Sample Development on Java Smart-Card Electronic Wallet Application

Cristian TOMA

Faculty of Cybernetics, Statistics and Economic Informatics
Department of IT&C Technologies
Academy of Economic Studies Bucharest, Romania
cristian.toma@ie.ase.ro

***Abstract**:* In this paper, are highlighted concepts as: complete Java card application, life cycle of an applet, and a practical electronic wallet sample implemented in Java card technology. As a practical approach it would be interesting building applets for ID, Driving License, Health-Insurance smart cards, for encrypt and digitally sign documents, for E-Commerce and for accessing critical resources in government and military field. The end of this article it is presented a java card electronic wallet application.

***Keywords**:* smart card, Java Card application, ISO 7816, JCRMI.

1. Introduction

The cards are classified in magnetic strip cards and smart cards. The smart ones are divided after many features. For instance, if we consider the way how they communicate with the card reader device, those are contact-less – the communication between card reader and smart card is made through radio wave, with contact – the smart card make a physical contact with the smart card, or combined.
Concerning the type of integrated circuit that a smart card could have, the smart cards are classifying in:

- *Cards with microprocessor chip*, short term chip cards, contains a microprocessor which is used for computations. Besides this microprocessor with 8, 16 or 32 bits register, the card could contain one or more memory chips which is using for read-only memory and for random access memory – RAM. This features offer to a card almost the power of a desktop computer. This type of cards are used in different informatics systems like banking credit cards, cards for access control in institutions, SIMs – Secure Identification Module – for mobile phones and cards for accessing digital TV networks.
- *Cards with memory chip*, contains different data but can not compute the store data because the card don't have a microprocessor. They are fully depended by the host application.

In our days most of the specialists agree on the idea, that a card is smart only if it can compute, only if it has a microprocessor or a microcontroller. Keeping on this approach, the difference between a smart card and a card only with memory chips or magnetic strips, is that the last one only can store data and can not compute the data. The informatics systems which are interacting with smart cards have an advantage because the access to the different data bases and the time of transactions could be considerably minimized. More than that, some smart cards contain non-volatile memories which provide a great advantage regarding the development of secure systems and applications, because in those memories they can store sensitive information like digital certificates, symmetric and asymmetric private keys. In order to improve the speed of computations, this kind of cards have also specialized cryptographic coprocessors. The coprocessors execute complicate cryptographic algorithms like RSA, AES-Rijndael, 3DES

or algorithms based on elliptic curves. In the following sections will be implemented step by step an electronic wallet implemented on a smart card as a Java card applet.

2. Complete applications for Java smart cards

A Java card application is an applet which is running in smart card. But often the applet needs to interact with different systems and applications. That's why in specialty literature a complete application for Java smart card is composed from the java applet which is running on smart card, a host application and back-end application systems which provide to the end-user a service.

In figure 1 is depicted a complete Java card application:

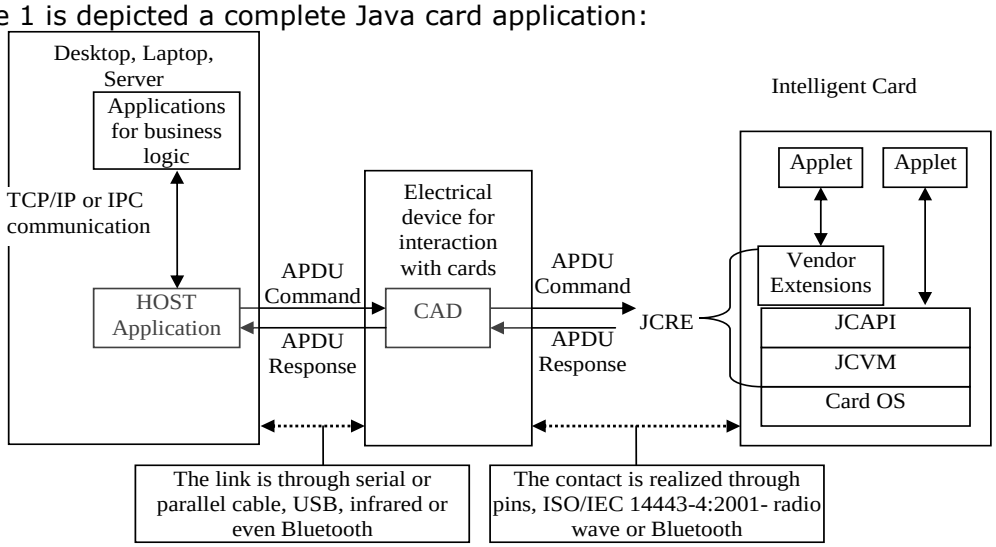

Fig. 1 Complete Java smart card application

With more details, informatics systems that use smart cards will have the following items from point of view of a complete Java card application:

- Back-end Applications – the ones that implement the business logic and connects to the data bases and web services;
- Host or off-card Applications – the ones that communicate with the card reader, they are the interface between Back-end applications and the card reader. These applications run on the desktop which is connected with the card reader. Also they can run on specialized terminal like ATM-Automatic Teller Machine or can run on a mobile phone which is used as a smart card reader;
- Card Reader Applications – these are running in the card reader and are responsible for the accomplishment and the coordination of the interaction with card's applications. The physical equipment, the card reader, plus with the applications that are running on it, is called CAD – Card Acceptance Device. The CAD is responsible how is realized the physical connection with the card, through electrical contact or radio wave. Also the card is responsible for providing energy for the card. The CAD takes APDU-Application Protocol Data Unit- commands – standard strings of bytes – from host applications and are forwarded to the smart cards;

ISSN 2067 – 4074

- Smart cards' Applications – in Java Card platform can be in the same time many applications, applets. The applets are run in JCRE – Java Card Runtime Environment.

There are three models which can be used in order to realize the communication between host application and Java applet. First model is quite simple and is supposing to send and receive template strings of bytes in typical format – *Message-Passing Model*. The second model is *Java Card Remote Method Invocation – JCRMI*, which is a set of classes and procedures likely with the ones from J2SE – Java 2 Standard Edition RMI, but basically this model use the first model. The third model for the communication between host and the applet from the card, is SATSA – Security and Trust Services API. SATSA defined in JSR 177, provide to the developers to use whatever model as base – Message-Passing Model or JCRMI, but is a more abstract API based on GFC – Generic Connection Framework API. So, most of developers use the first two models in order to develop smart cards complete applications.

2. 1 Message-Passing Model

This is the reference model and represents the base for the other two existent models, JCRMI and SATSA for developers. The communication between host application and the applets from smart card suppose to transmit some APDU – Application Protocol Data Unit from host to CAD – Card Acceptance Device, and then the same bytes strings are sent from the CAD to the card applet. The applet receive those bytes strings, is parsing the bytes and then will send back following the reverse path: Applet-CAD-Host. An APDU is composed from standard bytes blocks conform ISO/IEC 7816-3 and 7816-4. Respecting the standards the applet receives directly from CAD, *APDU Commands* and sends back to CAD, *APDU Responses*. The communication between the card reader and the card is physically realized through data link protocol. This protocol is likely data link level protocol from protocol stack ISO/OSI. The link protocol, defined in ISO/IEC7816-4, has four alternatives: T=0 – byte oriented, T=1 – bytes arrays oriented, T=USB – oriented Universal Serial Bus or T=RF – radio wave oriented, Radio Frequencies. The classes from Java Card API and JCRE specifications embed the physical details for APDU communication protocol.

2.1.1 APDU Commands

The general template for an ADPU Command is depicted in figure 2:

APDU Command						
CLA	**INS**	**P1**	**P2**	**Lc**	**Data Field**	**Le**
Header-mandatory				**Body-optional**		

Fig. 2 General structure for an APDU Command

There are other four specific structures for an APDU Command, but these structures are used only in data link protocol T=0.

The explication for the fields from the APDU command is the following:

- CLA – is one byte – 2 hexadecimal digits, and has different predefined values conform standard ISO7816. For instance, between the value 0x00 and 0x19 are values for accessing file system and security operations, from 0x20 to 0x7F are reserved for future using, and from 0x80 to 0x99 can be used for applets' specific instructions implemented by developers but between 0xB0

and 0xCF are specific instructions for all applets and not for a particular one. As matter of fact the most used value for this field is 0x80;

- INS – is one byte, and the standard defines a specific instruction in the field CLA. For instance, when CLA has the value between 0x00 and 0x09, but INS has the value 0xDC – means card's records update. In personal applications which are installed on the card, the field INS could have predefined values established by developers but according with the standard. For example, the developer chooses for this field the value 0x20 for checking sold amount from card if and only if the CLA field is 0x80;
- P1 – this represents the first parameter for a instruction and has one byte. This field is used when the developers want to send some parameters to the applet or want to qualify the INS field;
- P2 – this is the second parameter for an instruction and has one byte. Is used for the same scope like P1;
- Lc – has one byte, is optional and represents the bytes length for the field Data Field;
- Data Field – is not fixed and has a bytes' length equal with the value from the field's value Lc. In this field are stored data and parameters which are send from host application to applet;
- Le – stores the maxim number of bytes that should have Data Field from APDU Response (the number of bytes from response could be any value from the range 0 and the value from this field).

Practically a host application sends to the CAD but the CAD sends to the applet the same APDU commands with structures and values which respect the standards.

2.1.2 APDU Responses

The structure for an APDU Response is simple and is depicted in the figure 3:

APDU Response		
Data Field	SW1	SW2
Body-optional	Trailer-mandatory	

Fig. 3 Structure for an APDU Response

The fields' explication for APDU Response is the following:

- Data Field – has variable length which is determined by the value of the byte field Le from the APDU Command;
- SW1 – has one byte and represent the status word 1;
- SW2 – has one byte and represent the status word 2.

The fields SW1 and SW2 are parsed and interpreted together, but a communication process is called *complete* if there were no problems (SW1=0x61 and SW2=0x90 or any-0xnn) or if there were only warnings (SW1=0x62 or SW1=0x63 and SW2 contains the warning code). A communication process is called *failed* if there were execution errors (SW1=0x64 or SW1=0x65 and SW2 has the error code for execution) or checking errors (SW1=from 0x67 to 0x6F and SW2 has the code for checking error).

In message passing model there is all the time a selected applet in the card and when is receiving an APDU Command (if this command's request is not for switching the applets execution) JCRE calls automatically the applet method process() and passing the bytes values as argument for the method. In the method process(), the applet must parse and process the command and must generate an APDU Response. After that the methods return the control to the JCRE. In the section regarding the Life cycle for a Java applet

ISSN 2067 – 4074

are discuss all the steps which are made by an applet in order to communicate with the card reader conform to the protocol standard.

2.2 JCRMI Model

The second model represents a distributed object oriented model subset – J2SE RMI. In this model, even if behind the scene, physically, the communication is realized through APDU, the developer should not be aware by protocol details. The developer is focusing only on the problem and is not interested in the idea that on level below this model is using the message passing model.

In the JSE RMI model – Java Standard Edition Remote Method Invocation, a client application obtains references using an interface to the Java objects which are running in others JVM-Java Virtual Machine eventually on the different computers in network. Once the reference-pointer is obtained, the application forces the execution of an object's specific method which is running in other JVM. If a card applet knows to add two numbers, the developer only sends the two numbers as method's parameters to the card and the method from card is executing the add operation and return the results to the developer's application. So, the card is not store equipment, it is also an important tool for computing.

For instance, if an applet object implement the X(int p1) method then a host application which is running on a desktop computer obtains the reference to the object that implement method X. Using the reference the application is calling the method with a proper parameter, X(5). The parameter and others details are sending through APDU and the received APDU Responses are embedded respecting RMI technology.

3. Issues in the development and life cycle of a Java card applet

For an Java card applet-application is involved a series of elements and concepts, of which the most important are: Java Card Virtual Machine – JCVM, Java Card Advanced Programming Interface – JCAPI, Java Card Runtime Environment – JCRE, the life cycle of JCVM and of the Java card applet, the Java Card sessions, logical channels, isolation and partition of the applet objects, memory and memory objects management and persistent transactions.

For the Java Card platform, ***JCVM*** is divided in two parts. One part is external to the physical card and is used as a developing tool. This part converts, uploads and verifies the Java classes that have been compiled with a normal Java compiler. The finality of this external part is that from a class normally compiled Java-byte code, results a binary execution CAP file – Converted Applet, which will be executed by the JCVM on the card. The other part of JCVM resides on the card and is used to interpret the binary code produced by the first part and for the management of objects and classes. The card part of the JCVM has, analogically, approximately the same use as a JVM – Java Virtual Machine for a desktop computer. Of course JCVM has a series of limitations of syntax language as well as of organization structure. For instance, as limitations of syntax-language it is possible to mention the lack of support for some key words (native, synchronized, transient, volatile, strictfp), for some types (double, float, long) and for some classes, interfaces and exceptions (the majority of the classes, interfaces and exceptions from the packages java.io, java.lang, java.util). As limitations of organization structure language, it is mentioned the fact that a package cannot contain more than 255 classes and a class cannot directly or indirectly implement more than 15 interfaces.

More details are presented in the specifications of the virtual machine for Java Card platform [9].

JCAPI defines a subset of classes, interfaces and exceptions from Java 2 Standard Edition. There is no support for the multi-string execution programming, for important classes such as String, Boolean, Integer or BigInteger. The packages that work with by JCAPI are the following: *java.io*, *java.lang*, *java.rmi*, *javacard.framework*, *javacard.framework.service*, *javacard.security*, *javacardx.crypto*, *javacardx.rmi*.

It is mentioned as follows:

- From the package *java.io* there is kept only the *IOException* class to complete the hierarchy of classes concerning the exceptions from Remote Method Invocation;
- From the package *java.lang* there is kept the simplified version of the classes *Exception*, *Object* şi *Throwable* and it is introduced the class CardException;
- From the package *java.rmi* there is kept the *Remote* interface and the *RemoteException* class;
- There is introduced the package *javacard.framework* that contains interfaces (*ISO7816* – contains constants used by the standard, *MultiSelectable* – used by the applets that accept competitive selection, *PIN* – represents Personal Identification Number, *Shareable* – used for objects that can be partitioned on the card applets), classes (*AID* – identifies according to ISO7816-5 the unique identifications for Application Identifier applets, *APDU* – embeds Application Protocol Data Unit, that have been presented in the above paragraphs, as in ISO7816-4, *Applet* – abstract class that defines the application-applet which resides on the card, *JCSystem* – contains specific methods for controlling the life cycle of an applet, *OwnerPIN* – an implementation of the PIN interface, *Util* – contains methods such as arrayCompare() and arrayCopy() for editing the octet strings from the smart card memory) and exceptions (*APDUException*, *ISOException*, *SystemException*, *TransactionException*, *CardException*) intensively used for developing the applets for the Java Card platform;
- It is introduced the package *javacard.framework.service* that contains interfaces (*Service* – it is an interface for the basic service used by the applet for processing the APDU Commands and Answers by methods such as processCommand(), processDataIn() and processDataOut(), *RemoteService* – interface for the remote access to the card services by RMI, *SecurityService* – it extends the Service interface and provides methods such as isAuthenticated(), isChannelSecure() or isCommandSecure() for verifying the current security state), classes (*BasicService* – the pre-defined implementation of the Service interface and provides helping methods for collaborating with different services and for APDUs editing, *Dispatcher* – used when the same APDU Command is intended to be edited by different services) and exceptions (*ServiceException*) for the different services management;
- It is introduced the package *javacard.security* that contains interfaces (*Key*, *PrivateKey*, *PublicKey*, *SecretKey* and sub-interfaces specialized in algorithms such as *AESKey*, *DESKey*, *DSAKey*, *DSAPrivateKey*, *DSAPublicKey*, *ECKey*, *ECPrivateKey*, *ECPublicKey*, *RSAPrivateCrtKey*, *RSAPrivateKey*, *RSAPublicKey*), classes (*Checksum* – abstact class for algorithms used for the cyclic check of errors, *KeyAgreement* – normal class for algorithms based on the exchange of keys of Diffie-Helman type, *KeyBuilder* – assures the way of key creation, *Keypair* – a container, such as a vector, that holds the pairs of private and public keys, *MessageDigest* – basic class for algorithms of hash type, *RandomData* – basic class for random numbers, *Signature* – abstract class for electronic signature) and exceptions (*CryptoException*) for different

ISSN 2067 – 4074

cryptographic algorithms with public and private keys, digital signatures, hash functions and for the cyclic verification of redundancy (CRC);
- It is introduced two extension packages *javacardx.crypto* and *javacardx.rmi*.

For a better documentation it is recommended the developer's and user's guides [12], [13] and the specifications Java Card API [11].

JCRE is the connection component, the interface, between the native operating system of the smart card and the applet for the Java Card platform**,** as in Figure 3. JCRE assures for the applets the access to the services of the card operating system. JCRE is composed of JCVM, JCAPI and the extensions specific for the card producers. For a complete documentation concerning JCRE and the actions it is responsible for: the life cycle of JCVM, the persistency and partition of the objects in the card memory, the allocation of logical channels, the selection and isolation of applets, refers to the JCRE specifications [10].

The life cycle of JCVM and of a Java Card applet should be understood by any developer of such applications. The life duration of JCVM is the same with that of the card. If the power supply of the card stops, the entire content of JCVM is saved in the persistent memory, non-volatile. Everything in the internal memory-RAM-volatile of the card at the moment of the interruption of power is lost. Moreover, the objects created in the Java Card platform are non-volatile, and if it is sometime intended that an octet string should be in the volatile memory because, for instance, it holds only temporary data, we should use the method makeTransientByteArray() of the class *javacard.framework.JCSystem*.
The methods by which the applet life cycle is realized and that the applet should implement are presented in figure 4:

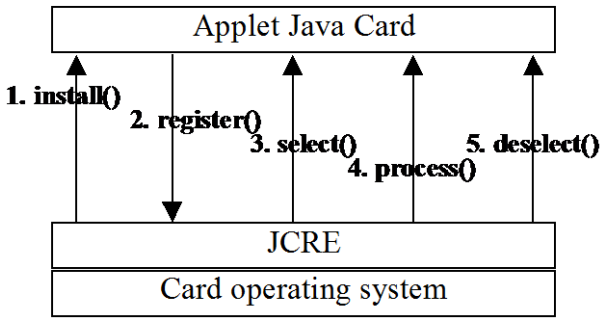

Fig. 4. Methods to be implemented by an applet to execute the complete life cycle

Each applet is uniquely identified by an octet string – between 5 and 16, as defined in ISO7816-5. The octet string is named AID – Application ID in the standard. As well, each applet should extend the abstract Applet class and implement the methods – install(), register(), select(), process(), deselect(), that represent the applet life cycle.

The applet life cycle starts immediately after it is downloaded into the card and JCRE forces the execution of the static method Applet.install() and at its turn the applet is registered in JCRE by calling the static method Applet.register(). After the applet is installed and registered, it exists on the card as „not selected", the equivalent denomination being inactive applet. An applet is activated to process APDU Commands only after the host type application sends to JCRE by CAD an APDU command of type SELECT or MANAGE CHANNEL. JCRE complies and notifies the concerned applet by forcing the execution of select() method which the applet implements. After the selection is done, all the APDU commands received from the host type application by CAD of JCRE

are sent to the applet by forcing calling the method process() implemented by the applet. The life cycle of the applet ends when the host application intends to select another applet for processing the APDU commands, moment when JCRE notifies the applet by forcing the execution of deselect() method of the applet.

In what concerns the ***Java Card sessions*** and the ***logical channels,*** these concepts are explained in the specifications of JCRE [10]. Starting with Java Card 2.2, a card may have up to 16 sessions simultaneous opened, if a session occupies just one logical channel. In what concerns the specifications, a session is the time period since the card is power supplied up to when it is disconnected from the power supply, time period when it exchanges APDU commands and answers with the card reader. The host type application sends an APDU command in sequence to JCRE, but according to each value from the one octet field-CLA, JCRE sends it to one applet or another without selecting and deselecting the applet each time. At its turn, an applet may receive several APDU commands pseudo-simultaneously because it may be designed to be multi-selectable meaning that it will implement the methods of the interface *javacard.framework.MultiSelectable*. Practically, this means that if there are two different APDU commands, each one on a different logical channel, an applet may interpret both of them, or, as well, two different applets may separately process a command of the two.

Isolation and partition of the applet objects is an important concept because on a card can co-habitat several applets produced by different developers. The main idea is that the objects created in the memory that belong to the same package are placed in a neighboring memory zone that is separated of another memory zone by a firewall. In the same memory zone separated by the firewall – in the same context according to the specific literature – con not co-exist objects from different packages. This has major implications for the applet development, because an object in the memory and belongs to a package cannot access the methods of a different object from another package, even if the methods are public. The access is made by JCRE. For instance, the object Applet1 requests to the JCRE access to the public methods of the object Applet2 by calling the system method JCSystem.getAppletShareableInterfaceObject(). Immediately JCRE asks the object Applet2 to give sharing interface to the object Applet1, by the automatic calling by JCRE of the method getShareableInterfaceObject(), implemented by the class of the object Applet2. If the object Applet2 admits the sharing, then Applet1 will obtain a reference to Applet2 by which it may access the public methods of the object Applet2. If Applet1 and Applet2 are in the same context, meaning that they belong to the same package, it is not necessary to take the steps described above in order to cooperate with the intermediary JCRE.

Memory and memory objects management is an important concept because the smart card memory is quite small and restrictive. Moreover, some implementations of the Java Card platform do not provide Garbage Collector – a program that cleans the memory occupied by objects not used any more. So, when an object is created it is created in the non-volatile memory. If it is desired that an octet string or an object should be moved to the volatile memory, meaning to become „transient", it is used one of the methods: *static byte[] makeTransientByteArray(short length, byte event)*, *static Object makeTransientObjectArray(short length, byte event)*, *static short[] makeTransientShortArray(short length, byte event)*. It must be taken into account that an object or an octet string that is „transient" has a series of disadvantages such as: it does not persist in the memory if something bad happens along the card sessions and cannot have it fields update during the transactions.

An important concept available for the Java Card platform is that concerning the ***persistent transactions***. Similar to databases, for the card operating system it should

ISSN 2067 – 4074

be applied the atomic modification of some memory zones, meaning that the fields of an object in the non-volatile memory wheather modify all in the same time or they do not modify at all. This can be achieved by: *JCSystem.beginTransaction()*, *JCSystem.commitTransaction()*, *JCSystem.abortTransaction()*. In the specifications it is mentioned that JCRE cannot sustain imbricate transactions.

4. Practical sample of developing a Java applet card

This practical example will be deployed as in the Message-Passing model. In what concerns the models JCRMI and SATSA, because of the space limits, they will be analyzed in future approaches. In order to develop a Java Card application it is needed a Java 2 Standard Edition compiler, preferably version 1.4.1 [8], a Sun Java Card Development Toolkit – SJCDT [5], [6] and optionally an integrated developing environment – IDE such as Borland JBuilder, Net Beans or IntelliJ IDEA. The source code with explanations in English is taken of the examples provided by SJCDT and is entirely presented in the annexes. The source code, whether it contains several syntax modifications, is as in the license property of Sun Microsystems. This paragraph is spitted in two parts: one which explains the way of compiling, converting and uploading on the card the example and one which explains significant parts of the code and interprets results.

4.1 Steps taken for compiling and uploading an applet on the card

After it is downloaded and extracted Sun Java Card Development Toolkit – SJCDT [5-6], the source code entirely presented in the annexes is written in a text file named Wallet.java. We set the Environment Variables, according to the user's guide for the developing kit [12] so that JC_HOME should contain the path for the kit distribution – the directory that includes the bin directory with the batch files apdutool.bat, cref.exe, converter.bat; and JAVA_HOME should contain the distribution J2SE 1.4.1 [8] – the directory which includes the bin directory with programs such as java.exe, javac.exe.

There are two ways to compile, develop and test the applets for the Java Card platform. The first is that which uses the Java simulator included in the developing kit named – JCWDE – Java Card Workstation Development Environment, and the second is that which uses – C-JCRE – C-language Java Card Runtime Environment. C-JCRE represents the reference interpretation written in C by Sun Microsystem as to simulate and implement JCRE according to the specifications. The most "real" way, for which the procedure is similar to that in the practice is thoroughly described in this chapter and represents the way it is used C-JCRE.

The steps taken for compiling, uploading and simulating a card applet are:

- Saving the source code as in the annex into the file Wallet.java in the directory structure
  - 'Wallet1\com\sun\javacard\samples\wallet';
- ***Step 1 – Compilation*** – Positioning into the directory 'Wallet1\com\sun\javacard\samples\wallet' and subsequently introducing the commands:
  - SET _CLASSES=.;%JC_HOME%\lib\apduio.jar;%JC_HOME%\lib\apdutool.jar;%JC_HOME%\lib\jcwde.jar;%JC_HOME%\lib\converter.jar;%JC_HOME%\lib\scriptgen.jar;%JC_HOME%\lib\offcardverifier.jar;%JC_HOME%\lib\api.jar;%JC_HOME%\lib\installer.jar;%JC_HOME%\lib\capdump.jar;%JC_HOME%\lib\javacardframework.jar;%JC_HOME%\samples\classes;%CLASSPATH%;

- %JAVA_HOME%\bin\javac.exe -g -classpath %_CLASSES% com\sun\javacard\samples\wallet\Wallet.java

- ***Step 2 – Editing the configuration file for card uploading*** – in the directory Wallet1 in the directory structure mentioned for step 1 it is created a text file for configuration named 'wallet.app' that includes the following:
  - // applet AID
com.sun.javacard.installer.InstallerApplet 0xa0:0x0:0x0:0x0:0x0:0x62:0x3:0x1:0x8:0x1
com.sun.javacard.samples.wallet.Wallet
0xa0:0x0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x6:0x1

- ***Step 3 – Conversion of the java byte code class into a binary file that can be interpreted by the JCVM on the card*** – with the configuration file from step 2, it is called in the command prompt, also from the directory Wallet1, the following instruction:

%JC_HOME%\bin\converter.bat -config com\sun\javacard\samples\wallet\ Wallet.opt

If the conversion has ended without any error, then in the directory (Wallet1\com\sun\javacard\samples\wallet) where there is the normal bytecode class (Wallet.class) and the configuration file (wallet.opt) there will appear a new directory that contains three files. The file with the extension CAP – Converted Applet – is the binary form which will be understood by the card JCVM. The file with the extension JCA – Java Card Assembly – is the text-assembler representation of the binery-compressed file CAP that will be uploaded on the card. The file with the extension EXP – export – differently from the CAP and like the JCA it is not uploaded on the card. EXP is used by the conversion instrument (converter.bat) to also convert the necessary elements from the classes or packages which are imported from the bytecode class (Wallet.class).

- ***Step 4 – Verifying the CAP, JCA, EXP files*** – this is OPTIONAL. It is separately executed for each of the EXP and CAP files, resulted in step 3, but not before copying all the structure of EXP files including the directories from the directory api_export_file of the distribution SJCDT [5-6], into the directory Wallet1. Of course it is possible also generate from the "assembler source code" – of the JCA file an equivalent CAP file with the command

%JC_HOME%\bin\capgen.bat com\sun\javacard\samples\wallet\ javacard\wallet.jca
The commands for verifying the EXP and CAP files are:
  - %JC_HOME%\bin\verifyexp.bat com\sun\javacard\samples\wallet\ javacard\wallet.exp
  - %JC_HOME%\bin\verifycap.bat com\sun\javacard\samples\wallet\ javacard\wallet.exp %JC_HOME%\api_export_files\java\lang\ javacard\lang.exp %JC_HOME%\api_export_files\javacard\ framework\javacard\framework.exp com\sun\javacard\samples\wallet\ javacard\wallet.cap

- ***Step 5 – uploading the binary execution file into the permanent memory of the card when this one is manufactured*** – it is EXCLUSIVE. EXCLUSIVE means that whether it is executed step 5 and stop, or it is not executed step 5 and we pass directly to 6. In the step 5, it is generated from the JCA file (resulted in step 3) a masking-file that is uploaded in the non-volatile memory of the card when this is manufactured and which will disappear from the memory only after the physical destruction of the card. The command (the file maskgen.bat is available only in some distributions directly colaborating with the card producers but may also be downloaded from the) for generating the masking file is:
%JC_HOME%\bin\maskgen.bat cref com\sun\javacard\samples\wallet\ javacard\wallet.jca

ISSN 2067 – 4074

- ***Step 6 – uploading the binary execution file in the volatile memory of the card*** – it is executed only if step 5 was excluded. It is run the so-called off-card installer. The first command is:
    - %JC_HOME%\bin\scriptgen.bat -o Wallet.scr com\sun\javacard\ samples\wallet\javacard\wallet.cap

This command creates the script command file APDU – Wallet.scr with the help of the binary file CAP (wallet.cap). The resulting file – Wallet.scr – must be adjusted and it contains the APDU Commands necessary to upload the applet into the card memory. The content of the file Wallet.scr will be briefly explained in this chapter. The adjustment is made by the text editing of the file Wallet.scr. It is added the line 'powerup;' at the beginning of the file and 'powerdown;' at the end of the file.
Then, in a new command prompt window it must be started the program which emulates JCRE. There are two programs that can emulate a JCRE: one Java implemented (jcwde.bat) and one C implemented (cref.exe). The one in C, called C-JCRE, is the most important and in the same time it represents the refference implementation of JCRE. This implementation is executed from the file cref.exe in the 'bin' directory, from the development kit distribution Java Card [5-6]. So, in the new command prompt window it is wrote the command:

- cref.exe –o eeprom1

The command is meant to save the EEPROM memory of the card, after it is modified it by APDU commands. The command launches the JCRE emulation, and JCRE listens to TCP/IP on the pre-defined port 9025, in order to receive APDU commands. The emulation of JCRE stops when it receives a powerdown; by the APDU command script file (extension scr).
Now in the old command prompt window it is run the command:

- %JC_HOME%\bin\apdutool.bat Wallet.scr > Wallet.scr.out

By this command, with the help of the program launched with cu apdutool.bat, it is transmitted APDU commands from the host-type application to JCRE. For the moment JCRE runs in a window and after the APDU commands received modifies the EEPROM memory and the content is saved in the file eeprom1. The file Wallet.scr.out includes the APDU Answers from the applet that runs over C-JCRE.

- ***Step 7 – simulating the action between the host application and the card applet by CAD*** – it is executed after step 6. At step 6, the JCRE simulation has ended. Now it must be relaunched by the command:
    - cref.exe –i eeprom1 –o eeprom1

The command takes on the memory image from the eeprom1 file, modifies it according to the received APDU commands and saves it again with the modifications in the same eeprom1 file. On the same JCRE port it simulates receiving APDU commands and by the command

- %JC_HOME%\bin\apdutool.bat demoWallet.scr > demoWallet.scr.cjcre.out

it is sent the APDU Commands for test and simulation of the applet, and the APDU Answers are in the file demoWallet.scr.cjcre.out.

In order to completely understand the process, we should understand from step 6 the content of Wallet.scr and Wallet.scr.out, and from step 7 the content of demoWallet.scr and demoWallet.scr.cjcre.out, i.e. to understand the user's guide [12] from the Java card kit. The content of demoWallet.scr and demoWallet.scr.cjcre.out is presented in the annexes.

4.2 Interpretation of the code and results

As can be seen in the source code presented in the first annex, the applet has to import the package *javacard.framework.*,* to extend the *Applet* class and to implement the

methods mentioned for the life cycle of an applet. In Table 1 is presented the draft of the application:

Table 1. Source code for the application draft

```
package com.sun.javacard.samples.wallet;
import javacard.framework.*;
public class Wallet extends Applet {
...
  //constructor
  private Wallet (byte[] bArray,short bOffset,byte bLength) {...}
  //Life-cycle methods – specifice fiecarui applet
  public static void install(...) {...}
  public void select() {...}
  public void deselect() {...}
  public void process(APDU apdu) {...}
  //private methods – specifice doar acestui applet
  private void credit(APDU apdu) {...}
  private void debit(APDU apdu) {...}
  private void getBalance(APDU apdu) {...}
  private void verify(APDU apdu) {...}
...
}
```

When the source code is written, the developer should decide in what concerns the structure of the APDU Commands and Answers. The structure of commands depends mostly upon the service that the applet-application provides. For instance, if it is an applet providing a service of electronic wallet, it should provide sub-services such as debit or credit transactions for the amount of money on the card, verifying the balance sheet of the card, assuring the security of the access to the applet by PIN. If it is about a loyalty applet for a gym hall or about a medical insurance applet, then it should offer services such as: personal identification information, access number in locations, and the person legally reliable for the card owner, diseases.

Practically, for the Wallet-electronic wallet application, there are given several models of APDU Commands designed by those who wrote the source code of the applet:

- **PIN verification**

Java private method	CLA 1 byte	INS 1 byte	P1 1 byte	P2 1 byte	Lc 1 byte	Data Field 5 bytes	Le
verify()	0x80	0x20	0x00	0x00	0x05	0x01 0x02 0x03 0x04 0x05	0x02

CLA with the value 80 hex means that we intent to access the application electronic wallet; INS with the value 20 hex means that it is desired to execute the method verify(); P1 and P2 are not defined; Lc has the value 5 i.e. the Data Field field will have 5 octets; the Data Field field has octets and clearly contains the PIN 12345, although in the real applications this should be encrypted; and Le has the value 2 that means that it is expected maximum 2 octets as an answer;

- **Interrogation to find which amount of money the card contains**

Java private method	CLA 1 byte	INS 1 byte	P1 1 byte	P2 1 byte	Lc 1 byte	Data Field 5 bytes	Le
getBalance()	0x80	0x50	0x00	0x00	0x00	Nothing	0x7F

The interpretation is the following: INS with the value 50 hex means that we intent to execute the method getBalance(); P1 and P2 are not defined; Lc has the value 0 that

ISSN 2067 – 4074

means that the field Data Field will not be defined; the field Data Field holds 0 octets; and Le has the value 2 i.e. it is expected a maximum of 2 octets as an answer;

- **APDU command for debit – e.g. to deposit money on the card – usually executed by a bank**

Java private method	CLA 1 byte	INS 1 byte	P1 1 byte	P2 1 byte	Lc 1 byte	Data Field 5 bytes	Le
debit()	0x80	0x40	0x00	0x00	0x01	0x64	0x7F

The interpretation is the following: INS with the value 40 hex means that it is desired to execute the method debit(); P1 and P2 are not defined; Lc has the value 1 i.e. the field Data Field will have one octet; the field Data Field 1 octet and contains the value 100 in decimals –means that the amount on the card should be increased with 100 monetary units; and Le has the value 0x7F meaning that it is expected a maximum of 127 octets as an answer;

It can be noticed in annex 1 that in the builder are allocated the objects needed for the entire life cycle of the applet. Then the static method *install()* should directly or indirectly call the static method *register()*, because JCRE, after installing the applet, this must at his turn register into JCRE. In this case, the register method is indirectly call by the builder by the *install()* method. Each time that JCRE receives an APDU command from the host-type application, it will call the *process()* method. The source code in the annex is readable and „self-explained" for a Java programmer – it self-explains by the comments.
In annex 3 are presented the APDU commands sent by the host to JCRE and immediately having the separator ',' follows the APDU answer. It considers the lines in Table 2:

Table 2. Interpretation of APDU commands and answers from the log file in Annex 3.

CLA: 80, INS: 40, P1: 00, P2: 00, Lc: 01, 64, Le: 00, SW1: 6a, SW2: 85
CLA: 80, INS: 30, P1: 00, P2: 00, Lc: 01, 64, Le: 00, SW1: 90, SW2: 00

It is obvious that by the first command we intended the extraction from the card of 100 monetary coins units from the card – INS=0x40 intends the execution of the *debit()* method and DataField=0x64 means 100 in decimal. The operation has been a failure because of the octets as answer status (SW1=0x6Abut SW2=0x85). The value of the answer is at the free choice of the developer when he throws the exception by the sequence „if ((short)(balance - debitAmount) < (short)0) ISOException.throwIt(SW_NEGATIVE_BALANCE);" where SW_NEGATIVE_BALANCE has the value 0x6A85 defined by the sequence „final static short SW_NEGATIVE_BALANCE = 0x6A85;" as in the source code in annex 1. This is the way we observe and interpret all the results and the source code in the annexes of this presentation. The chosen example is quite simple and does not use specific cryptography elements or atomic transactions as normal for a practical application.

5. Conclusions

Before Java Cards appear, smart card software was depended on the manufactures. Most smart card development kits were card and reader specific. Some have externalized the card and reader descriptions so that the buyer of the kit can adapt the software to new cards and readers. Also, most smart card systems had been closed systems, consisting of a specific card from a card manufacturer working with a specific

terminal from a terminal manufacturer. Sometimes the same company manufactured both the card and the reader. As a result, standard-specified, paper interoperability had rarely proved. Now, the developers should no more be afraid about the diversity of smart cards manufactures and operating systems, as long as they are using Java card platform.

References

[1] Zhiqun Chen, „Java Card Technology for Smart Cards: Architecture and Programmer's Guid", Addison Wesley Publishing House, USA, June 2000.

[2] C. Enrique Ortiz, May 2003, on-line article: http://developers.sun.com/techtopics/mobility/javacard/articles/javacard1/

[3] C. Enrique Ortiz, September 2003, on-line article: http://developers.sun.com/techtopics/mobility/javacard/articles/javacard2/

[4] Ion Ivan, Paul Pocatilu, Marius Popa, Cristian Toma, "The Digital Signature and Data Security in e-commerce", The Economic Informatics Review Nr. 3/2002, Bucharest 2002.

[5] Tools Sun Java Card Development Toolkit 2.2.1: http://java.sun.com/products/javacard/index.jsp

[6] Sun Java Card Development Toolkit 2.2.1: http://java.sun.com/products/javacard/dev_kit.html

[7] ISO/IEC 7816 Part 4: Interindustry command for interchange, http://www.ttfn.net/techno/smartcards/iso7816_4.html

[8] Java 2 Standard Edition Software Development Kit: http://java.sun.com/products/archive/j2se/1.4.1_07/

[9] Virtual Machine Speification 2.2.1, Octomber 2003: http://java.sun.com/products/javacard/specs.html

[10] Runtime Environment Specification 2.2.1, Octomber 2003 http://java.sun.com/products/javacard/specs.html

[11] Application Programming Specification 2.2.1, Octomber 2003: http://java.sun.com/products/javacard/specs.html

[12] Programming Manual, Octomber 2003, Application Progaming Notes 2.2.1 included in [5a]

[13] User Manual, Octomber 2003, Development Kit User Guide 2.2.1 included in [6]

[14] Paul Pocatilu, Cristian Toma, *Mobile Applications Quality*, International Conference "Science and economic education system role in development from Republic of Moldavia", Chişinău, September 2003, pg. 474-478

[15] Scott Guthery, Tim Jurgensen, „Smart Card Developer's Kit", Macmillan Computer Publishing House, ISBN: 1578700272, USA 1998: http://unix.be.eu.org/docs/smart-card-developer-kit/ewtoc.html

[16] Cristian TOMA, *Secure Protocol in Identity Management using Smart Cards*, Revista "Informatica Economica", vol. 9, Nr. 2, Bucuresti, 2005, p. 135 – 140

[17] Cristian TOMA - Security in Software Distributed Platforms, AES Publishing House, Bucharest, 2008, ISBN 978-606-505-125-6.

ISSN 2067 – 4074

ANNEX 1: Source code of electronic wallet applet for Java Card platform

```
package com.sun.javacard.samples.wallet;
import javacard.framework.*;
//import javacardx.framework.*;

public class Wallet extends Applet {
  /* constants declaration */
  // code of CLA byte in the command APDU header
  final static byte Wallet_CLA =  (byte) 0x80;
  // codes of INS byte in the command APDU header
  final static byte VERIFY =      (byte) 0x20;
  final static byte CREDIT =      (byte) 0x30;
  final static byte DEBIT =       (byte) 0x40;
  final static byte GET_BALANCE = (byte) 0x50;
  // maximum balance
  final static short MAX_BALANCE = 0x7FFF;
  // maximum transaction amount
  final static byte MAX_TRANSACTION_AMOUNT = 127;
  // maximum number of incorrect tries before the PIN is blocked
  final static byte PIN_TRY_LIMIT = (byte) 0x03;
  // maximum size PIN
  final static byte MAX_PIN_SIZE =  (byte) 0x08;
  // signal that the PIN verification failed
  final static short SW_VERIFICATION_FAILED =        0x6300;
  // signal the the PIN validation is required
  // for a credit or a debit transaction
  final static short SW_PIN_VERIFICATION_REQUIRED =  0x6301;
  // signal invalid transaction amount
  // amount > MAX_TRANSACTION_AMOUNT or amount < 0
  final static short SW_INVALID_TRANSACTION_AMOUNT = 0x6A83;
  // signal that the balance exceed the maximum
  final static short SW_EXCEED_MAXIMUM_BALANCE =     0x6A84;
  // signal the the balance becomes negative
  final static short SW_NEGATIVE_BALANCE =           0x6A85;
  /* instance variables declaration */
  OwnerPIN pin;
  short balance;

 private Wallet (byte[] bArray,short bOffset,byte bLength){
    // It is good programming practice to allocate
    // all the memory that an applet needs during
    // its lifetime inside the constructor
    pin = new OwnerPIN(PIN_TRY_LIMIT,   MAX_PIN_SIZE);
    byte iLen = bArray[bOffset]; // aid length
    bOffset = (short) (bOffset+iLen+1);
    byte cLen = bArray[bOffset]; // info length
    bOffset = (short) (bOffset+cLen+1);
    byte aLen = bArray[bOffset]; // applet data length
    // The installation parameters contain the PIN initialization value
    pin.update(bArray, (short)(bOffset+1), aLen);
    register();
  } // end of the constructor

  public static void install(byte[] bArray, short bOffset, byte bLength){
    // create a Wallet applet instance
    new Wallet(bArray, bOffset, bLength);
  } // end of install method
```

```
 public boolean select() {
   // The applet declines to be selected if the pin is blocked.
   if ( pin.getTriesRemaining() == 0 ) return false;
   return true;
 }// end of select method

 public void deselect() {
   // reset the pin value
   pin.reset();
 }

 public void process(APDU apdu) {
   // APDU object carries a byte array (buffer) to
   // transfer incoming and outgoing APDU header
   // and data bytes between card and CAD

   // At this point, only the first header bytes
   // [CLA, INS, P1, P2, P3] are available in
   // the APDU buffer.
   // The interface javacard.framework.ISO7816
   // declares constants to denote the offset of
   // these bytes in the APDU buffer
   byte[] buffer = apdu.getBuffer();
   // check SELECT APDU command
 buffer[ISO7816.OFFSET_CLA]=(byte)(buffer[ISO7816.OFFSET_CLA]&(byte)0xFC);

 if((buffer[ISO7816.OFFSET_CLA]==0)&&
 (buffer[ISO7816.OFFSET_INS]==(byte)(0xA4))) return;
   // verify the reset of commands have the
   // correct CLA byte, which specifies the command structure
   if (buffer[ISO7816.OFFSET_CLA] != Wallet_CLA)
ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
   switch (buffer[ISO7816.OFFSET_INS]) {
     case GET_BALANCE:   getBalance(apdu); return;
     case DEBIT:         debit(apdu);      return;
     case CREDIT:        credit(apdu);     return;
     case VERIFY:        verify(apdu);     return;
     default:       ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
   }
 }   // end of process method

 private void credit(APDU apdu) {
   // access authentication
if (! pin.isValidated())ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
   byte[] buffer = apdu.getBuffer();
   // Lc byte denotes the number of bytes in the
   // data field of the command APDU
   byte numBytes = buffer[ISO7816.OFFSET_LC];
// indicate that this APDU has incoming data and receive data starting // from the offset
ISO7816.OFFSET_CDATA following the 5 header bytes.
   byte byteRead = (byte)(apdu.setIncomingAndReceive());

   // it is an error if the number of data bytes
   // read does not match the number in Lc byte
   if (( numBytes != 1 ) || (byteRead != 1))
                       ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
```

ISSN 2067 – 4074

```
    // get the credit amount
    byte creditAmount = buffer[ISO7816.OFFSET_CDATA];

    // check the credit amount
    if (( creditAmount > MAX_TRANSACTION_AMOUNT) || ( creditAmount < 0 ))
      ISOException.throwIt(SW_INVALID_TRANSACTION_AMOUNT);

    // check the new balance
    if ((short)(balance + creditAmount) > MAX_BALANCE)
            ISOException.throwIt(SW_EXCEED_MAXIMUM_BALANCE);

    // credit the amount
    balance = (short)(balance + creditAmount);
  } // end of deposit method

  private void debit(APDU apdu) {
    // access authentication
    if (!pin.isValidated())
ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
    byte[] buffer = apdu.getBuffer();
    byte numBytes = (byte)(buffer[ISO7816.OFFSET_LC]);
    byte byteRead = (byte)(apdu.setIncomingAndReceive());
    if (( numBytes != 1 ) || (byteRead != 1))
      ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
    // get debit amount
    byte debitAmount = buffer[ISO7816.OFFSET_CDATA];
    // check debit amount
    if ((debitAmount > MAX_TRANSACTION_AMOUNT)||  ( debitAmount < 0 ))
      ISOException.throwIt(SW_INVALID_TRANSACTION_AMOUNT);
    // check the new balance
    if ((short)( balance - debitAmount ) < (short)0)
            ISOException.throwIt(SW_NEGATIVE_BALANCE);
    balance = (short) (balance - debitAmount);
  } // end of debit method

  private void getBalance(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
    // inform system that the applet has finished
    // processing the command and the system should
    // now prepare to construct a response APDU
    // which contains data field
    short le = apdu.setOutgoing();

    if (le < 2) ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
    //informs the CAD the actual number of bytes returned
    apdu.setOutgoingLength((byte)2);
    // move the balance data into the APDU buffer starting at the offset 0
    buffer[0] = (byte)(balance >> 8);
    buffer[1] = (byte)(balance & 0xFF);

    // send the 2-byte balance at the offset 0 in the apdu buffer
    apdu.sendBytes((short)0, (short)2);
  } // end of getBalance method
  private void verify(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
    // retrieve the PIN data for validation.
    byte byteRead = (byte)(apdu.setIncomingAndReceive());
```

```
    // check pin, the PIN data is read into the APDU buffer
    // at the offset ISO7816.OFFSET_CDATA, the PIN data length = byteRead
    if (pin.check(buffer, ISO7816.OFFSET_CDATA, byteRead) == false)
                        ISOException.throwIt(SW_VERIFICATION_FAILED);

  } // end of verify method
} // end of class Wallet
```

ISSN 2067 – 4074

ANNEX 2: APDU commands and expected responses from the test-simulation file demoWallet1.scr

```
////////////////////////////////////////////////////////////////////
// Select all installed Applets
////////////////////////////////////////////////////////////////////

powerup;

// Select the installer applet
0x00 0xA4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08 0x01 0x7F;
// 90 00 = SW_NO_ERROR

// create wallet applet
0x80 0xB8 0x00 0x00 0x14 0x0a 0xa0 0x0 0x0 0x0 0x62 0x3 0x1 0xc 0x6 0x1 0x08 0 0
0x05 0x01 0x02 0x03 0x04 0x05 0x7F;

////////////////////////////////////////////////////////////////////
// Initialize Wallet
////////////////////////////////////////////////////////////////////

//Select Wallet
0x00 0xA4 0x04 0x00 0x0a 0xa0 0x0 0x0 0x0 0x62 0x3 0x1 0xc 0x6 0x1 0x7F;
// 90 00 = SW_NO_ERROR

//Verify user pin
0x80 0x20 0x00 0x00 0x05 0x01 0x02 0x03 0x04 0x05 0x7F;
//90 00 = SW_NO_ERROR

//Get wallet balance
0x80 0x50 0x00 0x00 0x00 0x02;
//0x00 0x00 0x00 0x00 0x90 0x00 = Balance = 0 and SW_ON_ERROR

//Attemp to debit from an empty account
0x80 0x40 0x00 0x00 0x01 0x64 0x7F;
//0x6A85 = SW_NEGATIVE_BALANCE

//Credit $100 to the empty account
0x80 0x30 0x00 0x00 0x01 0x64 0x7F;
//0x9000 = SW_NO_ERROR

//Get Balance
0x80 0x50 0x00 0x00 0x00 0x02;
//0x00 0x64 0x9000 = Balance = 100 and SW_NO_ERROR

//Debit $50 from the account
0x80 0x40 0x00 0x00 0x01 0x32 0x7F;
//0x9000 = SW_NO_ERROR

//Get Balance
0x80 0x50 0x00 0x00 0x00 0x02;
//0x00 0x32 0x9000 = Balance = 50 and SW_NO_ERROR

//Credit $128 to the account
0x80 0x30 0x00 0x00 0x01 0x80 0x7F;
```

```
//0x6A83 = SW_INVALID_TRANSACTION_AMOUNT

//Get Balance
0x80 0x50 0x00 0x00 0x00 0x02;
//0x00 0x32 0x9000 = Balance = 50 and SW_NO_ERROR

//Debit $51 from the account
0x80 0x40 0x00 0x00 0x01 0x33 0x7F;
//0x6A85 = SW_NEGATIVE_BALANC

//Get Balance
0x80 0x50 0x00 0x00 0x00 0x02;
//0x00 0x32 0x9000 = Balance = 50 and SW_NO_ERROR

//Debit $128 from the account
0x80 0x40 0x00 0x00 0x01 0x80 0x7F;
//0x6A83 = SW_INVALID_TRANSACTION_AMOUNT

//Get Balance
0x80 0x50 0x00 0x00 0x00 0x02;
//0x00 0x32 0x9000 = Balance = 50 and SW_NO_ERROR

//Reselect Wallet applet so that userpin is reset
0x00 0xA4 0x04 0x00 0x0a 0xa0 0x0 0x0 0x0 0x62 0x3 0x1 0xc 0x6 0x1 0x7F;
// 90 00 = SW_NO_ERROR

//Credit $127 to the account before pin verification
0x80 0x30 0x00 0x00 0x01 0x7F 0x7F;
//0x6301 = SW_PIN_VERIFICATION_REQUIRED

//Verify User pin with wrong pin value
0x80 0x20 0x00 0x00 0x04 0x01 0x03 0x02 0x66 0x7F;
//0x6300 = SW_VERIFICATION_FAILED

//Verify user pin again with correct pin value
//0x80 0x20 0x00 0x00 0x08 0xF2 0x34 0x12 0x34 0x56 0x10 0x01 0x01 0x7F;
0x80 0x20 0x00 0x00 0x05 0x01 0x02 0x03 0x04 0x05 0x7F;
//0x9000 = SW_NO_ERROR

//Get balance with incorrrect LE value
0x80 0x50 0x00 0x00 0x00 0x01;
//0x6700 = ISO7816.SW_WRONG_LENGTH

//Get balance
0x80 0x50 0x00 0x00 0x00 0x02;
//0x00 0x32 0x9000 = Balance = 50 and SW_NO_ERROR

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

ANNEX 3: Received APDU responses from the C-JCRE when this was asked with APDU commands – file demoWallet1.scr.cjcre.out

```
Java Card 2.2.1 APDU Tool, Version 1.3
```

ISSN 2067 – 4074

Opening connection to localhost on port 9025.
Connected.
Received ATR = 0x3b 0xf0 0x11 0x00 0xff 0x00
CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 09, a0, 00, 00, 00, 62, 03, 01, 08, 01, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: b8, P1: 00, P2: 00, Lc: 14, 0a, a0, 00, 00, 00, 62, 03, 01, 0c, 06, 01, 08, 00, 00, 05, 01, 02, 03, 04, 05, Le: 0a, a0, 00, 00, 00, 62, 03, 01, 0c, 06, 01, SW1: 90, SW2: 00
CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 0a, a0, 00, 00, 00, 62, 03, 01, 0c, 06, 01, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: 20, P1: 00, P2: 00, Lc: 05, 01, 02, 03, 04, 05, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 00, SW1: 90, SW2: 00
CLA: 80, INS: 40, P1: 00, P2: 00, Lc: 01, 64, Le: 00, SW1: 6a, SW2: 85
CLA: 80, INS: 30, P1: 00, P2: 00, Lc: 01, 64, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 64, SW1: 90, SW2: 00
CLA: 80, INS: 40, P1: 00, P2: 00, Lc: 01, 32, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 32, SW1: 90, SW2: 00
CLA: 80, INS: 30, P1: 00, P2: 00, Lc: 01, 80, Le: 00, SW1: 6a, SW2: 83
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 32, SW1: 90, SW2: 00
CLA: 80, INS: 40, P1: 00, P2: 00, Lc: 01, 33, Le: 00, SW1: 6a, SW2: 85
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 32, SW1: 90, SW2: 00
CLA: 80, INS: 40, P1: 00, P2: 00, Lc: 01, 80, Le: 00, SW1: 6a, SW2: 83
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 32, SW1: 90, SW2: 00
CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 0a, a0, 00, 00, 00, 62, 03, 01, 0c, 06, 01, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: 30, P1: 00, P2: 00, Lc: 01, 7f, Le: 00, SW1: 63, SW2: 01
CLA: 80, INS: 20, P1: 00, P2: 00, Lc: 04, 01, 03, 02, 66, Le: 00, SW1: 63, SW2: 00
CLA: 80, INS: 20, P1: 00, P2: 00, Lc: 05, 01, 02, 03, 04, 05, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 00, SW1: 67, SW2: 00
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 32, SW1: 90, SW2: 00