolgere delle operazioni sulla scheda Cryptoflex 8K.

In particolare questo script svolge nell'ordine le seguenti operazioni:

- Creazione del Resource Manager e connessione alla scheda Cryptoflex 8K
- Autenticazione tramite chiave
- Autenticazione tramite PIN
- Selezione del master file
- Lettura d'informazioni sul master file
- Creazione di un elementary file di 10 byte con ID=0x1234
- Scrittura nel file dei dati 0x00112233445566778899
- Lettura del contenuto del file per verificare la precedente scrittura
- Cancellazione del file appena creato
- Chiusura della connessione con la scheda

La Figura 64 mostra l'esecuzione dello script.

```
#Connessione alla scheda Cryptoflex 8K
load iso7816-4
EstablishContext(SCARD_SCOPE_USER)
ListReaders()
Connect(context, readers[0], SCARD_SHARE_EXCLUSIVE,
SCARD_PROTOCOL_T0)

#Autenticazione tramite chiave (Key 1)
Transmit(0xF02A0001082C15E526E93E8A19)

#Creazione della variabile CLA
var sbyte CLA = 0xC0

#Autenticazione tramite PIN (CHV 1)
Verify(1, 8, 0x1111111111111111)

#Selezione del master file
Select(0x3F00)

#Lettura di informazioni sul master file
GetResponse(sw2)

#Creazione di un elementary file di 10 byte con ID=1234
Transmit(0xF0E0000010FFFF000A1234010000000000103000000)

#Scrittura nel file
Select(0x1234)
UpdateBinary(0, 10, 0x00112233445566778899)

#Lettura del contenuto del file
ReadBinary(0, 10)

#Cancellazione del file
Select(0x3F00)
Transmit(0xF0E40000021234)

#Chiusura della connessione alla scheda
Disconnect( SCARD_LEAVE_CARD )
ReleaseContext()
```

Figura 63 - Riepilogo comandi iso7816-4

```
scottsh
*** Welcome in SCOTT Shell ***

Execution of commands in the Scott Configuration file...

scottsh-1.0>autovars true
scottsh-1.0>implicitvars true
scottsh-1.0>dumpvars true
scottsh-1.0>load iso7816-4
Autoload of 'pcsc'.
scottsh-1.0>EstablishContext(SCARD_SCOPE_USER)
pcsc.Error res = SCARD_S_SUCCESS ( 0 )
SNumber context = 0xCD010000 ( 3439394816 )
scottsh-1.0>ListReaders()
pcsc.Error res = SCARD_S_SUCCESS ( 0 )
pcsc.Vstring readers = { "Todos Argos Mini II USB 0", "Ugo's Smart Card Reader"
}
scottsh-1.0>Connect(context, readers[0], SCARD_SHARE_EXCLUSIVE, SCARD_PROTOCOL_T
0)
pcsc.Error res = SCARD_S_SUCCESS ( 0 )
SNumber connection = 0xEA010000 ( 3925934080 )
pcsc.Protocol protocol = SCARD_PROTOCOL_T0 ( 1 )
scottsh-1.0>Transmit(0xF02A0001082C15E526E93E8A19)
pcsc.Error res = SCARD_S_SUCCESS ( 0 )
pcsc.Buffer receiveBuffer = 0x9000 (..)

scottsh-1.0>var sbyte CLA = 0xC0
scottsh-1.0>Verify(1, 8, 0x1111111111111111)
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x00 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>Select(0x3F00)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0x17 ( 23 )
SString desc = "sw2 bytes of the response data"
scottsh-1.0>GetResponse(sw2)
pcsc.Buffer data = 0x00001B8C3F0038FF4F444401090000050400 8A8A8A8A00 (....?.8.ODD
............)

SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x00 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>Transmit(0xF0E0000010FFFF000A12340100000000000103000000)
pcsc.Error res = SCARD_S_SUCCESS ( 0 )
pcsc.Buffer receiveBuffer = 0x9000 (..)

scottsh-1.0>Select(0x1234)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0x0F ( 15 )
SString desc = "sw2 bytes of the response data"
scottsh-1.0>UpdateBinary(0, 10, 0x00112233445566778899)
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x00 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>ReadBinary(0, 10)
pcsc.Buffer data = 0x00112233445566778899 (.."3DUfw..)

SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x00 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>Select(0x3F00)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0x17 ( 23 )
SString desc = "sw2 bytes of the response data"
scottsh-1.0>Transmit(0xF0E40000021234)
pcsc.Error res = SCARD_S_SUCCESS ( 0 )
pcsc.Buffer receiveBuffer = 0x9000 (..)

scottsh-1.0>Disconnect( SCARD_LEAVE_CARD )
pcsc.Error res = SCARD_S_SUCCESS ( 0 )
scottsh-1.0>ReleaseContext()
pcsc.Error res = SCARD_S_SUCCESS ( 0 )
scottsh-1.0>
```

Figura 64 - Esecuzione del riepilogo comandi iso7816-4

4.3 Il plugin “cryptoflex8”

Il plugin “**cryptoflex8**” definisce un insieme di comandi per manipolare i file contenuti all’interno delle schede Cryptoflex 8K descritte in Appendice A.

Nell’ambito di questa tesi sono stati implementati tredici dei ventisette comandi disponibili sulla scheda, in particolare quelli relativi alla creazione e manipolazione dei transparent file e all’autenticazione tramite PIN. Inoltre sono stati creati altri comandi più astratti che semplificano notevolmente l’interazione con la scheda.

4.3.1 Tipi definiti dal plugin “cryptoflex8”

Il plugin *cryptoflex8* definisce i seguenti tipi enumeratori:

- **FileType**: specifica il tipo di file.
  - Transparent, LinearFixed, LinearVariable, Cyclic, Dedicated
- **FileStatus**: indica lo stato di un file.
  - Invalid, Active
- **AccessCondition**: specifica una condizione di accesso.
  - ALW, CHV1, CHV2, PRO, AUT
  - CHV1_PRO, CHV2_PRO, CHV1_AUT, CHV2_AUT
  - NEV
- **ChvData**: indica quali file CHV sono presenti.
  - NoCHV, CHV1, CHV1_CHV2
- **AvailablePINs**: un enumeratore che indica quanti PIN sono disponibili: I possibili valori sono:
  - NoCHV, CHV1_UnblockCHV1
  - CHV1_UnblockCHV1_CHV2_UnblockCHV2
- **CommandRestrictions**: un enumeratore che specifica quali comandi sono disponibili. I possibili valori sono:
  - NoDecreaseNoIncrease, DecreaseIncrese

- DecreaseOnly, IncreaseOnly

- **ChvStatus**: un enumeratore che specifica lo stato di un file CHV. I possibili valori sono:
  - Absent, Present

Per rappresentare in maniera aggregata le informazioni su file e directory, sono state definite le strutture dati *FileInfo*, *DirInfo* e *DirSelectInfo*.

La struttura ***FileInfo*** contiene tutte le informazioni relative ad un file (EF). La Tabella 17 mostra i membri di questa struttura.

Membro	Tipo	Descrizione
fileSize	SNumber	Dimensione del file.
fileID	SNumber	ID del file.
fileType	FileType	Tipo di file.
file Status	FileStatus	Stato del file.
restrictions	CommandRestrictions	Restrizioni presenti.
readBinary SeekAC	AccessCondition	AC per i comandi ReadBinary e Seek
readBinary SeekKeyNum	SByte	Numero di chiave se AC=AUT.
updateBinary DecreaseAC	AccessCondition	AC per i comandi UpdateBinary e Decrease.
updateBinary DecreaseKeyNum	SByte	Numero di chiave se AC=AUT
increaseAC	AccessCondition	AC per il comando Increase.
increaseKeyNum	SByte	Numero di chiave se AC=AUT.
createRecordAC	AccessCondition	AC per il comando CreateRecord.
create Record KeyNum	SByte	Numero di chiave se AC=AUT.
rehabilitateAC	AccessCondition	AC per il comando Rehabilitate.
rehabilitateKeyNum	SByte	Numero di chiave se AC=AUT.
invalidateAC	AccessCondition	AC per il comando Invalidate
InvalidateKeyNum	SByte	Numero di chiave se AC=AUT.
recordLength	SByte	Lunghezza di ogni record.
numRecords	SByte	Numero di record.

Tabella 17 - Membri della struttura FileInfo

La struttura ***DirInfo*** contiene tutte le informazioni relative a una directory (DF). La Tabella 18 mostra i membri di questa struttura.

Membro	Tipo	Descrizione
fileSize	SNumber	Dimensione del file.
fileID	SNumber	ID del file.
file Status	FileStatus	Stato del file.
dirNextAC	AccessCondition	AC per il comando DirNext
dirNextKeyNum	SByte	Numero di chiave se AC=AUT.
deleteAC	AccessCondition	AC per il comando Delete.
deleteKeyNum	SByte	Numero di chiave se AC=AUT
createAC	AccessCondition	AC per il comando Create.
createKeyNum	SByte	Numero di chiave se AC=AUT.
numDF	SByte	Numero di sottocartelle.
numEF	SByte	Numero di file.

Tabella 18 - Membri della struttura DirInfo

La struttura ***DirSelectInfo*** contiene un maggiore numero d'informazioni rispetto a *DirInfo*. Questa struttura è usata dal comando *SelectDF()*. La mostra i membri di questa struttura.

Membro	Tipo	Descrizione
remainedSpace	SNumber	Spazio disponibile rimasto.
fileID	SNumber	ID del file.
file Status	FileStatus	Stato del file.
dirNextAC	AccessCondition	AC per il comando DirNext
dirNextKeyNum	SByte	Numero di chiave se AC=AUT.
deleteAC	AccessCondition	AC per il comando Delete.
deleteKeyNum	SByte	Numero di chiave se AC=AUT
createAC	AccessCondition	AC per il comando Create.
createKeyNum	SByte	Numero di chiave se AC=AUT.
numDF	SByte	Numero di sottocartelle.
numEF	SByte	Numero di file.
chvData	ChvData	File CHV presenti.
availablePINs	AvailablePINs	PIN disponibili.

chv1Status	ChvStatus	Stato del file CHV1.
remainedCHV1	SByte	Numero di tentativi rimasti per autenticarsi con il PIN1.
remainedUnblockCHV1	SByte	Numero di tentativi rimasti per sbloccare il PIN1.
chv2Status	ChvStatus	Stato del file CHV2.
remainedCHV2	SByte	Numero di tentativi rimasti per autenticarsi con il PIN2.
remainedUnblockCHV2	SByte	Numero di tentativi rimasti per sbloccare il PIN2.

Tabella 19 - Membri della struttura DirSelectInfo

Infine i tipi ***VFileInfo*** e ***VDirInfo*** sono dei vettori i cui elementi sono rispettivamente le strutture *FileInfo* e *DirInfo*. Questi due tipi sono utilizzati dal comando *GetFileInfo()*.

4.3.2 Il comando ChangeCHV()

Il comando ***ChangeCHV()*** permette di cambiare il valore di uno dei due PIN presenti sulla scheda. E' sufficiente specificare quale PIN si desidera cambiare (chvType), il valore del vecchio PIN (oldCHV) e il valore del nuovo PIN (newCHV).

La Figura 65 mostra un esempio di utilizzo del comando per cambiare il PIN 1 dal valore 0x1111111111111111 al valore 0x2222222222222222.

```
scottsh
scottsh-1.0>
scottsh-1.0>ChangeCHV(1, 0x1111111111111111, 0x2222222222222222)
SByte sw1  = 0x90 ( 144 )
SByte sw2  = 0x00 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>
```

Figura 65 - Comando cryptoflex8.ChangeCHV()

Per ottenere maggiori informazioni digitare “help cryptoflex8.ChangeCHV”.

4.3.3 Il comando CreateFile()

Il comando ***CreateFile()*** permette di creare un file o una directory. I parametri di input sono il valore di inizializzazione (*init*), il numero di record (*numRecords*) o zero, la lunghezza della struttura dati del file (*length*) e un vettore di byte codificato secondo quanto riportato nel manuale della scheda (data).

La Figura 66 mostra un esempio di utilizzo per creare un file Transparent con ID=0x1111 sotto il master file.

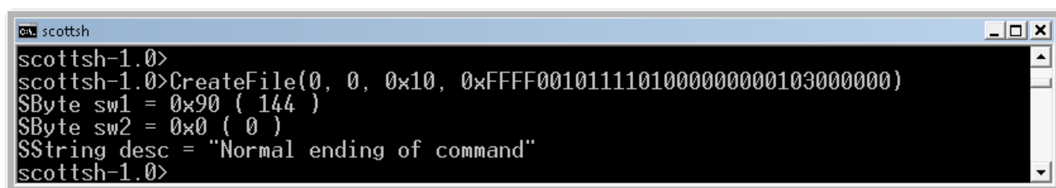


Figura 66 - Comando cryptoflex8.CreateFile()

Invece di usare il comando *CreateFile()* è opportuno utilizzare i comandi *CreateEF()* e *CreateDF()* che permettono di svolgere lo stesso compito senza conoscere complessi dettagli di basso livello sulla rappresentazione dei dati.

Per ottenere maggiori informazioni digitare “help cryptoflex8.CreateFile”.

4.3.4 Il comando CreateDF()

Il comando ***CreateDF()*** permette di creare una directory sotto quella correntemente selezionata. Tutte le informazioni sulla nuova directory sono specificate attraverso un parametro di tipo *DirInfo*.

La Figura 67 mostra un esempio che crea una directory di 100 byte con ID=0x2000 rendendo accessibili le operazioni solamente attraverso l’autenticazione tramite PIN1.

Figura 67 - Comando cryptoflex8.CreateDF()

Per ottenere maggiori informazioni digitare “help cryptoflex8.CreateDF”.

4.3.5 Il comando CreateEF()

Il comando ***CreateEF()*** permette di creare un elementary file (EF) sotto la directory correntemente selezionata. E’ necessario specificare un valore d’inizializzazione (*init*) e una variabile di tipo *FileInfo* con tutte le informazioni relative al nuovo file da creare.

La Figura 68 mostra un esempio di utilizzo del comando CreateEF() per creare un transparent file di 10 byte con ID=0x1111 e nessuna restrizione sulle operazioni che possono essere eseguite.

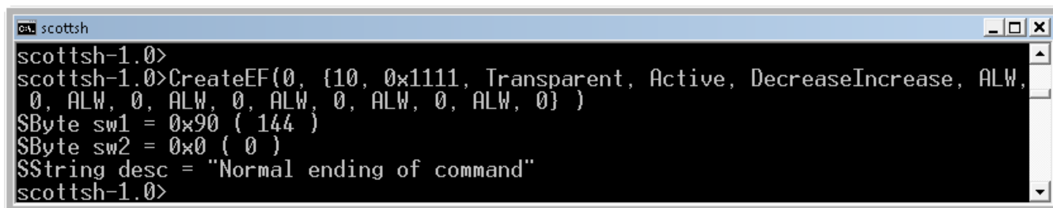


Figura 68 - Comando cryptoflex8.CreateEF()

Per ottenere maggiori informazioni digitare “help cryptoflex8.CreateEF”.

4.3.6 Il comando DeleteFile()

Il comando ***DeleteFile()*** permette di eliminare un file o una directory. Prima di eseguire il comando deve essere selezionata la directory padre e devono essere soddisfatte le condizioni di accesso. Per evitare problemi di frammentazione, i file devono essere cancellati in ordine inverso a quello di creazione. L'unico parametro del comando è l'identificatore del file che si desidera rimuovere.

La Figura 69 mostra un esempio di utilizzo del comando *DeleteFile()* per rimuovere un file, con ID=0x1234, presente sotto il master file.

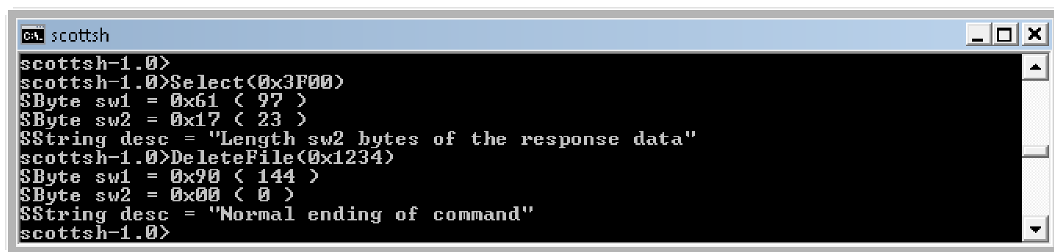


Figura 69 - Comando cryptoflex8.DeleteFile()

Per ottenere maggiori informazioni digitare "help cryptoflex8.DeleteFile".

4.3.7 Il comando DirNext()

Il comando ***DirNext()*** permette di ottenere informazioni sui file (EF o DF) presenti all'interno della directory correntemente selezionata. Ogni esecuzione del comando restituisce le informazioni su un singolo file. E' necessario quindi eseguirlo più volte fino a quando sarà restituito un opportuno messaggio che segnala che tutti i file sono stati considerati.

Due sono i parametri di output: *fileInfo* e *dirInfo* rispettivamente di tipo *FileInfo* e *DirInfo*. Solamente uno di questi due parametri sarà valorizzato a seconda che il file correntemente in analisi sia un EF o un DF.

Il testo immediatamente seguente, tratto da una sessione della shell, mostra un esempio di utilizzo del comando *DirNext()* per ottenere le informazioni su tutti i file presenti sotto il master file.

```
scottsh-1.0>
scottsh-1.0>Select(0x3F00)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0x17 ( 23 )
SString desc = "sw2 bytes of the response data"
scottsh-1.0>
scottsh-1.0>DirNext()
cryptoflex8.FileInfo fileInfo =
{
        fileSize = 0x8038 ( 32824 ),
        fileID = 0x11 ( 17 ),
        fileType = Transparent ( 0x1 ),
        fileStatus = Active ( 0x1 ),
        readBinarySeekAC = NEV ( 0xF ),
        readBinarySeekKeyNum = 0x1 ( 1 ),
        updateBinaryDecreaseAC = AUT ( 0x4 ),
        updateBinaryDecreaseKeyNum = 0x1 ( 1 ),
        increaseAC = NEV ( 0xF ),
        increaseKeyNum = 0x1 ( 1 ),
        createRecordAC = NEV ( 0xF ),
        createRecordKeyNum = 0x1 ( 1 ),
        rehabilitateAC = AUT ( 0x4 ),
        rehabilitateKeyNum = 0x1 ( 1 ),
        invalidateAC = AUT ( 0x4 ),
        invalidateKeyNum = 0x1 ( 1 )
}
cryptoflex8.DirInfo dirInfo =
{

}
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>DirNext()
cryptoflex8.FileInfo fileInfo =
{
        fileSize = 0x18 ( 24 ),
        fileID = 0x2 ( 2 ),
        fileType = Transparent ( 0x1 ),
        fileStatus = Active ( 0x1 ),
        readBinarySeekAC = ALW ( 0x0 ),
        readBinarySeekKeyNum = 0x0 ( 0 ),
        updateBinaryDecreaseAC = AUT ( 0x4 ),
        updateBinaryDecreaseKeyNum = 0x0 ( 0 ),
        increaseAC = NEV ( 0xF ),
        increaseKeyNum = 0x0 ( 0 ),
        createRecordAC = NEV ( 0xF ),
        createRecordKeyNum = 0x0 ( 0 ),
        rehabilitateAC = NEV ( 0xF ),
        rehabilitateKeyNum = 0x0 ( 0 ),
        invalidateAC = NEV ( 0xF ),
        invalidateKeyNum = 0x0 ( 0 )
}
cryptoflex8.DirInfo dirInfo =
{

}
```

```
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>DirNext()
cryptoflex8.FileInfo fileInfo =
{
        fileSize = 0xC028 ( 49192 ),
        fileID = 0x0 ( 0 ),
        fileType = Transparent ( 0x1 ),
        fileStatus = Active ( 0x1 ),
        readBinarySeekAC = ALW ( 0x0 ),
        readBinarySeekKeyNum = 0x0 ( 0 ),
        updateBinaryDecreaseAC = ALW ( 0x0 ),
        updateBinaryDecreaseKeyNum = 0x0 ( 0 ),
        increaseAC = ALW ( 0x0 ),
        increaseKeyNum = 0x0 ( 0 ),
        createRecordAC = ALW ( 0x0 ),
        createRecordKeyNum = 0x0 ( 0 ),
        rehabilitateAC = ALW ( 0x0 ),
        rehabilitateKeyNum = 0x0 ( 0 ),
        invalidateAC = ALW ( 0x0 ),
        invalidateKeyNum = 0x0 ( 0 )
}
cryptoflex8.DirInfo dirInfo =
{

}
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>DirNext()
cryptoflex8.FileInfo fileInfo =
{
        fileSize = 0xC028 ( 49192 ),
        fileID = 0x100 ( 256 ),
        fileType = Transparent ( 0x1 ),
        fileStatus = Active ( 0x1 ),
        readBinarySeekAC = ALW ( 0x0 ),
        readBinarySeekKeyNum = 0x0 ( 0 ),
        updateBinaryDecreaseAC = ALW ( 0x0 ),
        updateBinaryDecreaseKeyNum = 0x0 ( 0 ),
        increaseAC = ALW ( 0x0 ),
        increaseKeyNum = 0x0 ( 0 ),
        createRecordAC = ALW ( 0x0 ),
        createRecordKeyNum = 0x0 ( 0 ),
        rehabilitateAC = ALW ( 0x0 ),
        rehabilitateKeyNum = 0x0 ( 0 ),
        invalidateAC = ALW ( 0x0 ),
        invalidateKeyNum = 0x0 ( 0 )
}
cryptoflex8.DirInfo dirInfo =
{

}
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>DirNext()
cryptoflex8.FileInfo fileInfo =
{

}
```

```
cryptoflex8.DirInfo dirInfo =
{
        fileSize = 0x7C ( 124 ),
        fileID = 0x2000 ( 8192 ),
        fileStatus = Active ( 0x1 ),
        dirNextAC = ALW ( 0x0 ),
        dirNextKeyNum = 0x0 ( 0 ),
        deleteAC = ALW ( 0x0 ),
        deleteKeyNum = 0x0 ( 0 ),
        createAC = ALW ( 0x0 ),
        createKeyNum = 0x0 ( 0 ),
        numDF = 0x0 ( 0 ),
        numEF = 0x1 ( 1 )
}
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>DirNext()
cryptoflex8.FileInfo fileInfo =
{

}
cryptoflex8.DirInfo dirInfo =
{

}
SByte sw1 = 0x6A ( 106 )
SByte sw2 = 0x82 ( 130 )
SString desc = "File ID not found or no more files in current DF"
scottsh-1.0>
```

Per ottenere maggiori informazioni digitare “help cryptoflex8.DirNext”.

4.3.8 Il comando GetFilesInfo()

Il comando ***GetFilesInfo()*** consente di ottenere, in un colpo solo, tutte le informazioni relative ai file (EF e DF) presenti sotto la directory correntemente selezionata. L’implementazione di questo comando strutta l’esecuzione ripetuta di *DirNext()* per costruire un vettore di *FileInfo* (*filesInfo*) e un vettore di *DirInfo* (*dirsInfo*).

Il testo seguente, tratto da una sessione della shell Scott, mostra un esempio di utilizzo di quest’utile comando per ottenere tutte le informazioni su file e directory presenti sotto il master file.

```
scottsh-1.0>
scottsh-1.0>dumpvars false
scottsh-1.0>Select(0x3F00)
scottsh-1.0>GetFilesInfo()
scottsh-1.0>show dirsInfo

Type: cryptoflex8.VDirInfo
Value:
{

{
        fileSize = 0x7C ( 124 ),
        fileID = 0x2000 ( 8192 ),
        fileStatus = Active ( 0x1 ),
        dirNextAC = ALW ( 0x0 ),
        dirNextKeyNum = 0x0 ( 0 ),
        deleteAC = ALW ( 0x0 ),
        deleteKeyNum = 0x0 ( 0 ),
        createAC = ALW ( 0x0 ),
        createKeyNum = 0x0 ( 0 ),
        numDF = 0x0 ( 0 ),
        numEF = 0x1 ( 1 )
}
}

scottsh-1.0>show filesInfo

Type: cryptoflex8.VFileInfo
Value:
{

{
        fileSize = 0x8038 ( 32824 ),
        fileID = 0x11 ( 17 ),
        fileType = Transparent ( 0x1 ),
        fileStatus = Active ( 0x1 ),
        readBinarySeekAC = NEV ( 0xF ),
        readBinarySeekKeyNum = 0x1 ( 1 ),
        updateBinaryDecreaseAC = AUT ( 0x4 ),
        updateBinaryDecreaseKeyNum = 0x1 ( 1 ),
        increaseAC = NEV ( 0xF ),
        increaseKeyNum = 0x1 ( 1 ),
        createRecordAC = NEV ( 0xF ),
        createRecordKeyNum = 0x1 ( 1 ),
        rehabilitateAC = AUT ( 0x4 ),
        rehabilitateKeyNum = 0x1 ( 1 ),
        invalidateAC = AUT ( 0x4 ),
        invalidateKeyNum = 0x1 ( 1 )
},

{
        fileSize = 0x18 ( 24 ),
        fileID = 0x2 ( 2 ),
        fileType = Transparent ( 0x1 ),
        fileStatus = Active ( 0x1 ),
        readBinarySeekAC = ALW ( 0x0 ),
        readBinarySeekKeyNum = 0x0 ( 0 ),
        updateBinaryDecreaseAC = AUT ( 0x4 ),
        updateBinaryDecreaseKeyNum = 0x0 ( 0 ),
        increaseAC = NEV ( 0xF ),
        increaseKeyNum = 0x0 ( 0 ),
        createRecordAC = NEV ( 0xF ),
        createRecordKeyNum = 0x0 ( 0 ),
        rehabilitateAC = NEV ( 0xF ),
```

```
        rehabilitateKeyNum = 0x0 ( 0 ),
        invalidateAC = NEV ( 0xF ),
        invalidateKeyNum = 0x0 ( 0 )
},

{
        fileSize = 0xC028 ( 49192 ),
        fileID = 0x0 ( 0 ),
        fileType = Transparent ( 0x1 ),
        fileStatus = Active ( 0x1 ),
        readBinarySeekAC = ALW ( 0x0 ),
        readBinarySeekKeyNum = 0x0 ( 0 ),
        updateBinaryDecreaseAC = ALW ( 0x0 ),
        updateBinaryDecreaseKeyNum = 0x0 ( 0 ),
        increaseAC = ALW ( 0x0 ),
        increaseKeyNum = 0x0 ( 0 ),
        createRecordAC = ALW ( 0x0 ),
        createRecordKeyNum = 0x0 ( 0 ),
        rehabilitateAC = ALW ( 0x0 ),
        rehabilitateKeyNum = 0x0 ( 0 ),
        invalidateAC = ALW ( 0x0 ),
        invalidateKeyNum = 0x0 ( 0 )
},

{
        fileSize = 0xC028 ( 49192 ),
        fileID = 0x100 ( 256 ),
        fileType = Transparent ( 0x1 ),
        fileStatus = Active ( 0x1 ),
        readBinarySeekAC = ALW ( 0x0 ),
        readBinarySeekKeyNum = 0x0 ( 0 ),
        updateBinaryDecreaseAC = ALW ( 0x0 ),
        updateBinaryDecreaseKeyNum = 0x0 ( 0 ),
        increaseAC = ALW ( 0x0 ),
        increaseKeyNum = 0x0 ( 0 ),
        createRecordAC = ALW ( 0x0 ),
        createRecordKeyNum = 0x0 ( 0 ),
        rehabilitateAC = ALW ( 0x0 ),
        rehabilitateKeyNum = 0x0 ( 0 ),
        invalidateAC = ALW ( 0x0 ),
        invalidateKeyNum = 0x0 ( 0 )
}
}
scottsh-1.0>
```

Per ottenere maggiori informazioni digitare “help cryptoflex8.GetFilesInfo”.

4.3.9 Il comando GetResponse()

Il comando ***GetResponse()*** è sostanzialmente equivalente a quello omonimo implementato nel plugin *“iso7816-4”* e spiegato nel paragrafo 4.2.4. L’unica

differenza è l’assenza del parametro CLA in quanto è automaticamente specificato dall’implementazione interna del plugin.

Per ottenere maggiori informazioni digitare “help cryptoflex8.GetResponse”.

4.3.10 Il comando Invalidate()

Il comando ***Invalidate()*** permette di disattivare il file correntemente selezionato. Un file reso invalido non sarà più accessibile se non attraverso i comandi *Select()*, *DeleteFile()* e *Rehabilitate()*.

La Figura 70 mostra l’utilizzo del comando *Invalidate()* sul file con ID=0x1234.

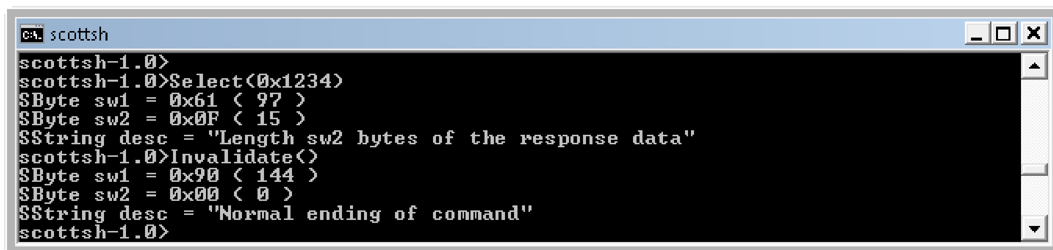


Figura 70 - Comando cryptoflex8.Invalidate()

Per ottenere maggiori informazioni digitare “help cryptoflex8.Invalidate”.

4.3.11 Il comando ReadBinary()

Il comando ***ReadBinary()*** è sostanzialmente equivalente a quello omonimo implementato nel plugin “*iso7816-4*” e spiegato nel paragrafo 4.2.6. L’unica differenza è l’assenza del parametro CLA in quanto è automaticamente specificato dall’implementazione interna del plugin.

Per ottenere maggiori informazioni digitare “help cryptoflex8.ReadBinary”.

4.3.12 Il comando Rehabilitate()

Il comando ***Rehabilitate()*** permette di riattivare un file che è stato disattivato con il comando *Invalidate()*.

La Figura 71 mostra l'utilizzo del comando *Rehabilitate()* sul file con ID=0x1234.

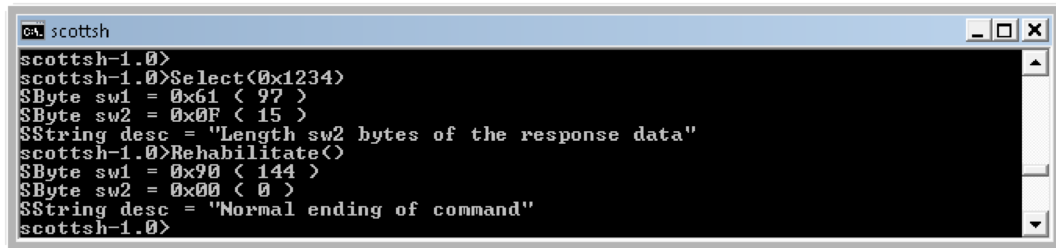


Figura 71 - Comando Rehabilitate()

Per ottenere maggiori informazioni digitare "help cryptoflex8.Rehabilitate".

4.3.13 Il comando RetrieveCLA()

Il comando ***RetrieveCLA()*** permette di ottenere il valore del byte CLA utilizzato dalla scheda Cryptoflex8 nei comandi che aderiscono allo standard Iso7816-4. In combinazione con il meccanismo delle variabili automatiche una chiamata a questo comando comporta la creazione della variabile CLA che attraverso il meccanismo delle variabili implicite sarà automaticamente passata a tutti i comandi implementati dal plugin "*iso7816-4*".

La Figura 72 mostra l'utilizzo del comando *RetrieveCLA()* seguito da una chiamata al comando standard *Select()*.

Per ottenere maggiori informazioni digitare "help cryptoflex8.RetrieveCLA".

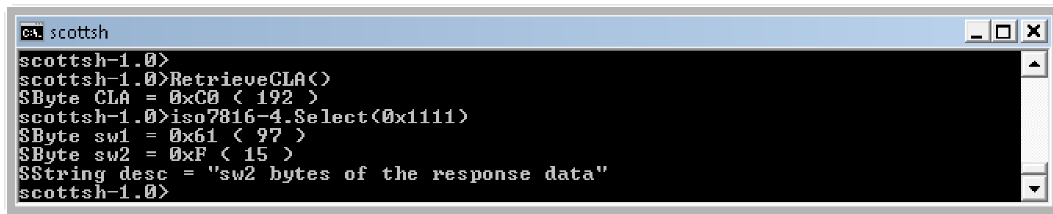


Figura 72 - Comando cryptoflex8.RetrieveCLA()

4.3.14 Il comando Select()

Il comando ***Select()*** è sostanzialmente equivalente a quello omonimo implementato nel plugin *"iso7816-4"* e spiegato nel paragrafo 4.2.3. L'unica differenza è l'assenza del parametro CLA in quanto è automaticamente specificato dall'implementazione interna del plugin.

Per ottenere maggiori informazioni digitare "help cryptoflex8.Select".

4.3.15 Il comando SelectDF()

Il comando ***SelectDF()*** permette di selezionare una directory (DF) ed ottenere molte informazioni su di essa attraverso una variabile di tipo *DirSelectInfo*.

La Figura 73 mostra un esempio di utilizzo del comando *SelectDF()* per ottenere informazioni sul master file.

Per ottenere maggiori informazioni digitare "help cryptoflex8.SelectDF".

```
scottsh
scottsh-1.0>
scottsh-1.0>SelectDF(0x3F00)
cryptoflex8.DirSelectInfo dirSelectInfo =
{
        remainedSpace = 0x1B90 ( 7056 ),
        fileID = 0x3F00 ( 16128 ),
        fileStatus = Active ( 0x1 ),
        dirNextAC = AUT ( 0x4 ),
        deleteAC = AUT ( 0x4 ),
        createAC = AUT ( 0x4 ),
        numDF = 0x0 ( 0 ),
        numEF = 0x5 ( 5 ),
        chvData = CHV1_CHV2 ( 0x9 ),
        availablePINs = CHV1_UnblockCHV1_CHV2_UnblockCHV2 ( 0x4 ),
        chv1Status = Present ( 0x8 ),
        remainedCHV1 = 0xA ( 10 ),
        remainedUnblockCHV1 = 0xA ( 10 ),
        chv2Status = Present ( 0x8 ),
        remainedCHV2 = 0xA ( 10 ),
        remainedUnblockCHV2 = 0xA ( 10 )
}
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>_
```

Figura 73 - Comando cryptoflex8.SelectDF()

4.3.16 Il comando SelectEF()

Il comando ***SelectEF()*** permette di selezionare un elementary file (EF) ed ottenere informazioni su di esso attraverso una variabile di tipo *FileInfo*.

La Figura 74 mostra un esempio di utilizzo del comando *SelectEF()* per ottenere informazioni sul file con ID=0x1111.

```
scottsh
scottsh-1.0>
scottsh-1.0>SelectEF(0x1111)
cryptoflex8.FileInfo fileInfo =
{
        fileSize = 0xA ( 10 ),
        fileID = 0x1111 ( 4369 ),
        fileType = Transparent ( 0x1 ),
        fileStatus = Active ( 0x1 ),
        restrictions = NoDecreaseNoIncrease ( 0x0 ),
        readBinarySeekAC = ALW ( 0x0 ),
        updateBinaryDecreaseAC = ALW ( 0x0 ),
        increaseAC = ALW ( 0x0 ),
        createRecordAC = ALW ( 0x0 ),
        rehabilitateAC = ALW ( 0x0 ),
        invalidateAC = ALW ( 0x0 )
}
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>
```

Figura 74 - Comando cryptoflex8.SelectEF()

Per ottenere maggiori informazioni digitare “help cryptoflex8.SelectEF”.

4.3.17 Il comando UnblockCHV()

Il comando ***UnblockCHV()*** permette di sbloccare e resettare un PIN bloccato. E' sufficiente specificare quale PIN si desidera sbloccare (chvType), il PIN di sblocco (unblockCHV) e il nuovo valore del PIN (newCHV).

La Figura 75 mostra un esempio di utilizzo del comando. Prima è mostrato un utilizzo del comando *VerifyCHV()* per evidenziare che non sono più disponibili tentativi e che quindi l'autenticazione tramite PIN2 è bloccata, poi viene sbloccato il PIN2 e resettato al valore 0x3333333333333333. Infine si effettua ancora l'autenticazione ma con il nuovo PIN per dimostrare che l'operazione è andata a buon fine.

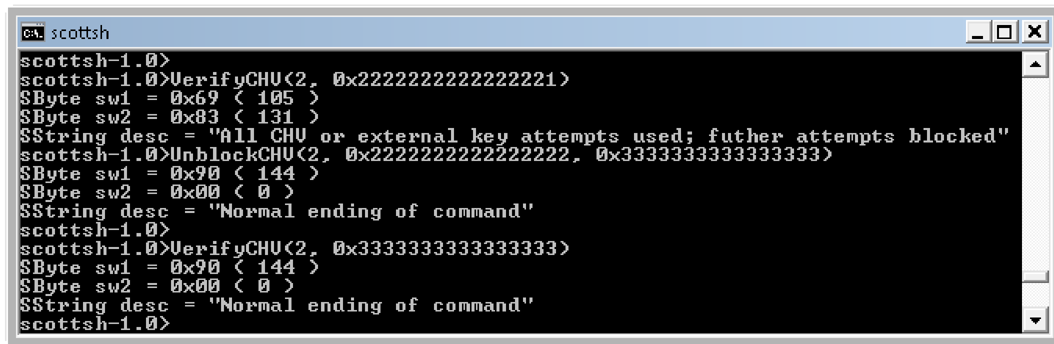


Figura 75 - Comando cryptoflex8.UnblockCHV()

Per ottenere maggiori informazioni digitare "help cryptoflex8.UnblockCHV".

4.3.18 Il comando UpdateBinary()

Il comando ***UpdateBinary()*** è sostanzialmente equivalente a quello omonimo implementato nel plugin "*iso7816-4*" e spiegato nel paragrafo 4.2.7. L'unica differenza è l'assenza del parametro CLA in quanto è automaticamente specificato dall'implementazione interna del plugin.

Per ottenere maggiori informazioni digitare "help cryptoflex8.UpdateBinary".

4.3.19 Il comando VerifyCHV()

Il comando ***VerifyCHV()*** è sostanzialmente equivalente al comando *Verify()* implementato nel plugin "*iso7816-4*" e spiegato nel paragrafo 4.2.5. Il parametro CLA non è presente in quanto è automaticamente specificato dall'implementazione interna del plugin. Inoltre non è presente nemmeno il parametro *length* perché la lunghezza del PIN è sempre di 8 byte.

Per ottenere maggiori informazioni digitare "help cryptoflex8.VerifyCHV".

4.3.20 Il comando VerifyKey()

Il comando ***VerifyKey()*** permette di verificare una delle chiavi che si trova nel file riservato "EF External Key" in modo da ottenere l'autenticazione AUT necessaria per eseguire con successo la maggior parte dei comandi offerti del plugin. E' necessario specificare il numero della chiave che si vuole verificare (chvType) e la chiave stessa (chvKey).

La Figura 76 mostra come ottenere l'autenticazione AUT utilizzando il comando *VerifyKey()*.

Figura 76 - Comando cryptoflex8.VerifyKey()

Per ottenere maggiori informazioni digitare "help cryptoflex8.VerifyKey".

4.3.21 Un esempio completo di utilizzo

In questo paragrafo è mostrato un esempio di script che utilizza i principali comandi definiti dal plugin "*cryptoflex8*" e alcuni comandi del plugin "pcsc" per svolgere delle operazioni sul file system della scheda Cryptoflex 8K.

In particolare questo script svolge nell'ordine le seguenti operazioni:

- Creazione del Resource Manager e connessione alla scheda Cryptoflex 8K
- Autenticazione tramite chiave (Key 1)
- Selezione e visualizzazione d'informazioni sul Master File
- Creazione di un file con ID=0x1234 sotto il Master File
- Creazione di una directory con ID=0x2000 protetta da PIN1
- Autenticazione tramite PIN1
- Creazione del file con ID=0x2001 sotto la nuova directory 0x2000
- Creazione del file con ID=0x2002 sotto la nuova directory 0x2000
- Lettura d'informazioni sul file con ID=0x2002
- Scrittura e lettura nel file con ID=0x1234
- Cancellazione dei file e delle directory create per ripristinare lo stato iniziale della scheda.

Di seguito è riportato integralmente il testo dello script:

```
# Creazione del Resource Manager e connessione alla scheda Cryptoflex 8K
load cryptoflex8
EstablishContext(SCARD_SCOPE_USER)
ListReaders()
Connect(context, readers[0], SCARD_SHARE_EXCLUSIVE, SCARD_PROTOCOL_T0)

# Autenticazione tramite chiave (Key 1)
VerifyKey(1, 0x2C15E526E93E8A19)

RetrieveCLA()

SelectDF(0x3F00)

# Creazione del file con ID=0x1234
CreateEF(0, {10, 0x1234, Transparent, Active, NoDecreaseNoIncrease, ALW, 0, ALW,
0, ALW, 0, ALW, 0, ALW, 0, ALW, 0} )

# Creazione della directory con ID=0x2000
Select(0x3F00)
CreateDF( {100, 0x2000, Active, CHV1, 0, CHV1, 0, CHV1, 0} )

# Autenticazione tramite PIN1 (necessaria per creare i file)
```

```
VerifyCHV(1, 0x1111111111111111)

# Creazione del file con ID=0x2001
Select(0x2000)
CreateEF(0, {20, 0x2001, Transparent, Active, NoDecreaseNoIncrease, ALW, 0, ALW,
0, ALW, 0, ALW, 0, ALW, 0, ALW, 0} )

# Creazione del file con ID=0x2002
Select(0x2000)
CreateEF(0, {20, 0x2002, Transparent, Active, NoDecreaseNoIncrease, ALW, 0, ALW,
0, ALW, 0, ALW, 0, ALW, 0, ALW, 0} )

# Lettura informazioni sul file con ID=0x2002
SelectEF(0x2002)

# Scrittura e lettura nel file con ID=0x1234
Select(0x3F00)
Select(0x1234)
UpdateBinary(0, 10, 0x41424344454647484950)
ReadBinary(0, 10)

# Si riporta la scheda nella configurazione iniziale
Select(0x2000)
DeleteFile(0x2002)
DeleteFile(0x2001)
Select(0x3F00)
DeleteFile(0x2000)
DeleteFile(0x1234)
```

Di seguito è mostrato il risultato dell'esecuzione dello script all'interno della shell.

```
*** Welcome in SCOTT Shell ***

Execution of commands in the Scott Configuration file...

scottsh-1.0>autovars true
scottsh-1.0>implicitvars true
scottsh-1.0>dumpvars true
scottsh-1.0>load cryptoflex8
Autoload of 'pcsc'.
Autoload of 'iso7816-4'.
scottsh-1.0>EstablishContext(SCARD_SCOPE_USER)
pcsc.Error res = SCARD_S_SUCCESS ( 0x0 )
SNumber context = 0xCD010000 ( 3439394816 )
scottsh-1.0>ListReaders()
pcsc.Error res = SCARD_S_SUCCESS ( 0x0 )
pcsc.Vstring readers =
{
"Todos Argos Mini II USB 1",
"Ugo's Smart Card Reader"
}
scottsh-1.0>Connect(context, readers[0], SCARD_SHARE_EXCLUSIVE, SCARD_PROTOCOL_T
0)
pcsc.Error res = SCARD_S_SUCCESS ( 0x0 )
SNumber connection = 0xEA010000 ( 3925934080 )
pcsc.Protocol protocol = SCARD_PROTOCOL_T0 ( 0x1 )
scottsh-1.0>VerifyKey(1, 0x2C15E526E93E8A19)
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
```

```
scottsh-1.0>RetrieveCLA()
SByte CLA = 0xC0 ( 192 )
scottsh-1.0>SelectDF(0x3F00)
cryptoflex8.DirSelectInfo dirSelectInfo =
{
        remainedSpace = 0x1BAC ( 7084 ),
        fileID = 0x3F00 ( 16128 ),
        fileStatus = Active ( 0x1 ),
        dirNextAC = AUT ( 0x4 ),
        deleteAC = AUT ( 0x4 ),
        createAC = AUT ( 0x4 ),
        numDF = 0x0 ( 0 ),
        numEF = 0x4 ( 4 ),
        chvData = CHV1_CHV2 ( 0x9 ),
        availablePINs = CHV1_UnblockCHV1_CHV2_UnblockCHV2 ( 0x4 ),
        chv1Status = Present ( 0x8 ),
        remainedCHV1 = 0xA ( 10 ),
        remainedUnblockCHV1 = 0xA ( 10 ),
        chv2Status = Present ( 0x8 ),
        remainedCHV2 = 0xA ( 10 ),
        remainedUnblockCHV2 = 0xA ( 10 )
}
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>CreateEF(0, {10, 0x1234, Transparent, Active, NoDecreaseNoIncrease,
ALW, 0, ALW, 0, ALW, 0, ALW, 0, ALW, 0, ALW, 0} )
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>Select(0x3F00)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0x17 ( 23 )
SString desc = "sw2 bytes of the response data"
scottsh-1.0>CreateDF( {100, 0x2000, Active, CHV1, 0, CHV1, 0, CHV1, 0} )
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>VerifyCHV(1, 0x1111111111111111)
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>Select(0x2000)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0x17 ( 23 )
SString desc = "sw2 bytes of the response data"
scottsh-1.0>CreateEF(0, {20, 0x2001, Transparent, Active, NoDecreaseNoIncrease,
ALW, 0, ALW, 0, ALW, 0, ALW, 0, ALW, 0, ALW, 0} )
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>Select(0x2000)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0x17 ( 23 )
SString desc = "sw2 bytes of the response data"
scottsh-1.0>CreateEF(0, {20, 0x2002, Transparent, Active, NoDecreaseNoIncrease,
ALW, 0, ALW, 0, ALW, 0, ALW, 0, ALW, 0, ALW, 0} )
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>SelectEF(0x2002)
cryptoflex8.FileInfo fileInfo =
{
        fileSize = 0x14 ( 20 ),
```

```
        fileID = 0x2002 ( 8194 ),
        fileType = Transparent ( 0x1 ),
        fileStatus = Active ( 0x1 ),
        restrictions = NoDecreaseNoIncrease ( 0x0 ),
        readBinarySeekAC = ALW ( 0x0 ),
        updateBinaryDecreaseAC = ALW ( 0x0 ),
        increaseAC = ALW ( 0x0 ),
        createRecordAC = ALW ( 0x0 ),
        rehabilitateAC = ALW ( 0x0 ),
        invalidateAC = ALW ( 0x0 )
}
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>Select(0x3F00)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0x17 ( 23 )
SString desc = "sw2 bytes of the response data"
scottsh-1.0>Select(0x1234)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0xF ( 15 )
SString desc = "sw2 bytes of the response data"
scottsh-1.0>UpdateBinary(0, 10, 0x41424344454647484950)
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>ReadBinary(0, 10)
pcsc.Buffer data = 0x41424344454647484950 (ABCDEFGHIP)
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>Select(0x2000)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0x17 ( 23 )
SString desc = "sw2 bytes of the response data"
scottsh-1.0>DeleteFile(0x2002)
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>DeleteFile(0x2001)
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>Select(0x3F00)
SByte sw1 = 0x61 ( 97 )
SByte sw2 = 0x17 ( 23 )
SString desc = "sw2 bytes of the response data"
scottsh-1.0>DeleteFile(0x2000)
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>DeleteFile(0x1234)
SByte sw1 = 0x90 ( 144 )
SByte sw2 = 0x0 ( 0 )
SString desc = "Normal ending of command"
scottsh-1.0>
```

5. Progettazione ed implementazione di SCOTT

5.1 Metodologia di sviluppo e scelte tecnologiche

SCOTT è stato realizzato tenendo in considerazione i requisiti di progettazione presentati nel paragrafo 2.2. Questi requisiti di alto livello sono stati progressivamente dettagliati attraverso la realizzazione di un semplice documento di testo. Utilizzando una strategia bottom-up, in seguito, sono stati realizzati diversi diagrammi delle classi utilizzando la notazione UML. Il processo di creazione del documento di specifica e dei vari diagrammi UML ha consentito di ottenere un visione abbastanza chiara del progetto in modo tale da passare direttamente alla fase successiva d'implementazione.

Sostanzialmente quindi, la metodologia di sviluppo software seguita è quella classica del "modello a cascata" che prevede nell'ordine l'analisi e specifica dei requisiti, la progettazione di sistema e la sua specifica, la codifica, il testing e infine il deployment.

5.1.1 Linguaggio di programmazione

Il linguaggio di programmazione che si è scelto per sviluppare il progetto è il C++. L'utilizzo del diagramma delle classi UML presuppone implicitamente la scelta di un linguaggio di programmazione orientato agli oggetti quale il C++. Inoltre, essendo le principali API di accesso alle smart card realizzate utilizzando il linguaggio c, la scelta è stata pressoché obbligata. Il motivo per cui si è scelto il C++ invece del C è essenzialmente basato sul fatto di voler sfruttare le potenzialità che offre il design *object oriented* lavorando ad un livello di astrazione più alto grazie anche all'utilizzo della libreria standard del C++ e di altre librerie disponibili sulla rete.

5.1.2 Ambienti e strumenti di sviluppo

Lo sviluppo sulla piattaforma Windows è avvenuto utilizzando una macchina con installato il sistema operativo Windows Vista Home Premium.

Gli strumenti utilizzati durante lo sviluppo sono:

- Microsoft Visual Studio 2008
- Microsoft Windows SDK
- Visual Assist X for Visual Studio
- Tortoise SVN
- Visual C++ Memory Leak Detector

Microsoft Visual Studio 2008 è l'ambiente di sviluppo attraverso il quale è stato possibile scrivere, compilare e linkare i file del progetto che sono stati opportunamente organizzati in soluzioni. Il Microsoft Windows SDK è un insieme di strumenti, esempi di codice, documentazione, compilatori, intestazioni e librerie che gli sviluppatori possono utilizzare per creare applicazioni per sistemi operativi Microsoft Windows. Per quanto riguarda SCOTT il Microsoft Windows SDK si è reso necessario per avere a disposizione le intestazioni e le librerie per scrivere applicazioni basate su smart card. Visual Assist X, è un plugin per Visual Studio che aumenta notevolmente la produttività durante lo sviluppo delle proprie applicazioni grazie alle avanzate funzionalità di refactoring e di supporto alla digitazione del codice. Attraverso il Visual C++ Memory Leak Detector inoltre è stato possibile verificare l'assenza di perdite di memoria a run-time.

Lo sviluppo sulla piattaforma Linux è avvenuto utilizzando una macchina con installato il sistema operativo Ubuntu 9.

Gli strumenti utilizzati sono:

- Compilatore gcc
- Make
- KDevelop 4
- KdeSVN

La compilazione e il linking di SCOTT in Linux si effettua attraverso l'utilizzo di opportuni makefile che automatizzano ed ottimizzano tutto il processo di build. KDevelop è un editor di codice che è stato utilizzato solamente alcune volte in quanto la maggior parte è stato scritto su Windows utilizzando Visual Studio.

Tutto il codice di SCOTT, organizzato in opportune sottocartelle, è stato inserito fin dal principio in un repository sub version in modo da avere a disposizione il controllo di versione e tenere sincronizzati i codici del progetto su Windows e Linux. A tal proposito sono stati utilizzati dei client sub version per effettuare in modo semplice le operazioni di commit e update (Tortoise SVN su Windows e KdeSVN su Linux).

5.1.3 Librerie di supporto

Il progetto SCOTT è stato realizzato sfruttando diverse librerie a supporto dello sviluppo. E' stato fatto un uso massiccio della **Standard Template Library** (**STL**) e in particolare dei container string, vector<T>, map<T>, list<T>, queue<T> e stack<T> e dei relativi iteratori per accedere e per manipolare queste strutture dati.

Sono state inoltre utilizzate diverse librerie presenti all'interno di **Boost** (Boost C++ Libraries s.d.) quali:

- Smart Pointer
- Regex
- File System e System
- Lexical cast
- Foreach
- String algo
- Testing

E' importante sottolineare che l'intero progetto è completamente basato sull'utilizzo degli Smart Pointer che assicurano un totale controllo sulla memoria

dinamica allocata, garantendo l'assenza di memory leak. Per maggiori informazioni sulle librerie di Boost fare riferimento a (Boost C++ Libraries s.d.).

Per realizzare il parser del descrittore del plugin è stata utilizzata la libreria **TinyXml** che offre semplici funzionalità per supportare la lettura e la manipolazione di file XML. La libreria TinyXml è stato scelta soprattutto perché è estremamente facile da utilizzare ed apprendere al contrario di altre sicuramente più avanzate ma con una curva di apprendimento decisamente più lenta. TinyXml soddisfa perfettamente i semplici requisiti di manipolazione XML richiesti dal toolkit. Per maggiori informazioni sulla libreria fare riferimento a (Tinyxml 2009).

La libreria **GNU Readline** infine è stata utilizzata per aggiungere alla shell funzionalità tipiche delle comuni shell, in particolare la possibilità di richiamare i comandi inseriti più recentemente in modo da poterli modificare e rieseguire con molta più semplicità. Per maggiori informazioni sulla libreria fare riferimento a (Readline library s.d.).

5.1.4 Testing

Il codice principale di SCOTT, racchiuso all'interno della "SCOTT Core Library", è stato in gran parte testato attraverso l'utilizzo di unit test in modo da evitare problematiche di regressione nei successivi aggiornamenti.

In particolare i test coprono il sistema dei tipi di SCOTT, le funzioni di formattazione, il parsing dei comandi e la risoluzione delle espressioni.

Per la scrittura dei test è stato utilizzato il framework di test presente all'interno di Boost, in modo da non aggiungere ulteriori dipendenze utilizzando un framework di test esterno ma altrettanto valido come ad esempio CppUnit.

La Figura 77 mostra l'esecuzione dell'applicazione Test che esegue l'intera batteria di test definiti per SCOTT.

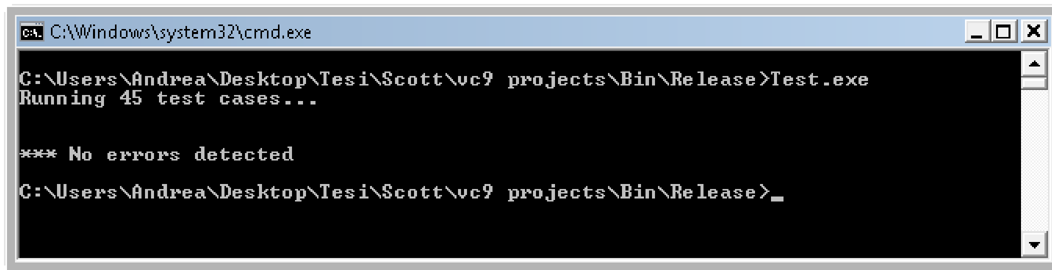


Figura 77 - Esecuzione della batteria di test per SCOTT

5.1.5 Documentazione del prodotto

Tutte le classi, i metodi e i membri presenti all'interno del progetto sono stati documentati attentamente attraverso l'utilizzo di commenti.

Doxygen è uno strumento per la generazione di documentazione di codice sorgente. Grazie ad esso è stato possibile costruire automaticamente una documentazione in formato HTML dell'intero progetto SCOTT che può essere consultata per comprenderne meglio il funzionamento. Inoltre, attraverso il software **GraphViz**, è stato possibile integrare all'interno della documentazione prodotta da Doxygen anche un insieme di grafici che descrivono le relazioni tra le varie classi presenti.

5.1.6 Licenza del prodotto

Affinché SCOTT possa diventare un toolkit ampiamente diffuso per l'invio di comandi alle smart card presenti sul mercato, un importante requisito è la sua totale apertura verso la comunità degli sviluppatori.

Per questo motivo la licenza con cui SCOTT è rilasciato è la GPL, la più diffusa licenza per il software libero. Grazie a questa licenza, l'intero codice sorgente sarà disponibile a tutti e potrà essere nel tempo migliorato. Sulla base di questa scelta è stato anche possibile sfruttare delle librerie open source rilasciate con licenza GPL quali TinyXml che altrimenti non si sarebbe potuto utilizzare.

5.1.7 Apparecchiatura usata

Per effettuare il testing di SCOTT, in particolare per quanto riguarda la verifica della corretta implementazione dei comandi sulle smart card, sono stati utilizzati i seguenti dispositivi:

- Lettore di Smart Card “Todos Argos Mini II USB”
- Smartcard Cryptoflex 8K

La Figura 78 mostra questi dispositivi.

Inoltre, solamente su Windows, in alcune occasioni è stato utilizzato un emulatore di smart card chiamato “Ugo's Smart Card Emulator”. Per maggiori informazioni sull’emulatore consultare (Chirico 2009).

Figura 78 - Lettore “Todos Argos Mini II” e smart card “Cryptoflex 8K”

5.2 Considerazioni preliminari e convenzioni

Per riuscire a comprendere il codice sorgente di SCOTT è importante capire le convenzioni che si è scelto di adottare per la sua stesura.

In tutto il progetto, il tipo *shared_ptr<T>* definito in Boost, nel file d'intestazione *<shared_ptr.hpp>*, è usato per definire uno smart pointer al tipo T e è utilizzato allo stesso modo di un comune puntatore al tipo T istanziato in memoria dinamica. Spesso, è necessario avere a disposizione strutture dati complesse quali vettori e mappe che addirittura devono poter essere passate come argomenti di opportuni metodi. Risulta quindi indispensabile anche avere la possibilità di creare vettori di smart pointer e anche smart pointer di vettori di smart pointer in modo da potersi disinteressare delle problematiche relative alla gestione della memoria dinamica soprattutto in presenza di eccezioni. Sarà l'implementazione interna degli smart pointer a rilasciare la memoria quando sarà opportuno.

L'indubbio vantaggio di potersi concentrare sul design piuttosto che su problematiche di basso livello, come la gestione della memoria, porta ad una riduzione della leggibilità del codice a causa di un eccessivo utilizzo di template annidati. La Figura 79 mostra la creazione di una istanza di uno smart pointer a un vettore di smart pointer a SValue e la creazione di una istanza di uno smart pointer a una mappa con chiave stringa e valore smart pointer a SValue. La figura rende immediatamente evidente la pesantezza della notazione.

```
shared_ptr<vector<shared_ptr<SValue>>> v(new vector<shared_ptr<SValue>>());
shared_ptr<map<string, shared_ptr<SValue>>> m(new map<string, shared_ptr<SValue>>());
```

Figura 79 - Creazione di smart pointer senza utilizzare alias

All'interno del toolkit si è quindi deciso di creare degli alias per i tipi smart pointer più comuni utilizzando la parola chiave *typedef*. La convenzione scelta per un tipo smart pointer è aggiungere al nome del tipo il suffisso "Sp". A questo scopo sono state create le macro CREATE_ALIAS (Figura 80) e CREATE_MAP_ALIAS (Figura 81), definite all'interno del file *<alias.h>*, che dato il nome di un tipo permettono di creare automaticamente tutti gli alias necessari. E' buona cosa utilizzare queste macro prima della dichiarazione di

ogni nuovo tipo in modo tale da rendere disponibili tutti gli alias attraverso l'inclusione del solo file d'intestazione.

```
#define CREATE_ALIAS( T )                                        \
    class DECLARATION_DLL T ;                                    \
    typedef shared_ptr< T > T##Sp ;                              \
    typedef vector< T##Sp > T##SpVector ;                        \
    typedef shared_ptr< T##SpVector > T##SpVectorSp ;            \
```

Figura 80 - Macro CREATE_ALIAS

```
#define CREATE_MAP_ALIAS( T )                              \
    typedef map<string, T##Sp > T##SpMap;                  \
    typedef pair<string, T##Sp >  T##SpMap##Pair;          \
    typedef shared_ptr< T##SpMap > T##SpMapSp ;
```

Figura 81 - Macro CREATE_MAP_ALIAS

Grazie a queste due macro è possibile aumentare notevolmente la leggibilità del codice come mostrato in Figura 82.

```
SValueSpVectorSp v(new SValueSpVector());
SValueSpMapSp m(new SValueSpMap());
```

Figura 82 - Creazione di smart pointer utilizzando alias

5.3 Architettura del sistema di tipi

Il sistema di tipi è sicuramente la parte principale della SCOTT Core Library. E' importante comprendere e conoscere le classi che definiscono i tipi in particolare per coloro che desiderano sviluppare plugin. Tuttavia, i diagrammi delle classi che sono mostrati nei seguenti paragrafi non sono completi e presentano solamente i membri e i metodi più significativi. Per maggiori informazioni fare sempre riferimento alla documentazione generata da Doxygen o accedere direttamente ai file sorgenti del progetto.

5.3.1 Tipi semplici

La Figura 83 mostra il diagramma delle classi relative ai tipi semplici di SCOTT, che sono stati presentati nel paragrafo 3.1.

La classe astratta ***SValue*** è la classe base per tutti i tipi (sia semplici che complessi) definiti all'interno di SCOTT. Il metodo *GetType()* restituisce il tipo della variabile sotto forma di un enumeratore chiamato *SType*, mentre il metodo *GetTypeName()* restituisce una stringa che rappresenta il nome fully qualified del tipo. I metodi *IsSimpleType()* e *IsComplexType()* permettono di sapere se un tipo è semplice o complesso. Il metodo *ToString()* permette di ottenere una rappresentazione sotto forma di stringa del valore dell'oggetto e il metodo *ToString(format:string)* permette anche di specificare il formato di rappresentazione.

Tutti i tipi semplici derivano da una classe astratta di nome ***SSimpleType*** che a sua volta deriva da *SValue*.

Le classi che permettono di creare tipi semplici sono ***SByte***, ***SNumber***, ***SChar*** e ***SString***. Ciascuna classe ha un campo privato per memorizzare il valore della variabile e un metodo *Get()* per leggerlo.

Per creare un'istanza di una variabile di tipo semplice si utilizza il metodo statico *Create()* che restituisce uno smart pointer al tipo stesso. E' possibile anche creare una variabile di tipo semplice partendo dalla sua rappresentazione come stringa attraverso il metodo *Parse()*.

La Figura 84 mostra come, via codice, è possibile creare alcune variabili di tipo semplice attraverso i metodi *Create()* e *Parse()*.

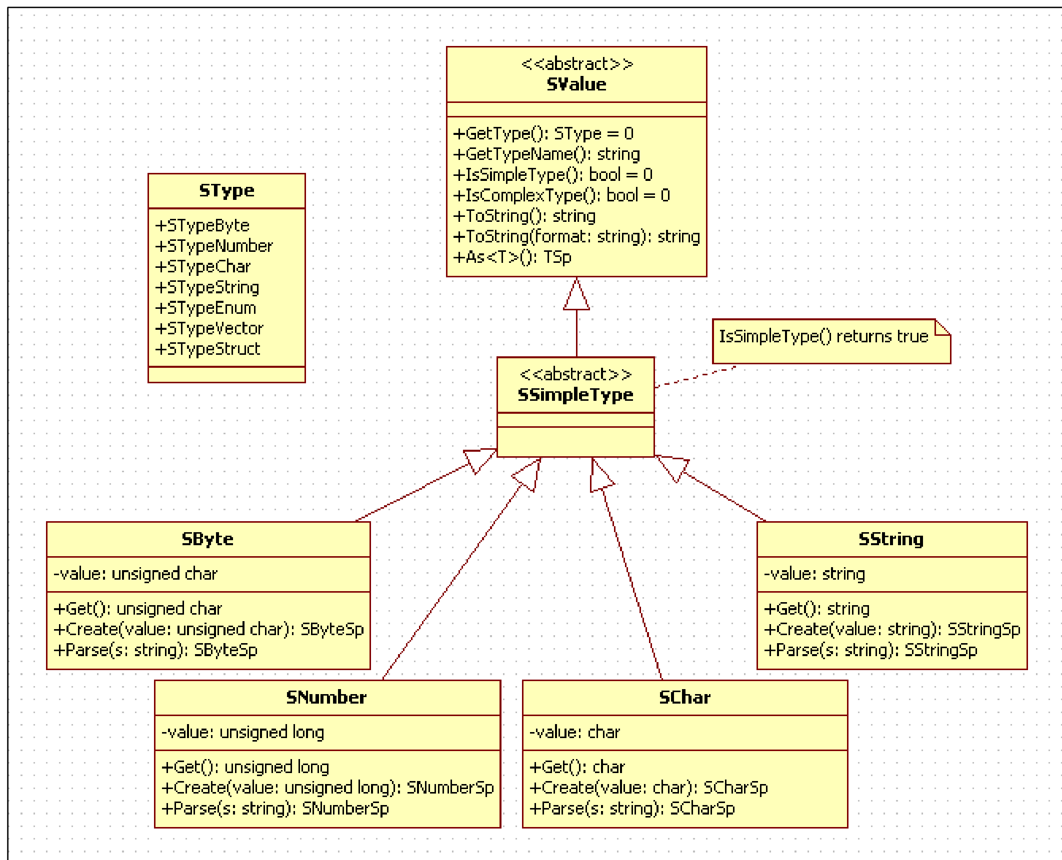


Figura 83 - Gerarchia delle classi per i tipi semplici

```
SByteSp b1 = SByte::Create(10);
SByteSp b2 = SByte::Parse("0x0A");

SNumberSp n1 = SNumber::Create(1000);
SNumberSp n2 = SNumber::Parse("0x3E8");

SCharSp c1 = SChar::Create('A');
SCharSp c2 = SChar::Parse("'A'");

SStringSp s1 = SString::Create("SCOTT");
SStringSp s2 = SString::Parse("\"SCOTT\"");
```

Figura 84 - Creazione di tipi semplici via codice

5.3.2 Tipi complessi

La Figura 85 mostra il diagramma delle classi relative ai tipi complessi di SCOTT, che sono stati presentati nel paragrafo 3.1.

Tutti i tipi complessi derivano da una classe astratta di nome ***SComplexType*** che a sua volta deriva da *SValue*. Un tipo complesso è descritto attraverso uno specifico descrittore di tipo che può essere ottenuto attraverso il metodo comune *GetTypeDescriptor()*. Esiste quindi una gerarchia di descrittori di tipo come mostrato dalla Figura 86. In particolare i descrittori di tipo per gli *SEnum* e per le *SStruct* utilizzano internamente altre classi per memorizzare informazioni relative alle entry e ai membri. Queste classi appartengono ad un'altra gerarchia mostrata in Figura 87.

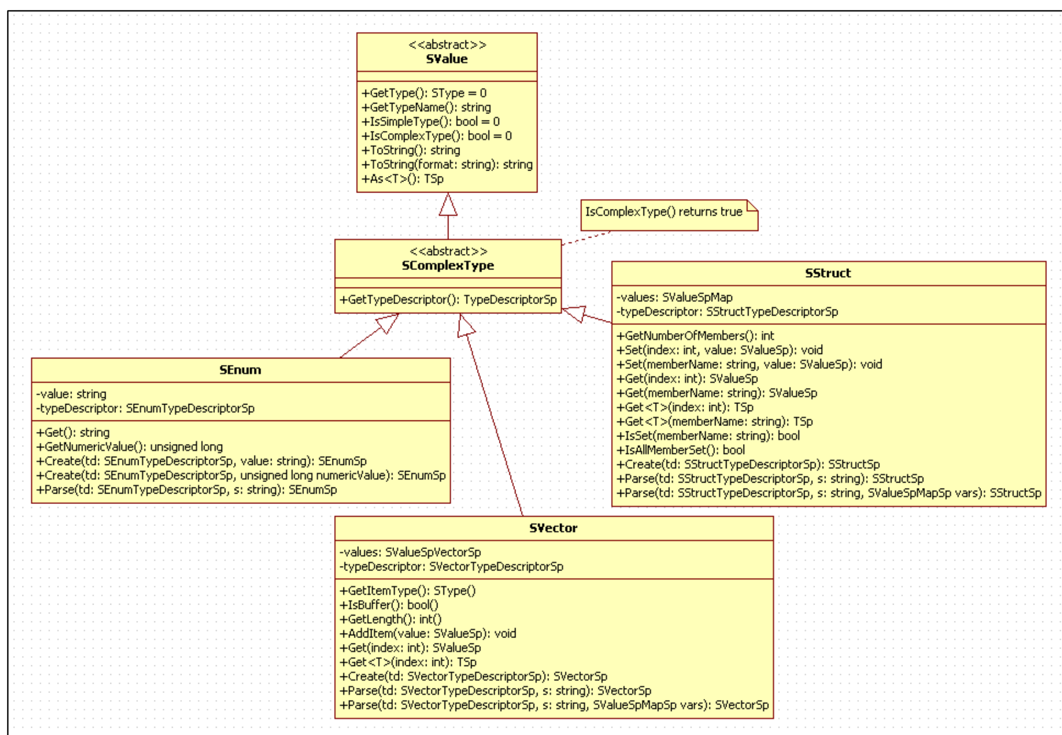


Figura 85 - Gerarchia delle classi per i tipi complessi

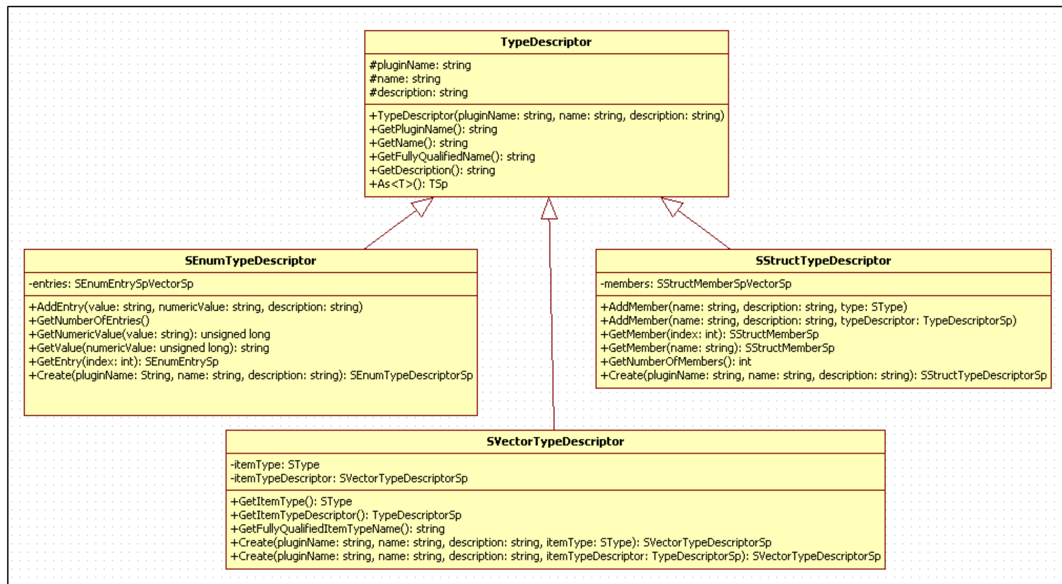


Figura 86 - Gerarchia dei Type Descriptor

Le classi che permettono di creare tipi complessi sono ***SEnum***, ***SVector*** e ***SStruct***. Ciascuna di queste classi ha una interfaccia differente, quindi saranno analizzate separatamente.

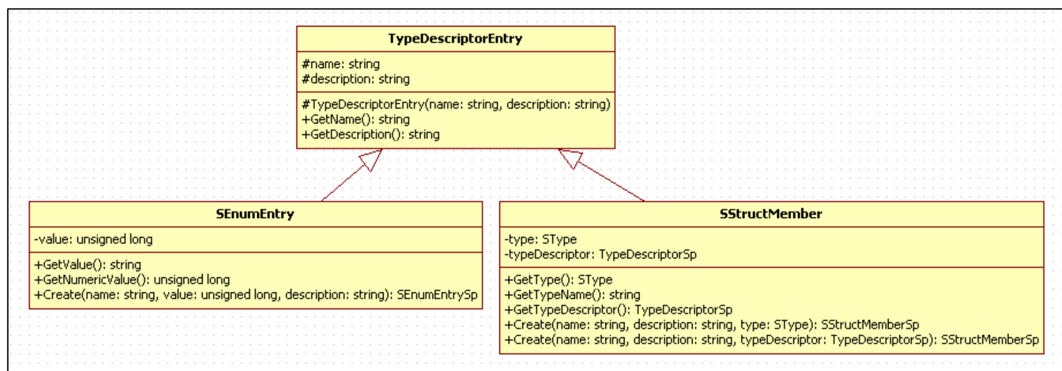


Figura 87 - Gerarchia dei TypeDescriptorEntry

Prima di creare una variabile di tipo *SEnum* è necessario costruire una istanza di un *SEnumTypeDescriptor* che permette di specificare il range di valori che l'enumeratore può assumere. Il descrittore è creato attraverso il metodo

SEnumTypeDescriptor::Create() a cui si devono passare il nome del plugin, il nome e una descrizione del tipo. Il metodo *AddEntry()* permette di inserire una entry nel descrittore specificandone il nome, il valore numerico associato e la descrizione. A questo punto attraverso il metodo *SEnum::Create()* è possibile creare una variabile di tipo *SEnum* specificando come argomenti il descrittore da utilizzare e il valore di inizializzazione. Una volta che è stata creata la variabile, è possibile leggere i suoi valori attraverso i metodi *Get() e GetNumericValue()*. Inoltre è anche possibile costruire una variabile di tipo *SEnum* partendo da una sua rappresentazione su stringa utilizzando il metodo *SEnum::Parse()*. La Figura 88 mostra un esempio di creazione ed utilizzo di una variabile di tipo *SEnum*.

Prima di creare una variabile di tipo *SVector* è necessario costruire una istanza di un *SVectorTypeDescriptor* che permette di specificare il tipo degli elementi del vettore. Il descrittore è creato attraverso uno dei metodi *SEnumTypeDescriptor::Create()* a cui si devono passare il nome del plugin, il nome e una descrizione del tipo e il tipo degli elementi specificato con un *SType* o un *TypeDescriptorSp* a seconda se il tipo degli elementi è semplice o complesso. A questo punto attraverso il metodo *SVector::Create()* è possibile creare una variabile di tipo *SVector* specificando come unico argomento il descrittore da utilizzare. Una volta che è stata creata la variabile, è possibile aggiungere elementi con il metodo *AddItem()* e leggere gli elementi attraverso i metodi *Get() e Get<T>()*. Inoltre è anche possibile costruire una variabile di tipo *SVector* partendo da una sua rappresentazione su stringa utilizzando il metodo *SEnum::Parse()*. La Figura 89 mostra un esempio di creazione e utilizzo di una variabile di tipo *SVector*.

```
SEnumTypeDescriptorSp etd =
    SEnumTypeDescriptor::Create("plugin", "colors", "A color enumerator.");

etd->AddEntry("red", 0, "red color");
etd->AddEntry("yellow", 1, "yellow color");
etd->AddEntry("green", 2, "green color");

SEnumSp e = SEnum::Create(etd, "red");

string val = e->Get();
unsigned long num = e->GetNumericValue();
```

Figura 88 - Creazione e utilizzo di una variabile di tipo SEnum

```
SVectorTypeDescriptorSp vtd =
    SVectorTypeDescriptor::Create("plugin", "vnumber", "Vector of SNumber", STypeNumber);

SVectorSp v = SVector::Create(vtd);
v->AddItem( SNumber::Create(1) );
v->AddItem( SNumber::Create(2) );
v->AddItem( SNumber::Create(3) );

SValueSp v0 = v->Get(0);
SNumberSp v1 = v->Get<SNumber>(1);
```

Figura 89 - Creazione e utilizzo di una variabile di tipo SVector

Un procedimento del tutto analogo è necessario per creare una variabile di tipo *SStruct.* Prima si deve costruire una istanza di un *SStructTypeDescriptor* che permette di specificare le informazioni relative ai membri della struttura. Il descrittore è creato attraverso il metodo SStruct*TypeDescriptor::Create()* a cui si devono passare il nome del plugin, il nome e una descrizione del tipo. Successivamente con uno dei metodi *AddMember()* è possibile definire membri di tipo semplice o complesso, specificando per ciascun tipo, nome e descrizione. A questo punto attraverso il metodo *SStruct::Create()* è possibile creare una variabile di tipo *SStruct* specificando come unico argomento il descrittore da utilizzare. Una volta che è stata creata la variabile, è possibile impostare il valore dei membri attraverso i metodi *Set()* e leggere il valore dei membri attraverso i metodi *Get() e Get<T>()*. Inoltre è anche possibile costruire una variabile di tipo *SStruct* partendo da una sua rappresentazione su stringa utilizzando il metodo *SStruct::Parse()*. La Figura 90 mostra un esempio di creazione ed utilizzo di una variabile di tipo *SStruct*.

```
SStructTypeDescriptorSp std =
    SStructTypeDescriptor::Create("plugin", "SimpleStruct", "Una semplice struttura");
std->AddMember("b", "Un byte", STypeByte);
std->AddMember("num", "Un numbero", STypeNumber);
std->AddMember("c", "Un carattere", STypeChar);
std->AddMember("str", "Una stringa", STypeString);

SStructSp s = SStruct::Create(std);
s->Set("b", SByte::Create(10) );
s->Set("num", SNumber::Create(1000) );
s->Set("c", SChar::Create('C') );
s->Set("str", SString::Create("Hello") );

SByteSp b = s->Get<SByte>("b");
SNumberSp num = s->Get<SNumber>("num");
SCharSp c = s->Get<SChar>("c");
SStringSp str = s->Get<SString>("str");
```

Figura 90 - Creazione e utilizzo di una variabile di tipo SStruct

5.4 Il meccanismo dei comandi

Ogni comando di plugin all'interno della SCOTT Core Library è gestito attraverso una istanza della classe *Command* (Figura 91). Prima di creare un comando ed eseguirlo è importante definire la sua struttura rappresentata da nome, parametri di input e parametri di output. La classe *CommandDescriptor* (Figura 92) permette di memorizzare tutte le informazioni relative alla struttura del Comando. Per memorizzare le informazioni relative ai parametri si utilizzano le classi *InputParameter* e *OutputParameter* che appartengono alla gerarchia dei *Parameter* (Figura 93).

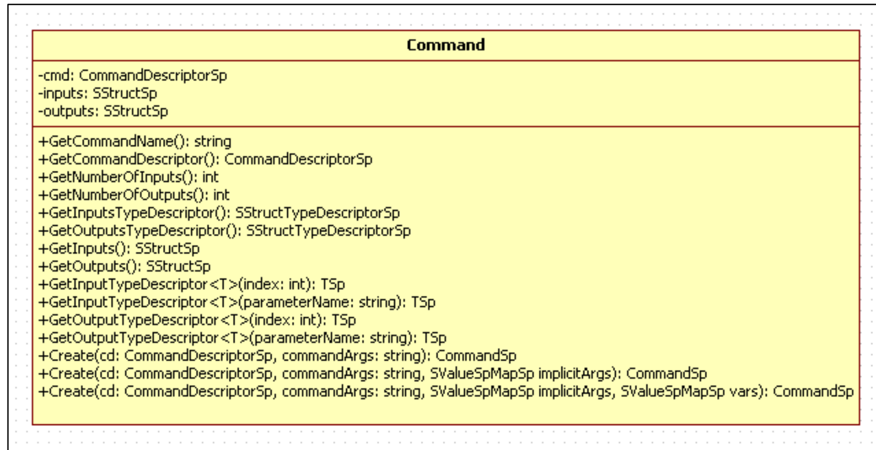


Figura 91 - La classe Command

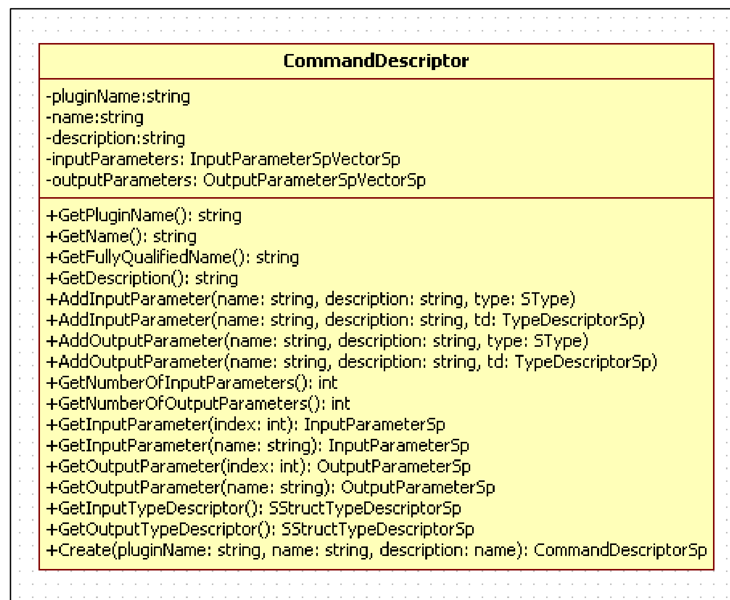


Figura 92 - La classe CommandDescriptor

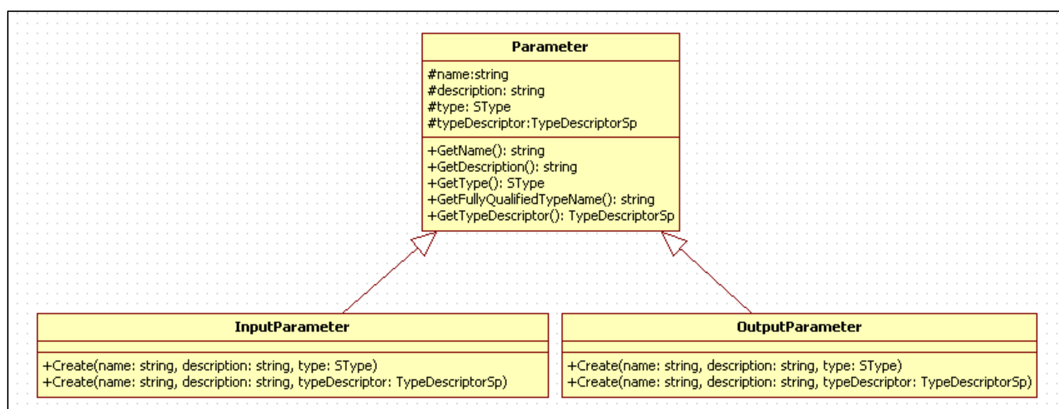


Figura 93 - Gerarchia dei Parameter

Per creare una istanza di un *CommandDescriptor* si utilizza il metodo *CommandDescriptor::Create()* cui si deve passare il nome del plugin, il nome del comando e una descrizione del comando. Poi con i metodi *AddInputParameter()* e *AddOutputParameter()* è possibile inserire le informazioni relative ai parametri di input ed ai parametri di output. Per ogni parametro si deve specificare il nome, una descrizione e il tipo.

Mentre la classe *CommandDescriptor* rappresenta la struttura di uno specifico comando, la classe *Command* rappresenta una specifica istanza di un comando con tutti i rispettivi valori che si desidera associare ai parametri di input. Una istanza della classe *Command* contiene tutte le informazioni necessarie affinché il motore di SCOTT possa eseguire direttamente il comando.

Per creare una istanza di un *Command* si utilizza il metodo *Command::Create()* a cui si deve passare almeno il *CommandDescriptor* e la rappresentazione su stringa dei parametri di input che si vuole passare al comando. Questa rappresentazione è del tutto equivalente a quella usata per valorizzare una struttura (senza le parentesi graffe) solamente che in questo caso i membri sono i parametri di input del comando. E' importante sottolineare che la classe *Command* utilizza internamente una struttura per memorizzare i parametri di input e quindi genera automaticamente un descrittore a partire dal *CommandDescriptor* fornito come parametro. La stessa cosa avviene per quanto riguarda i parametri di Output. Esistono altri due overload del metodo *Command::Create()* per supportare il meccanismo delle variabili implicite (presentato nel paragrafo 3.2.8) e l'utilizzo di espressioni nella stringa di input. Possono essere infatti passati al metodo un insieme di variabili da utilizzare per la risoluzione delle espressioni e un insieme di variabili implicite.

Una volta che si ha a disposizione una istanza di un comando, è possibile leggere il valore dei parametri di input chiamando il metodo *GetInputs*(). Attraverso il metodo *GetOutputs()* è invece possibile ottenere un riferimento alla struttura che dovrà contenere i parametri di output. Impostare un valore ad un membro di questa struttura di fatto permette di impostare il valore di un parametro di

output del comando. Questi due metodi sono utilizzati principalmente dagli sviluppatori di plugin per SCOTT.

La Figura 94 mostra un esempio di creazione, tramite codice, di un comando che permette di fare la somma tra due numeri. Nell'esempio è calcolata la somma e salvata nel parametro di output. In realtà le istruzioni relative al salvataggio dei risultati dovranno essere inserite all'interno di un plugin come mostrato in dettaglio nel capitolo 6.

```
// Creazione del CommandDescriptor
CommandDescriptorSp cd =
    CommandDescriptor::Create("math", "somma", "Somma due numeri.");

cd->AddInputParameter("a", "Primo addendo", STypeNumber);
cd->AddInputParameter("b", "Secondo addendo", STypeNumber);

cd->AddOutputParameter("ris", "Risultato della somma", STypeNumber);

// Creazione del comando
CommandSp cmd = Command::Create(cd, "a = 10, b = 20");

// Prelevo i parametri di input del comando
SStructSp inputs = cmd->GetInputs();

unsigned long a = inputs->Get<SNumber>("a")->Get();
unsigned long b = inputs->Get<SNumber>("b")->Get();

// Imposto il risultato del comando
SStructSp outputs = cmd->GetOutputs();

outputs->Set("ris", SNumber::Create( a + b ) );
```

Figura 94 - Creazione e utilizzo di un comando via codice

5.5 Il repository dei descrittori

La classe *DescriptorsRepository* riveste un ruolo molto importante nell'architettura di SCOTT. Le sue responsabilità consistono nella gestione di tutti i descrittori di tipo, i descrittori di comando e nel fornire una interfaccia completa per accedere ad essi attraverso la notazione fully qualified.

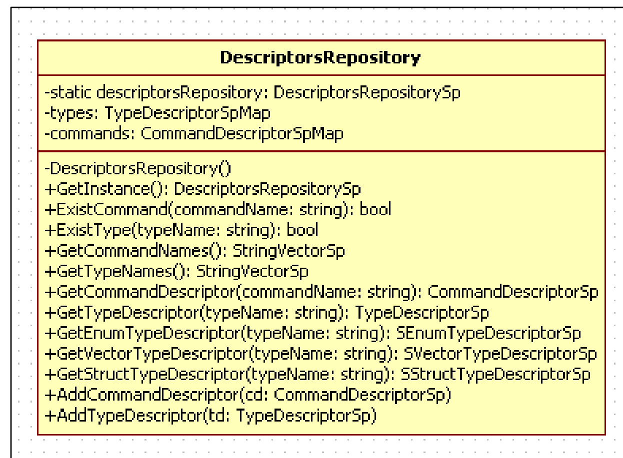


Figura 95 - Classe DescriptorsRepository

Esiste una sola istanza di questa classe all'interno di SCOTT che può essere ottenuta attraverso il metodo *DescriptorsRepository::GetInstance()*. L'implementazione segue il design pattern Singleton (Gamma 2002).

I metodi *ExistCommand()* ed *ExistType()* verificano se un particolare comando o tipo è definito. I metodi *GetCommandNames()* e *GetTypeNames()* restituiscono un elenco completo di tutti i comandi e di tutti i tipi caricati. I metodi *GetCommandDescriptor()* e *GetTypeDescriptor()* restituiscono un particolare descrittore di comando o descrittore di tipo. Esistono inoltre versioni specializzate per ottenere direttamente una istanza del descrittore di tipo desiderato. Con i metodi *AddCommandDescriptor()* e *AddTypeDescriptor()* infine si registrano i descrittori all'interno del repository.

5.6 Il sistema di risoluzione dei nomi

Il sistema di risoluzione dei nomi permette di determinare il nome assoluto (fully qualified) di un tipo o di un comando a partire da un nome relativo o assoluto. La risoluzione dei nomi può essere realizzata secondo logiche differenti. Per questo motivo è stata creata una interfaccia di nome *INameResolutor* (Figura 96) per

astrarre le operazioni richieste. Il metodo *ResolveTypeName()* permette di risolvere il nome di un tipo mentre il metodo *ResolveCommandName()* permette di risolvere il nome di un comando. Il metodo *AddOrSelectPlugin()* invece permette di specificare i plugin da coinvolgere nel processo di risoluzione dei nomi. Maggiori informazioni sulla classe Plugin si possono trovano nel capitolo 6.4.1.

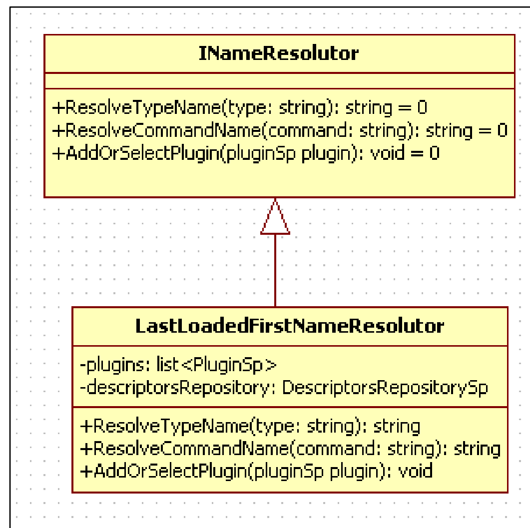


Figura 96 - Classi per la risoluzione dei nomi

La classe *LastLoadedFirstNameResolutor* è l'unica implementazione del sistema dei nomi che utilizza una logica di tipo "Last Loaded First". Questa logica consiste nel dare, nel processo di risoluzione dei nomi, la priorità all'ultimo plugin caricato.

5.7 La risoluzione delle espressioni

Una espressione può essere:

- direttamente il valore di una variabile
- il nome esatto di una variabile
- un accesso ad un membro di una struttura
- un accesso ad un elemento di un vettore

- una sequenza di accessi a membri di strutture ed elementi di vettori

La classe *ExpressionResolutor* (Figura 97) offre tre metodi da utilizzare per risolvere le espressioni. Tutti questi metodi accettando in ingresso una stringa che rappresenta l'espressione da risolvere e un insieme di variabili da utilizzare durante il processo di risoluzione. E' possibile specificare un ulteriore parametro per indicare il tipo della variabile che dovrebbe essere restituita. In quest'ultimo caso se l'espressione individua una variabile che non corrisponde al tipo specificato, viene sollevata una eccezione.

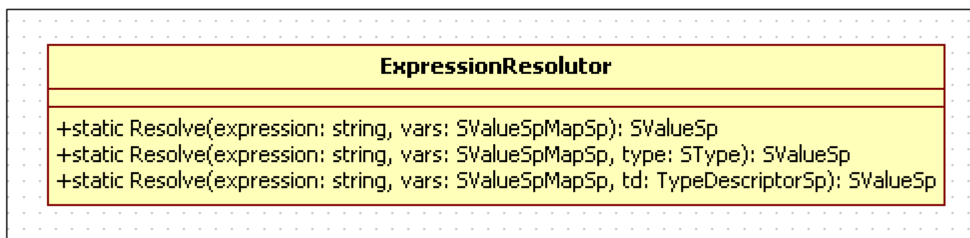


Figura 97 - Classe ExpressionResolutor

5.8 La gestione delle eccezioni

Tutto il codice di Scott utilizza il meccanismo delle eccezioni per segnalare la presenza di errori durante l'esecuzione del codice. La Figura 96 mostra la gerarchia delle eccezioni di SCOTT.

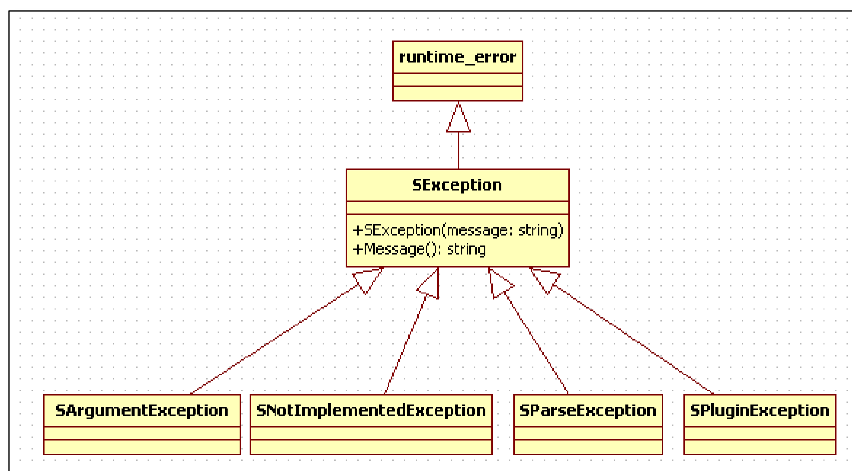


Figura 98 - Gerarchia delle eccezioni di SCOTT

In particolare *SArgumentException* viene sollevata quando un argomento passato ad un metodo non è valido, *SParseException* quando si verifica una problema durante una operazione di parsing e *SPluginException* quando si verifica un errore all'interno di un plugin. L'ultima classe è quella che deve essere utilizzata dagli sviluppatori di plugin per segnalare la presenza di errori all'interno dei comandi implementati.

5.9 L'interfaccia della SCOTT Core Library

La classe Scott rappresenta il punto d'ingresso verso la SCOTT Core Library. Sostanzialmente il compito principale della libreria è fare da ponte tra gli applicativi che rappresentano le viste (come ad esempio la Shell) e le funzionalità offerte dai plugin. Con il metodo *GetInstalledPluginNames()* è possibile ottenere l'elenco dei plugin installati sulla macchina mentre con il metodo *CreatePlugin()* è possibile caricare un particolare plugin. In particolare quest'ultimo metodo accetta come parametri il nome del plugin da caricare, una mappa dei plugin già caricati (necessaria per verificare se eventuali dipendenze sono rispettate) e un riferimento ad un risolutore di nomi.

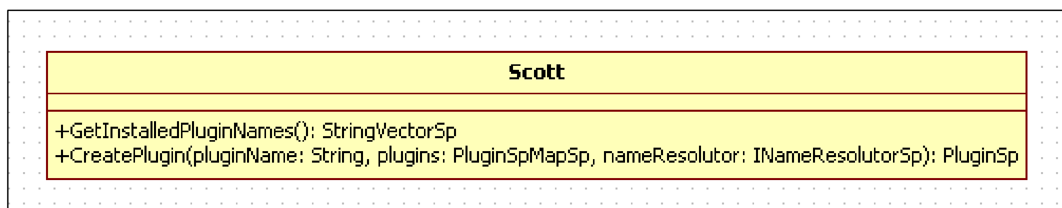


Figura 99 - Classe Scott

5.10 Come ottenere il codice sorgente di SCOTT

Il sorgente di SCOTT è disponibili all'indirizzo http://scott.codeplex.com/.

6. Estendere SCOTT

6.1 Come installare un nuovo plugin

Ogni plugin di SCOTT è costituito da due file:

- Un descrittore di plugin (Scott Plugin Descriptor)
- Una libreria dinamica (Scott Plugin Library)

Il primo file è un documento xml con estensione ***.spd*** che contiene tutte le informazioni relative a dipendenze, tipi e comandi definiti dal plugin. Il secondo file è una libreria dinamica (con estensione ***.dll*** su Windows e ***.so*** su Linux) che racchiude l'implementazione effettiva dei comandi del plugin. Entrambi questi file devono avere un nome esattamente uguale a quello del plugin stesso.

Per installare un nuovo plugin è quindi sufficiente possedere questi due file e copiarli all'interno della cartella *Plugins* sotto la directory d'installazione del toolkit.

6.2 Descrizione del plugin math

In questo paragrafo, si riporta la descrizione di un semplice plugin di esempio (chiamato ***math***) che sarà realizzato in forma completa all'interno di questo capitolo per mostrare, da un punto di vista pratico, tutti i passi necessari durante lo sviluppo di un nuovo plugin per SCOTT.

Sono definiti i seguenti tipi di dato:

- **Point**: una struttura che memorizza attraverso due *SNumber* le coordinate x e y di un punto nel piano cartesiano. Si ricorda che il tipo *SNumber* è in grado solamente di memorizzare valori numerici positivi. Le coordinate dovranno essere comprese tra 0 e 100 estremi inclusi.

- **Quadrante**: un enumeratore utilizzato per indicare un quadrante all'interno della regione di spazio ammissibile. I possibili valori sono:
    - Primo, Secondo, Terzo e Quarto

  La Figura 100 mostra graficamente la posizione dei quadranti.
- **VPoint**: un vettore di Point.

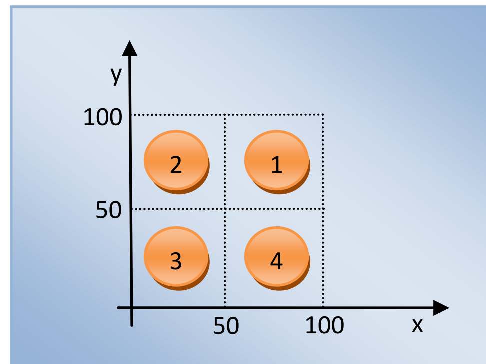


Figura 100 - Posizione dei quadranti in 'math'

Il plugin implementa solamente due comandi:

- ***GetQuadrante***()
- ***Filtro***()

Il comando *GetQuadrante()* prende in ingresso un punto e restituisce il quadrante in cui si trova quel punto.

Il comando *Filtro()* invece, prende in ingresso un vettore di punti e un quadrante e restituisce un vettore contenente solamente i punti che appartengono al quadrante specificato. Di fatto questo comando si comporta come un filtro.

Questo plugin è molto semplice ma sufficiente per illustrare in maniera abbastanza completa le operazioni da compiere per estendere SCOTT.

6.3 Il descrittore di plugin

Il descrittore di plugin (SCOTT Plugin Descriptor) permette di specificare dipendenze, tipi e comandi definiti all'interno del plugin in modo estremamente

semplice, grazie all'utilizzo di opportuni elementi ed attributi xml. Grazie ad esso lo sviluppatore del plugin non si dovrà preoccupare di costruire manualmente via codice le istanze dei Type Descriptor o dei Command Descriptor in quanto queste saranno costruite dalla Core Library al momento del caricamento del plugin.

6.3.1 Lo schema XSD

La grammatica del descrittore di plugin è descritta da uno Schema XSD che per completezza è riportato in questo paragrafo.

Questa grammatica è contenuta all'interno di un file di nome ***spd.xsd*** che può essere utilizzato insieme a un editor XML per rendere più agevole la scrittura del descrittore di plugin. In Microsoft Visual Studio, copiando questo file nella cartella *Xml\Schemas*, si riesce ad avere un completo riconoscimento di tag e attributi. Questo file quindi oltre a permettere la validazione del documento consente di aumentare la produttività nella creazione del descrittore di plugin.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
targetNamespace="http://www.scott.it/spd-schema"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Plugin">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Dependencies" minOccurs="0" maxOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Dependency" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:attribute name="plugin" type="xs:string" use="required" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Commands" minOccurs="0" maxOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Command" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Description" type="xs:string" minOccurs="0" />
                    <xs:element name="Inputs">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="Parameter" minOccurs="0"
                                                       maxOccurs="unbounded">
                            <xs:complexType>
                              <xs:attribute name="name" type="xs:string" use="required" />
                              <xs:attribute name="type" type="xs:string" use="required" />
                              <xs:attribute name="description" type="xs:string"
                                            use="optional" />
```

```
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="Outputs" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Parameter" minOccurs="0"
                                               maxOccurs="unbounded">
                    <xs:complexType>
                      <xs:attribute name="name" type="xs:string" use="required" />
                      <xs:attribute name="type" type="xs:string" use="required" />
                      <xs:attribute name="description" type="xs:string"
                                    use="optional" />
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
          <xs:attribute name="name" type="xs:string" use="required" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Types" minOccurs="0" maxOccurs="1">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Enums" minOccurs="0" maxOccurs="1">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Enum" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Entry" minOccurs="1" maxOccurs="unbounded">
                    <xs:complexType>
                      <xs:attribute name="name" type="xs:string" use="required" />
                      <xs:attribute name="value" type="xs:string" use="optional"/>
                      <xs:attribute name="description" type="xs:string"
                                    use="optional" />
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
                <xs:attribute name="name" type="xs:string" use="required" />
                <xs:attribute name="description" type="xs:string" use="optional"/>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Vectors" minOccurs="0" maxOccurs="1">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Vector" minOccurs="0" maxOccurs="1">
              <xs:complexType>
                <xs:attribute name="name" type="xs:string" use="required" />
                <xs:attribute name="itemType" type="xs:string" use="required" />
                <xs:attribute name="description" type="xs:string" use="optional"/>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Structs" minOccurs="0" maxOccurs="1">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Struct" minOccurs="0" maxOccurs="1">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Member" minOccurs="1" maxOccurs="unbounded">
                    <xs:complexType>
                      <xs:attribute name="name" type="xs:string" use="required" />
                      <xs:attribute name="type" type="xs:string" use="required" />
```

```
                              <xs:attribute name="description" type="xs:string"
                                            use="optional" />
                            </xs:complexType>
                          </xs:element>
                        </xs:sequence>
                        <xs:attribute name="name" type="xs:string" use="required" />
                        <xs:attribute name="description" type="xs:string" use="optional"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

La Figura 101 mostra la struttura di base di un descrittore di plugin. Il tag di root è *<Plugin>* e l'attributo *xmlns* specifica il namespace relativo allo schema XSD descritto precedentemente.

La sezione *<Dependencies>* permette di specificare quali sono i plugin da cui si dipende; le sezioni *<Commands>* e *<Types>* definiscono rispettivamente i comandi e i tipi complessi del plugin.

Nei successivi paragrafi saranno spiegate nel dettaglio le varie sezioni di cui è composto un generico descrittore.

```
<?xml version="1.0" encoding="utf-8"?>

<Plugin xmlns="http://www.scott.it/spd-schema">

  <Dependencies>
  </Dependencies>

  <Commands>
  </Commands>

  <Types>
  </Types>

</Plugin>
```

Figura 101 - Struttura base del descrittore di plugin.

6.3.2 Specifica delle dipendenze

Attraverso la sezione ***<Dependencies>*** è possibile specificare il nome dei plugin dai quali dipende il plugin che si sta realizzando. La Figura 102 mostra un esempio tratto dal descrittore del plugin “cryptoflex8”.

```
<Dependencies>
  <Dependency plugin="pcsc" />
  <Dependency plugin="iso7816-4" />
</Dependencies>
```

Figura 102 - Sezione <Dependencies> nel descrittore di plugin

I plugin specificati in questa sezione saranno caricati automaticamente (se non è già stato fatto prima) nel momento in cui si carica questo plugin.

6.3.3 Definizione dei tipi di dato

All’interno della sezione *<Types>* sono definiti tutti i tipi di dato complessi necessari al plugin attraverso i tag *<Enums>*, *<Vectors>* e *<Structs>*.

Grazie al tag *<Enums>* è possibile definire un elenco di enumeratori ciascuno descritto attraverso il tag *<Enum>*. Per ogni enumeratore è possibile specificare nome e descrizione attraverso gli attributi *name* e *description*. Con l’elemento *<Entry>* s’inseriscono tutti i valori che possono essere assunti dal nuovo tipo specificando nome, valore e descrizione tramite gli attributi *name*, *value* e *description*. L’attributo *value* non è obbligatorio e se non è specificato, sarà impostato automaticamente al valore successivo a quello dell’elemento precedente. Se il primo elemento non è impostato, il suo valore sarà considerato pari a zero.

La Figura 103 mostra la definizione del tipo *Protocol* all’interno del plugin “*pcsc*”.

```
<Enums>

  <Enum name="Protocol" description="Acceptable protocols for the connection.">
    <Entry name="SCARD_PROTOCOL_UNDEFINED" value="0x00000000" description="" />
    <Entry name="SCARD_PROTOCOL_T0" value="0x00000001" description="T=0 is an acceptable protocol." />
    <Entry name="SCARD_PROTOCOL_T1" value="0x00000002" description="T=1 is an acceptable protocol." />
    <Entry name="SCARD_PROTOCOL_RAW" value="0x00010000" description="" />
    <Entry name="SCARD_PROTOCOL_Tx" value="0x00000003" description="T=0 or T=1" />
    <Entry name="SCARD_PROTOCOL_DEFAULT" value="0x80000000" description="" />
  </Enum>

  ...

</Enums>
```

Figura 103 - Sezione <Enums> nel descrittore di plugin

Grazie al tag *<Vectors>* è possibile definire un elenco di vettori ciascuno descritto attraverso il tag *<Vector>*. Per ogni vettore è possibile specificare nome e descrizione attraverso gli attributi *name* e *description*. Inoltre con l'attributo *itemType* si specifica il tipo (eventualmente anche complesso) di ciascuno degli elementi del vettore. Il tipo deve essere specificato utilizzando la notazione fully qualified e può essere anche definito in uno dei plugin dipendenti.

La Figura 104 mostra la definizione dei vettori *Buffer* e *Vstring* del plugin "*pcsc*".

```
<Vectors>
  <Vector name="Vstring" itemType="sstring" description="Vector of sstring."/>
  <Vector name="Buffer" itemType="sbyte" description="Vector of sbyte."/>
</Vectors>
```

Figura 104 - Sezione <Vectors> nel descrittore di plugin

Grazie al tag *<Structs>* è possibile definire un elenco di strutture ciascuna descritta attraverso il tag *<Struct>*. Per ogni struttura è possibile specificare nome e descrizione attraverso gli attributi *name* e *description*. Con l'elemento *<Member>* s'inseriscono tutti i membri della struttura specificando nome, tipo e descrizione tramite gli attributi *name*, *type* e *description*. Il tipo deve essere specificato utilizzando la notazione fully qualified e può essere anche definito in uno dei plugin dipendenti.

La Figura 105 mostra la definizione di un tipo *point* che memorizza le coordinate di un punto nello spazio cartesiano.

```xml
<Structs>

  <Struct name="point" description="Un punto.">
    <Member name="x" type="snumber" description="Coordinata X" />
    <Member name="y" type="snumber" description="Coordinata Y" />
  </Struct>

  ...

</Structs>
```

Figura 105 - Sezione <Structs> nel descrittore di plugin

6.3.4 Definizione dei comandi

La sezione ***<Commands>*** contiene un elenco di elementi *<Command>* ciascuno dei quali definisce uno dei comandi del plugin. Per ogni comando è possibile specificare il nome e una descrizione attraverso l'attributo *name* e l'elemento *<Description>*. Con gli elementi *<Inputs>* e *<Outputs>* si specificano quelli che sono i parametri di input e i parametri di output. Ogni parametro possiede un nome (name), un tipo (type) e una descrizione (description). Il tipo deve essere specificato utilizzando la notazione fully qualified e può essere anche definito in uno dei plugin dipendenti.

La Figura 106 mostra la definizione del comando *ListReaders()* presente nel plugin "pcsc".

```xml
<Commands>

  <Command name="ListReaders">
    <Description>
      This command provides the list of readers.
    </Description>
    <Inputs>
      <Parameter name="context" type="snumber" description="Handle to the resource manager context." />
    </Inputs>
    <Outputs>
      <Parameter name="res" type="pcsc.Error" description="The command result." />
      <Parameter name="readers" type="pcsc.Vstring" description="The vector of reader names" />
    </Outputs>
  </Command>

  ...

</Commands>
```

Figura 106 - Sezione <Commands> nel descrittore di plugin

6.3.5 Il descrittore del plugin math

Nel paragrafo 6.2 è stato descritto il plugin *math*. Il primo passo per realizzare un nuovo plugin è scrivere il relativo descrittore. E’ importante che il nome del file corrisponda esattamente al nome del plugin, in modo tale che la libreria core di SCOTT sia in grado di trovarlo nella fase di caricamento. Nel nostro caso quindi il descrittore è salvato in un file di nome ***math.spd***.

Di seguito è riportato integralmente il contenuto del descrittore del plugin:

```xml
<?xml version="1.0" encoding="utf-8"?>

<Plugin xmlns="http://www.scott.it/spd-schema">

  <Commands>

    <Command name="GetQuadrante">
      <Description>
        Restituisce il quadrante in cui si trova il punto specificato.
      </Description>
      <Inputs>
        <Parameter name="p" type="math.Point" description="Punto da valutare." />
      </Inputs>
      <Outputs>
        <Parameter name="q" type="math.Quadrante"
                   description="Quandrante in sui si trova il punto"/>
      </Outputs>
    </Command>

    <Command name="Filtra">
      <Description>
        Restituisce un vettore con i punti nel quadrante specificato.
      </Description>
      <Inputs>
        <Parameter name="vpi" type="math.VPoint"
                   description="Vettore di punti in ingresso." />
        <Parameter name="q" type="math.Quadrante"
                   description="Quadrante da considerare."/>
      </Inputs>
      <Outputs>
        <Parameter name="vpo" type="math.VPoint"
                   description="Vettore di punti filtrato."/>
      </Outputs>
    </Command>

  </Commands>

  <Types>

    <Enums>
      <Enum name="Quadrante">
        <Entry name="Primo" />
        <Entry name="Secondo" />
        <Entry name="Terzo" />
        <Entry name="Quarto" />
      </Enum>
    </Enums>

    <Vectors>
      <Vector name="VPoint" itemType="math.Point" description="Vettore di Point."/>
    </Vectors>
```

```
    <Structs>
      <Struct name="Point" description="Rappresenta un punto 2D.">
        <Member name="x" type="snumber" description="Coordinata X"/>
        <Member name="y" type="snumber" description="Coordinata Y"/>
      </Struct>
    </Structs>

  </Types>

</Plugin>
```

6.4 Scrivere il codice del nuovo plugin

Dopo aver completato la realizzazione del descrittore di plugin, è necessario iniziare a scrivere codice. In particolare si devono creare tre file, utilizzando opportune convenzioni di naming:

- <NomePlugin>Plugin.h
- <NomePlugin>Plugin.cpp
- entrypoint.cpp

Tutto quello che si deve fare per creare un nuovo plugin è derivare dalla classe astratta ***Plugin***. La convenzione per assegnare un nome a questa classe è far seguire al nome del plugin (con la prima lettera maiuscola) il suffisso Plugin. Nel nostro caso quindi la classe si chiama ***MathPlugin*** e i file d'intestazione e implementazione sono rispettivamente ***MathPlugin.h*** e ***MathPlugin.cpp***.

Il file *entrypoint.cpp* contiene solamente la definizione della funzione *LoadPlugin()* che ha il compito di costruire un'istanza della classe del Plugin e restituirla. Questa funzione è quella che permette a SCOTT di caricare dinamicamente il plugin in fase di esecuzione.

6.4.1 La classe Plugin

Prima di entrare nel merito dell'implementazione, è importante conoscere la classe base *Plugin* in termini di funzionalità rese disponibili.

La Figura 107 mostra i principali membri e metodi definiti nella classe Plugin.

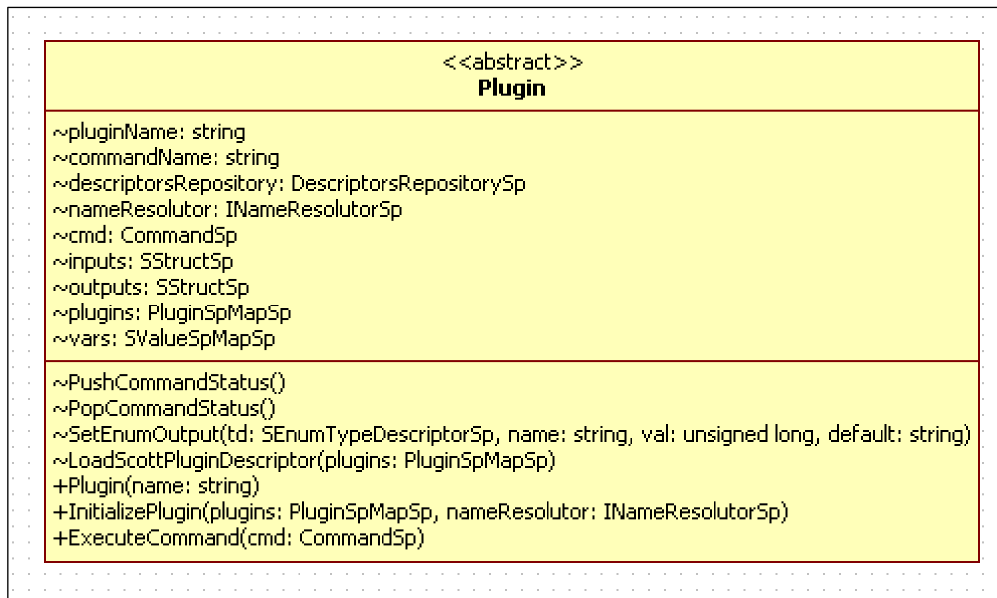


Figura 107 - Classe Plugin

Per quanto riguarda l'interfaccia pubblica, gli unici metodi disponibili sono *InizializePlugin()* ed *ExecuteCommand()*. Il primo metodo è chiamato subito dopo la creazione del plugin ed ha il compito di eseguire operazioni d'inizializzazione. L'implementazione di default chiama il metodo *LoadScottPluginDescriptor()* che si occupa di effettuare il parsing del descrittore di plugin e costruire automaticamente in memoria le istanze di tutti i *TypeDescriptor* e *CommandDescriptor* definiti in esso. Il secondo metodo è invece utilizzato per eseguire uno specifico comando. Prende in ingresso un'istanza di un Command che al termine conterrà i risultati dell'operazione.

La classe *Plugin* presenta un certo numero di membri e metodi protetti che possono essere utilizzati per scrivere il corpo dei comandi del nuovo plugin.

In membri disponibili sono:

- ***pluginName***: contiene il nome del plugin.
- ***commandName***: contiene il nome del comando attualmente in esecuzione all'interno del plugin. Questo valore è impostato all'interno dell'implementazione di base del metodo *ExecuteCommand()*.

- ***descriptorsRepository***: riferimento al repository dei descrittori. Grazie a questo membro è possibile accedere a qualunque descrittore di tipo o di comando. Per maggiori informazioni fare riferimento al paragrafo 5.5.
- ***nameResolutor***: riferimento al risolutore dei nomi. Grazie ad esso è possibile determinare il nome assoluto di un qualsiasi tipo o comando. Per maggiori informazioni fare riferimento al paragrafo 5.6.
- ***cmd***: riferimento al comando attualmente in esecuzione. Per maggiori informazioni sul meccanismo dei comandi fare riferimento al paragrafo 5.4.
- ***inputs***: struttura che contiene i parametri di input del comando attualmente in esecuzione.
- ***outputs***: struttura da utilizzare per impostare i parametri di output del comando attualmente in esecuzione.
- ***plugins***: mappa contenente tutti i plugin caricati.
- **vars**: mappa di variabili utile quando si desidera invocare un comando di un plugin dipendente o un comando differente da quello attualmente in esecuzione.

Il metodo *`SetEnumOutput()`* permette di valorizzare un parametro di output di tipo *SEnum* a partire dal suo valore numerico. Il metodo prende in ingresso il descrittore di tipo, il nome del parametro di output, il valore numerico e un valore da utilizzare nell'eventualità che il valore numerico specificato non sia definito.

I metodi *`PushCommandStatus()`* e *`PopCommandStatus()`* devono essere utilizzati quando si desidera invocare un comando dello stesso plugin all'interno di un altro. Questi metodi utilizzano uno stack per gestire il salvataggio e il ripristino dello stato corrente di un comando. Non è necessario utilizzare questi metodi nel caso si desidera utilizzare un comando implementato in uno dei plugin da cui si dipende.

6.4.2 SCOTT Plugin Creator

Ora che abbiamo capito ciò che la classe `Plugin` può offrirci è giunta l'ora di scrivere la classe concreta con l'implementazione effettiva dei comandi.

Invece di costruire manualmente questa classe è stato realizzato un generatore di codice chiamato **scottpc** (SCOTT Plugin Creator). Questo strumento a linea di comando prende in ingresso il percorso al file che contiene il descrittore di plugin e genera automaticamente i tre file necessari per implementare il nuovo plugin. E' possibile specificare facoltativamente una directory di destinazione, dove saranno salvati i file auto generati. In caso contrario sarà utilizzata quella corrente.

La Figura 108 mostra l'utilizzo di questo strumento per creare l'implementazione del plugin *math*.

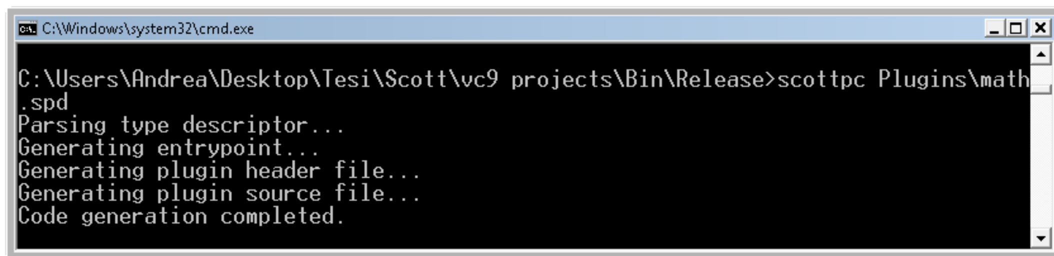


Figura 108 - Utilizzo di SCOTT Plugin Creator.

6.4.3 Analisi dei file autogenerati

I file auto generati da SCOTT Plugin Creator sono:

- entrypoint.cpp
- MathPlugin.h
- MathPlugin.cpp

Il contenuto del file *entrypoint.cpp* non merita altri commenti ed è riportato qui di seguito:

```
#include "MathPlugin.h"

extern "C"
{
	/**
	 *  Create the plugin instance
	 */
	DECLARATION_DLL Plugin* LoadPlugin()
	{
		return new MathPlugin();
	}
};
```

Il file *MathPlugin.h* contiene il seguente codice:

```
#if !defined(_MathPlugin_H)
#define _MathPlugin_H

#include <scott/Plugin.h>
#include <winscard.h>
#include <iostream>

CREATE_ALIAS( MathPlugin );

class DECLARATION_DLL MathPlugin :  public Plugin
{
	// Useful file descriptors references
	SStructTypeDescriptorSp PointTd;
	SEnumTypeDescriptorSp   QuadranteTd;
	SVectorTypeDescriptorSp VPointTd;

	// Enumerators default values
	string QuadranteDefault;

public:

	MathPlugin();

	virtual void InitializePlugin(PluginSpMapSp plugins,
				      INameResolutorSp nameResolutor);

	virtual void ExecuteCommand(CommandSp command);

	// Plugin Command Handler Declarations
	virtual void Math_Filtra();
	virtual void Math_GetQuadrante();

	// Methods to set enumerators
	virtual void SetQuadrante(string outputParameterName, string enumValue);
	virtual void SetQuadrante(string outputParameterName,
					unsigned long enumNumericValue);
};

#endif  // _MathPlugin_H
```

Infine si riporta il contenuto del file *MathPlugin.cpp*:

```
#include "MathPlugin.h"


MathPlugin::MathPlugin() : Plugin("math")
{
	// Set the default enumerator values
	QuadranteDefault = "Primo";
}

void MathPlugin::InitializePlugin( PluginSpMapSp plugins, INameResolutorSp nameResolutor )
{
	Plugin::InitializePlugin(plugins, nameResolutor);
```

```
        // Get useful references to type descriptors
        PointTd = descriptorsRepository->GetStructTypeDescriptor("math.Point");
        QuadranteTd = descriptorsRepository->GetEnumTypeDescriptor("math.Quadrante");
        VPointTd = descriptorsRepository->GetVectorTypeDescriptor("math.VPoint");
}

void MathPlugin::ExecuteCommand(CommandSp cmd)
{
        Plugin::ExecuteCommand(cmd);

        // Call the right command handler
        if ( commandName == "Filtra" )  Math_Filtra();
        else if ( commandName == "GetQuadrante" )       Math_GetQuadrante();
        else throw SPluginException("Error: command not implemented by plugin.");
}

void MathPlugin::Math_Filtra()
{
        // Input preparation
        SVectorSp vpi  = inputs->Get<SVector>("vpi");
        SEnumSp q  = inputs->Get<SEnum>("q");

        // Insert plugin operations here

        // Setting outputs
        SVectorSp vpo = SVector::Create( VPointTd);
        outputs->Set("vpo", vpo);
}

void MathPlugin::Math_GetQuadrante()
{
        // Input preparation
        SStructSp p  = inputs->Get<SStruct>("p");

        // Insert plugin operations here

        // Setting outputs
        SetQuadrante("q", "Primo");
}

/******************************************************/
/*           Enumerator support methods               */
/******************************************************/

void MathPlugin::SetQuadrante( string outputParameterName, string enumValue )
{
        if ( QuadranteTd->IsDefined(enumValue) )
        {
                outputs->Set(outputParameterName, SEnum::Create(QuadranteTd, enumValue) );
        }
        else
        {
                outputs->Set(outputParameterName, SEnum::Create(QuadranteTd,
                                                                QuadranteDefault) );
        }
}
void MathPlugin::SetQuadrante( string outputParameterName, unsigned long enumNumericValue)
{
        string enumValue = QuadranteDefault;

        if ( enumNumericValue == Primo)   enumValue = "Primo";
        if ( enumNumericValue == Secondo) enumValue = "Secondo";
        if ( enumNumericValue == Terzo)   enumValue = "Terzo";
        if ( enumNumericValue == Quarto)  enumValue = "Quarto";

        outputs->Set(outputParameterName, SEnum::Create(QuadranteTd, enumValue) );
}
```

Guardando i sorgenti generati è possibile notare le seguenti cose:

- Per ogni tipo complesso definito dal plugin, viene creata una variabile d'istanza per avere un riferimento diretto ai corrispondenti descrittori. I membri creati (***PointTd***, ***VPointTd*** e ***QuadranteTd***) sono inizializzati all'interno del metodo `InitializePlugin`() utilizzando il repository dei descrittori accessibile tramite il membro protetto *descriptorsRepository*. La convenzione di naming utilizzata è far seguire al nome del tipo il suffisso Td che rappresenta un acronimo di Type Descriptor.
- Per ogni comando definito dal plugin, viene creato un metodo che conterrà la rispettiva implementazione. L'overload del metodo `ExecuteCommand()` sulla base del valore contenuto in *commandName* ha il compito di invocare il metodo corrispondente al comando che si desidera mandare in esecuzione. Nel corpo di ognuno di questi metodi sono prelevati i parametri di input e impostati i parametri di output con valori predefiniti.
- Per ogni tipo enumeratore sono creati dei metodi e dei membri di utilità per semplificare la fase d'impostazione dei risultati. I membri permettono di specificare un valore di default per ogni enumeratore mentre i metodi permettono di impostare una variabile di output di tipo enumeratore passando la costante denominata oppure direttamente il valore numerico. Questi metodi in caso si tenti di impostare un valore non definito assegneranno quello di default.

E' importante sottolineare che *scottpc* rappresenta solamente uno strumento di utilità al fine di aumentare la produttività dello sviluppatore e il suo utilizzo non è strettamente necessario. Inoltre alcune funzioni di utilità potrebbero non servire e in tal caso è buona norma rimuoverle.

Nel caso del plugin *math*, ad esempio, il metodo che permette di impostare il quadrante dal suo valore numerico può essere eliminato anche perché tale metodo ha senso solamente quando le costanti denominate dell'enumeratore sono identiche a macro definite in qualche libreria come nel caso del tipo

pcsc.Error che può assumere come valori proprio le macro definite all'interno del file d'intestazione *winscard.h*.

Tutto quello che deve fare lo sviluppatore di un plugin, è quindi solamente scrivere il corpo di ogni comando. SCOTT Plugin Creator consente di concentrarsi fin da subito sul cuore dell'implementazione del plugin.

6.4.4 Implementazione del corpo dei comandi

A questo punto non resta che vede il codice che implementa i due comandi del plugin *math* preso come esempio.

```
void MathPlugin::Math_GetQuadrante()
{
   // Input preparation
   SStructSp p  = inputs->Get<SStruct>("p");

   // Insert plugin operations here
   unsigned long x = p->Get<SNumber>("x")->Get();
   unsigned long y = p->Get<SNumber>("y")->Get();

   if ( x > 100 || y > 100 )
   {
      throw SPluginException("Il punto p e' al di fuori della regione ammissibile.");
   }

   string quadrante = "Quarto";

   if ( x >= 50 && y >= 50 )      quadrante = "Primo";
   else if ( x <= 50 && y >= 50 ) quadrante = "Secondo";
   else if ( x <= 50 && y <= 50 ) quadrante = "Terzo";

   // Setting outputs
   SetQuadrante("q", quadrante);
}


void MathPlugin::Math_Filtra()
{
   // Input preparation
   SVectorSp vpi  = inputs->Get<SVector>("vpi");
   SEnumSp q  = inputs->Get<SEnum>("q");

   SVectorSp vpo = SVector::Create(VPointTd);

   // Insert plugin operations here
   for(int i=0; i<vpi->GetLength(); ++i)
   {
        // Si ottiene il punto in posizione i
        SStructSp p = vpi->Get<SStruct>(i);
        string quadrante;

        // Chiamata al comando GetQuadrante()
        PushCommandStatus();
        try
        {
           CommandDescriptorSp getQuadranteTd =
                      descriptorsRepository->GetCommandDescriptor("math.GetQuadrante");
           (*vars)["p"] = p;
```

```
            CommandSp getQuadranteCmd = Command::Create(getQuadranteTd, "", vars);
            ExecuteCommand(getQuadranteCmd);

            SStructSp getQuadranteOutputs = getQuadranteCmd->GetOutputs();
            quadrante = getQuadranteOutputs->Get<SEnum>("q")->Get();
        }
        catch(SPluginException ex)
        {
            PopCommandStatus();
            outputs->Set("vpo", vpo);

            stringstream os;
            os << "Il punto vpi[" << i <<
                  "] e' al di fuori della regione ammissibile.";
            throw SPluginException(os.str());
        }

        PopCommandStatus();

        // Se il quadrante è quello richiesto si aggiunge all’output
        if ( q->Get() == quadrante )
        {
            vpo->AddItem(p);
        }
    }

    // Setting outputs
    outputs->Set("vpo", vpo);
}
```

E’ importante evidenziare alcuni aspetti:

- Per manipolare i parametri di input e i parametri di output si utilizza il modello a oggetti del sistema di tipi di SCOTT ampiamente discusso nel paragrafo 5.3.
- Per segnalare un errore all’interno di un plugin si deve sollevare un’eccezione di tipo ***SPluginException*** con un espressivo messaggio di errore. Nell’esempio questa eccezione è sollevata per segnalare la presenza tra gli input di un punto al di fuori della regione ammissibile.
- Il comando *Filtra()* compie un ciclo tra i punti del vettore di ingresso e su ognuno di essi invoca il comando *GetQuadrante()*. Sulla base del risultato ottenuto il punto considerato sarà aggiunto nell’output. La chiamata al metodo innestato *GetQuadrante()* è racchiusa tra la coppia di metodi *PushCommandStatus()* e *PopCommandStatus()*. Questo è necessario perché il comando *GetQuadrante()* fa parte dello stesso plugin in cui è definito *Filtra()*.

6.5 Utilizzare il nuovo plugin

Terminata la fase di scrittura del codice del plugin si può effettuare la compilazione del codice all'interno di una libreria dinamica. Utilizzando le istruzioni riportate nel paragrafo 6.1 è possibile installare il nuovo plugin e renderlo quindi accessibile all'interno della SCOTT Shell.

La Figura 109 mostra un esempio di utilizzo del nuovo plugin.

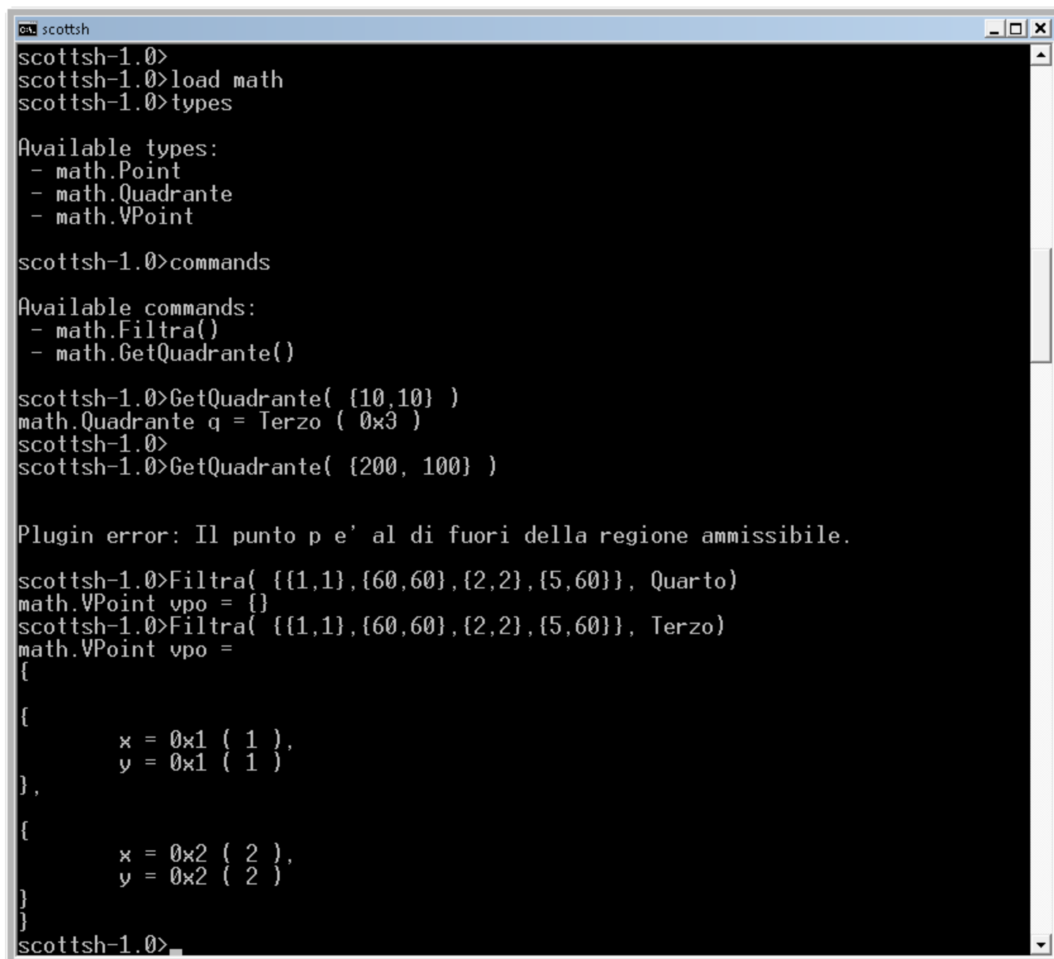


Figura 109 - Utilizzo del plugin 'math'

7. Conclusioni e sviluppi futuri

Nonostante la continua e crescente diffusione delle smartcard nei principali ambiti della società, come dimostrano le statistiche realizzate da Eurosmart, lo sviluppo di applicazioni basate su questi dispositivi richiede ancora oggi figure professionali veramente qualificate. La complessità di realizzare applicazioni è legata principalmente a forti problemi d'interoperabilità tra le schede e alla mancanza di strumenti adeguati per compiere le operazioni di testing e debugging. Anche se esistono diversi standard, al fine di regolare e uniformare il comportamento delle schede in vari settori, i produttori hanno tutto l'interesse a deviare da essi per realizzare dispositivi con funzionalità proprietarie in grado di distinguerli dalla concorrenza. Gli interessi commerciali quindi influiscono negativamente anche per quanto riguarda il supporto che può essere fornito durante le attività di realizzazione di soluzioni. Gli strumenti commerciali disponibili, oltre ad essere notevolmente costosi, hanno il grosso limite di supportare esclusivamente un numero ridotto di dispositivi. Negli ultimi anni, all'interno del mondo open-source, sono stati realizzati un numero elevato di strumenti per venire incontro alla necessità di semplificare le attività di testing delle proprie soluzioni. Questi strumenti, tuttavia, anche se offrono i tipici vantaggi del software libero, sono in grado di eseguire un numero limitato di funzionalità e supportano un numero ridotto di schede. Sviluppare un'applicazione interoperabile richiede quindi uno sforzo notevole, poiché si è costretti a imparare un certo numero di strumenti differenti per risolvere sostanzialmente un unico problema.

Da queste basi nasce l'idea di realizzare SCOTT (Smart Card Open Test Toolkit). L'obiettivo ambizioso di SCOTT è di diventare lo strumento di riferimento per supportare il testing di soluzioni basate su smartcard.

I principali requisiti che hanno guidato la fase di progettazione sono:

- Deve essere uno strumento completamente modulare ed espandibile.
- Deve permettere di interagire con qualsiasi dispositivo di smartcard.
- Deve offrire un'interfaccia uniforme all'esecuzione di comandi.
- Deve integrare almeno un interprete a riga di comando (shell) con caratteristiche tali da essere facilmente utilizzabile in procedure di test automatiche.
- Deve essere liberamente utilizzabile per favorire l'interoperabilità.
- Deve essere utilizzabile almeno sulle piattaforme Windows e Linux.

SCOTT è stato realizzato nella sua totalità e soddisfa pienamente tutti questi requisiti. I maggiori sforzi fatti vanno nella direzione di semplificare il più possibile le possibilità di espansione di questa piattaforma attraverso opportuni plugin. Creare un plugin per SCOTT è reso banale grazie all'introduzione di un documento xml, il descrittore di plugin, per definire tipi e comandi implementati al suo interno e di uno strumento, chiamato SCOTT Plugin Creator, in grado di generare gran parte del codice del plugin. Nell'ambito di questa tesi a titolo dimostrativo sono stati realizzati tre plugin. Il plugin "pcsc" implementa tutte le funzioni dello standard PC/SC e permette di interagire con le smartcard attraverso un'interfaccia di basso livello. Il plugin "iso7816-4" realizza alcune funzioni dello standard ISO7816-4 necessarie per manipolare i file presenti sulle schede compatibili. Infine il plugin "cryptoflex8" implementa gran parte dei comandi presenti sulla smartcard Cryptoflex 8K che è stata utilizzata durante tutto il mio lavoro.

Gli obiettivi alla base di questo lavoro sono stati raggiunti e la piattaforma è completamente operativa. Tuttavia, affinché SCOTT possa affermarsi, è necessario procedere nello sviluppo di un numero considerevole di plugin. La validità del progetto e soprattutto il fatto di essere uno strumento rilasciato con una licenza open-source, fa ben sperare che la comunità di sviluppatori o altri laureandi si occuperanno della creazione di nuovi plugin.

Per quanto riguarda l'immediato futuro del progetto, quindi, è possibile fare le seguenti previsioni:

- Saranno creati nuovi plugin per supportare un numero sempre maggiore di dispositivi e di standard.
- Gli strumenti open-source attuali saranno integrati all'interno di SCOTT mediante la realizzazione di opportuni plugin.
- SCOTT sarà utilizzabile su un numero maggiore di sistemi operativi oltre a Windows e Linux.
- Saranno aggiunte nuove funzionalità alla shell, in particolare per un suo utilizzo in modalità interattiva.
- Sarà creato uno strumento grafico, da affiancare alla shell, per un utilizzo più intuitivo delle funzionalità offerte da SCOTT.

Questo processo porterà nel medio e lungo termine a rendere SCOTT l'unico e il più valido strumento di riferimento per interagire con le smartcard e supportare il testing delle applicazioni basate su di esse.

Appendice A - La smartcard Cryptoflex 8K

La Cryptoflex 8K è una smart card realizzata da Schlumberger progettata per lo sviluppo di applicazioni di sicurezza.

Nell'ambito della crittografia la scheda presenta le seguenti funzionalità:

- Autenticazione con DES
- Generazione della firma per DES e RSA
- Generazione chiavi per RSA

Il sistema operativo gestisce differenti risorse della scheda come la memoria, la CPU offrendo forti garanzie di sicurezza. Esso offre:

- Un'architettura di file sicura
- Un'interfaccia di comunicazione (basata sul protocollo standard T=0)
- Un insieme di comandi basati sullo standard ISO7816-3,4

La Tabella 20 mostra gli identificatori dei file riservati.

File ID	File Name	File Type
0000	CHV1 (user PIN)	Transparent EF
0001	Internal key	Transparent EF
0002	Card serial number	Transparent EF
0005	Reserved for internal use	—
0011	External key	Transparent EF
0012	Private key	Transparent EF
0015	Reserved for internal use	—
0100	CHV2 (admin PIN)	Transparent EF
1012	Public key	Transparent EF
2F01	ATR	Transparent EF
3F00	Master	Root DF
3F11	Reserved for internal use	—
3FFF	Reserved for internal use	—
FFFF	Reserved for internal use	—

Tabella 20 - Identificatori dei file riservati

Un elenco completo dei comandi disponibili è riportato nella Tabella 21.

Command	Cla	Com	P1	P2	P3
Change CHV	F0	24	00	CHV num	10
Create File	F0	E0	init type	num recs	$lgth + x$
Create Record	C0	E2	00	00	$lgth + x$
Decrease	F0	30	00	00	$03 + x$
Delete File	F0	E4	00	00	$02 + x$
Dir Next	F0	A8	00	00	$(01 \leq lgth \leq 0F) + x$
External Authenticate	C0	82	00	00	07
Get Challenge	C0	84	00	00	$lgth$
Get Response	C0	C0	00	00	$lgth$
Increase	F0	32	00	00	$03 + x$
Internal Authenticate	C0	88	00	key num	08
Invalidate	F0	04	00	00	$00 + x$
Read Binary	C0	B0	offset MSB	offset LSB	$lgth$
Read Record	C0	B2	rec num	mode	$lgth$
Rehabilitate	F0	44	00	00	$00 + x$
RSA Key Generate	F0	46	00	40/60/80	04
RSA Signature (Int Auth)	C0	88	00	key num	40/60/80
Seek	F0	A2	offset	mode	$lgth$
Select	C0	A4	00	00	02
SHA-1 Intermediate	14	40	00	00	40
SHA-1 Last	04	40	00	00	$lgth$
Unblock CHV	F0	2C	00	CHV num	10
Update Binary	C0	D6	offset MSB	offset LSB	$lgth + x$
Update Binary Enciphered	C0	DE	offset MSB	offset LSB	$lgth$
Update Record	C0	DC	rec num	mode	$lgth + x$
Verify CHV	C0	20	00	CHV num	08
Verify Key	F0	2A	00	key num	$lgth$

Tabella 21 - Comandi della scheda Cryptoflex 8K

La Tabella 22 presenta le sedici possibili Access Conditions (ACs) che possono essere applicate sia a file (EF) che alle directory (DF).

Per maggiori informazioni consultare il manuale della scheda.

Hex Value	Access Condition
0	ALW
1	CHV1
2	CHV2
3	PRO
4	AUT
5	RFU
6	CHV1 and PRO
7	CHV2 and PRO

Hex Value	Access Condition
8	CHV1 and AUT
9	CHV2 and AUT
A	RFU
B	RFU
C	RFU
D	RFU
E	RFU
F	NEV

Tabella 22 - Access Condition della scheda Cryptoflex 8K

Tabella degli acronimi

Acronimo	Significato
SCOTT	Smart Card Open Test Toolkit
SPD	SCOTT Plugin Creator
PC/SC	PC / Smart Card
ICC	Integrated Chip Card
IFD	Interface Device
GSM	Global System for Mobile Communications
UMTS	Universal Mobile Telecommunication System
RAM	Random Access Memory
ROM	Read Only Memory
EEPROM	Electrically Erasable and Programmable Read Only Memory
DES	Data Encryption Standard
TDES	Triple DES
AES	Advanced Encryption Standard
RSA	Rivest Shamir Adleman
API	Application Program Interface
SIM	Subscriber Identity Module
SMS	Short Message Service
CNS	Carta Nazionale dei Servizi
CIE	Carta di Identità Elettronica
PIN	Personal Identification Number
ISO	International Standard Organization
USB	Universal Serial Bus
PCMCIA	Peripheral Component Microchannel Interconnect Architecture
MSDN	Microsoft Developer Network
MUSCLE	Movement for the Use of Smart Cards in a Linux Environment
EF	Elementary File
DF	Dedicated File
MF	Master File

APDU	Application Protocol Data Unit
CHV	Card Holder Verification
AC	Access Condition
UML	Unified Modeling Language
SDK	Software Development Kit
SVN	Subversion
STL	Standard Template Library
XML	eXtensible Markup Language
GNU	GNU is Not Unix
GPL	General Public License
HTML	HyperText Markup Language
XSD	XML Schema Definition

Bibliografia

.NET Smart Card Framework.
http://www.gemalto.com/products/dotnet_card/dotnet_framework.html.

International Standard Organization. ISO/IEC 7816-8: Identification cards -- Integrated circuit cards -- Part 8: Commands for security operations. 2004.

International Standard Organization. ISO/IEC 7816-3: Identification cards - Integrated circuit cards - Part 3: Cards with contacts - Electrical interface and transmission protocols. 2006.

International Standard Organization. ISO/IEC 7816-4: Identification cards - Integrated circuit cards - Part 4: Organization, security and commands for interchange. 2005.

International Standard Organization. ISO/IEC 7816-9: Identification cards - Integrated circuit cards - Part 9: Commands for card management. 2004.

Computer Science Software Engineering (CSSE) Journal, Vol. 20, No. 6 - An open middleware for smart-cards. November 2005.
http://retis.sssup.it/~tommaso/publications/CSSE-2005.pdf.

Allegra, Roberto. *Lavorare con C++.* Edizioni Master, 2006.

Amedeo, Enrico. *UML - Unified Modeling Language.* Apogeo, 2007.

Beri, Marco. *Espressioni regolari.* Apogeo, 2007.

Boost C++ Libraries. http://www.boost.org/.

Canducci, Massimo. *XML.* Apogeo, 2005.

Chirico, Ugo. *Programmazione delle SmartCard.* Cryptoware, 2009.

CNS, Carta Nazionale dei Servizi.
http://www.progettocns.it/cittadino/progettoCns.aspx.

Deitel, Deitel &. *C++ Tecniche avanzate di programmazione.* Apogeo, 2001.

Eurosmart - The Voice of the Smart Security Industry. http://www.eurosmart.com/index.php/home.html.

Gamma, Helm, Johnson, Vlissides. *Design Patterns - Elementi per il riuso di software a oggetti.* Addison-Wesley, 2002.

Global Platform. http://www.globalplatform.org/.

GPShell. http://sourceforge.net/projects/globalplatform/files/.

Graphical program to manage a MuscleCard smartcard. http://packages.ubuntu.com/hardy/misc/xcardii.

ISO 7816-1 - Physical Charcteristics of Integrated Circuit Cards. http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816-1.aspx.

ISO 7816-2 - Dimensions and Location of the Contacts. http://www.cardwerk.com/smartcards/smartcard_standard_ISO7816-2.aspx.

Java Card Technology. http://java.sun.com/javacard/.

Keith Mayes, Konstantinos Markantonakis. *Smart Cards, Tokens, Security and Applications.* Springer, 2008.

Kernighan, Brian W. *Il linguaggio C - Principi di programmazione e manuale di riferimento.* Pearson, 2004.

Logan, Syd. *Cross-Platform Development in C++: Building Mac OS X, Linux, and Windows Applications.* Addison-Wesley Professional.

McConnell, Steve. *Ingegneria del codice - Manuale pratico per la costruzione di software completo.* Mondadori Informatica, 2005.

Muscle Card - Movement for the use of the smart cards in a linux environment. http://www.musclecard.com/.

Nicholas A. Solter, Scott J. Kleper. *Professional C++.* Wrox, 2005.

OpenSC Tools. http://www.opensc-project.org/opensc/wiki/Tools.

PC/SC Workgroup. http://www.pcscworkgroup.com/.

pcsc-lite SCard API. http://pcsclite.alioth.debian.org/pcsc-lite/.

Petersen, Richard. *Linux - La Guida Completa.* Mc Graw Hill, 2001.

Pialorsi, Paolo. *Programmare con XML.* Mondadori Informatica, 2004.

Rankl, Wolfgang. *Smart Card Applications - Design models for using and programming smart cards.* Wiley, 2007.

Readline library. http://tiswww.case.edu/php/chet/readline/readline.html.

Readline per Windows. http://gnuwin32.sourceforge.net/packages/readline.htm.

Smart Card. http://it.wikipedia.org/wiki/Smart_card.

Smart Card Alliance. http://www.smartcardalliance.org/.

Smart Card Resource Manager API. http://msdn.microsoft.com/en-us/library/aa380149%28VS.85%29.aspx.

Smart Sign Project. 2004. http://smartsign.sourceforge.net/.

String.Format in MSDN. http://msdn.microsoft.com/it-it/library/b1csw23d%28VS.80%29.aspx.

Stutz, Michael. *Linux - Guida Pratica.* Mondadori Informatica, 2004.

"Tesi di dottorato di Tommaso Cucinotta: Issues in authentication by means of smart card devices." 2004. http://retis.sssup.it/~tommaso/Cucinotta-PhD-Thesis.pdf.

"Tesi di Laurea di Tommaso Cucinotta: Progettazione e realizzazione di un sistema per firma digitale e autenticazione basato su smartcard." 1999. http://retis.sssup.it/~tommaso/Cucinotta-TesiDiLaurea.pdf.

Tinyxml. 2009. http://www.grinninglizard.com/tinyxml/.

Ugo's smart card and emulator. http://www.ugosweb.com/scemuit.aspx.

Ward, Brian. *Usare Linux - Guida avanzata.* Mondadori Informatica, 2005.

Wolfgang Rankl, Wolfgang Effing. *Smart Card Handbook.* Wiley.