IBM

OpenCard Framework 1.2 Programmer's Guide

BOEB-OCFP-00

IBM

OpenCard Framework 1.2 Programmer's Guide

BOEB-OCFP-00

OpenCard Framework 1.2 — Programmer’s Guide

Fourth Edition

December, 1999

This book provides information about the OpenCard Framework.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

IBM welcomes your comments. A form for readers’ comments may be provided at the back of this publication, or you may address your comments or questions to the following address:

IBM Deutschland Entwicklung GmbH
Information Development, Dept. 3248
Schoenaicher Strasse 220
71032 Boeblingen
Germany

FAX (Germany): 07031 16 3456
FAX (Other Countries): (+49) 7031 16 3456

IBM Mail Exchange: DEIBMBM9 at IBMMAIL
Internet e-mail: `idepubs@vnet.ibm.com`
World Wide Web: `www.opencard.org`

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:
- Title and order number of this book
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Trademarks and service marks

Contents

Figures

Preface

By convention, a preface contains *meta-information* about a book describing its structure and how the reader can best make use of it. We recommend to read this preface to understand how this manual is organized, what typesetting conventions are used, where to get sample code, where to find additional information, etc.

About this manual

This manual describes the OPENCARD FRAMEWORK, a *smart card*[1] middleware implemented in the JAVA programming language.

The OPENCARD FRAMEWORK sits between a smart card-aware application or applet written in JAVA and the *card reader* (also known as *Card Acceptance Device (CAD)* or *Interface Device (IFD)*). The framework enables programmers to write smart card applications against a high-level programming interface that hides the complexities of interacting with cards and card readers from different vendors. Thus, applications developed on top of the OPENCARD FRAMEWORK are readily usable on all platforms supporting the JAVA environment and provide the interoperability across card and reader vendors that is required to deploy applications written as applets in an INTERNET environment.

How to use this manual

This manual is intended for developers and programmers who want to learn the basics of using the OPENCARD FRAMEWORK to develop smart card aware applications and applets in the JAVA programming language.

The manual is also intended for OPENCARD component developers who implement pluggable framework components for card terminals or cards.

Throughout this manual, we've provided sample code to illustrate programming with the OPENCARD FRAMEWORK reference implementation. Notice that the code is incomplete and thus would not compile. For the sake of brevity, we've shown only those code snippets relevant to illustrate some point and haven't always shown how some instance variables were initialized. However, we have tried to include sufficient comments and variable declarations to make the code illustrations self-explanatory.

Assumptions

We assume that the reader is already familiar with the JAVA programming language and development environment.

However, the reader may need some help in familiarizing himself with smart card technology and the terminology used in this field in order to better understand the concepts presented in this document.

1. The term *smart card* refers to integrated circuit cards as specified in the ISO / IEC standards 7816, JAVACARDS as specified in SUN's *JavaCard 2.0* specifications, or any other smart tokens (including smart accessories).

Where you can get OpenCard Framework

The OpenCard Consortium maintains a web-site at `http://www.opencard.org/` which includes a download section from which you can obtain documentation, binaries, class files, and source code.

You can access sample code for the Java Developer at `http://www.javaworld.com/jw-10-1998/javadev/javadev.zip`.

Font conventions used in this manual

Italics are used for

- new terms, when they are defined;
- the titles of documents, papers, and books appearing in references.

`Typewriter font` is used for

- anything (in particular source code) that would be typed verbatim.

Smallcaps are used for

- names (such as company names, product names, registered trademarks, etc.).

Related documents

- *OpenCard Framework 1.0 - White Paper*. This paper provides a gentle introduction to the OpenCard Framework, describing its objectives, the architecture, and the reference implementation.

Feedback

Please help us improve the quality of this manual by telling us about any flaws you can spot or by making suggestions as to how you think the material could be presented better.

Send any corrections, amendments, suggestions, or comments to this e-mail address:

`opencard-info@opencard.org`

Chapter 1. Introduction

This chapter prepares the stage for OCF. We'll briefly talk about the fundamentals of smart card application development in general and show how the OpenCard Framework comes in and simplifies the task. Everyone should read this chapter, though readers familiar with developing smart card applications may want to skip the first section.

Developing smart card applications

Smart card applications consist of a *card-external* application interacting with a *card-resident* component. The card-external application comprises the program code running on some computing platform such as a personal computer, a network computer, an *ATM* (automatic-teller machine), or a personal digital assistant. The card-resident component comprises data and functions in the smart card itself[2].

Let's take a closer look at what's involved when the application interacts with the smart card. Don't be worried by the technical details presented here, because the OpenCard Framework will hide most of the complexity from you. The intent here is to provide some background and understanding of what is happening so that the need for and the usefulness of OCF can be appreciated.

Interactions with the smart card occur by exchanging pairs of *APDUs* (application protocol data units) and are initiated by the external application. Communication with the smart card is accomplished via the card reader, an I/O device attached to the computing platform into which the smart card is inserted. The application sends a *CommandAPDU* to the smart card by handing it to the card reader's device driver, which forwards it to the smart card. In return, the card sends back a *ResponseAPDU*, which the device driver hands back to the application.

A smart card's functions are determined by the set of CommandAPDUs that it understands. Although standards for smart cards do exist, the functionality may vary significantly. In other words, depending on the card vendor and/or the type of card, different functionalities are offered and the exact definition of the set of Command- and ResponseAPDUs may differ.

Similarly, there is a broad range of card readers on the market with widely differing functionalities. Very simple readers merely provide basic reader functionality offering a single slot to insert the card. More sophisticated readers offer multiple slots or include a *PIN*[3] pad and a display that can be used to perform cardholder verification. Card readers can attach to different I/O ports including serial port and PC Card slot. There are even versions available that can be inserted into the floppy drive. They come with proper driver software for a selection of operating systems. There is no single API to access the card reader driver.

Given these technicalities, development of smart card applications has been much of an art in the past and was confined to a relatively small number of specialized programmers. In addition, the dependencies on the make of the card and the card reader have usually prevented application developers from deploying their

2. The card-resident component is referred to as an *application* in ISO 7816.

3. Personal Identification Number

applications across a wide range of cards or card readers. As a consequence, smart card applications have been primarily ″vertical″ applications mostly implemented with a single type of card and maybe a small number of card readers.

The emergence of multi-application cards, made possible by advances in the underlying chip-card technology and the need to deploy smart card applications and applets in an INTERNET setting, where it is virtually impossible to know in advance what type of card or card reader the application will have to interact with, make it imperative that alternate ways of developing smart card applications be made available.

Developing applications using OpenCard Framework

Developers are normally concerned with implementing both parts of a smart card application, the card-resident component and the card-external program. OCF helps developers primarily in developing the card-external program code. They can then program against high-level APIs that let make the functionality and information contained in the smart card fully accessible. Having OCF sit in between the application and the smart card hides the complexity of interacting with the smart card. At the same time, the high-level of abstraction offered by the OPENCARD FRAMEWORK achieves the desired transparency with regard to dependencies on the particular type of smart card and/or card reader used in a particular setting. By virtue of its framework nature, OPENCARD FRAMEWORK adapts its capabilities at run-time to match the characteristics of a particular card reader device and/or the particular smart card inserted into the reader. For programmers, this means that they can concentrate on the application logic and need not be concerned with the intricacies of dealing with a particular reader or card.

In order to adapt itself to a particular situation, OCF relies on the availability of adequate JAVA components that can be plugged into the framework to address a particular card reader and card inserted. But don't worry, these components are typically developed by card reader manufacturers and card-chip manufacturers and are not something which application programmer have to implement. All they have to do is to make sure that the card reader and card type they choose to deploy the application are supported by OCF-compliant components. An up-to-date list of the devices supported can be found at the OCF web site.

OCF will not provide support in developing the card-resident part of the application. Normally, smart card vendors offer their cards together with a development toolkit that supports the development of the card-resident application component.

For conventional smart cards, the toolkit may contain tools supporting all or a subset of the following tasks:

Layout definition
This is the process of generating an EEPROM image from a high-level definition of the EEPROM layout. Most smart card applications maintain information in the smart card. This information is typically kept in EEPROM-based files on the card's file system. Layout definition is about identifying the information items that should go in the card and defining an appropriate file structure in the card. The latter includes specifying the type of file (transparent, record-oriented, cyclic), file names and/or identifiers, access conditions, initial data, etc.

Card initialization
This is the process of writing initialization data to the smart card EEPROM

(it could be compared to formatting a disk). The EEPROM image generated during the layout definition is transferred to the smart card. Depending on volume, this can be supported by special types of card readers that achieve a high throughput.

Card personalization
This is the process of writing cardholder-specific data to the smart card. After initialization, the smart card EEPROM reflects the basic file system structure as defined in the layout definition and may contain some initial data that is constant across all cards, such as *meta-information* about the file system structure, cryptographic key information, etc. During personalization, the information peculiar to an individual cardholder is written to the card prior to issuing it.

Look at the documentation accompanying the given OCF-compliant smart card for details on how to prepare the application's card-resident component. For smart cards compliant with the JAVACARD standard, the set of tools will likely be rather different. To date, only very few vendors offer JAVACARD-compliant smart cards. Look at the respective documentation to gain a better understanding of how to develop the card-resident *Applet*

Chapter 2. Getting started

This chapter provides a guide for writing a first simple smart card application. It will become apparent how simple and efficient smart card application development can be using the OpenCard Framework.

Preparing the system

Before one can start developing one's own smart card aware applications using the OpenCard Framework, there are a number of prerequisites to fulfill:

1. Get and install Jdk (Java Developer's Kit) - preferably version 1.1.6 or later, if you have not already done so. You can find versions of the Jdk for a number of operating systems at Java Soft's Jdk WWW site[4]. Alternatively, you can get the Jdk with one of the commercially available *Integrated Development Environments (IDE)* for Java.
2. Likewise, download and install the OpenCard Framework reference implementation available at the OCF WWW site[5] and follow the installation instructions.
3. Obtain an OCF-compliant smart card reader device. Check the list at `http://www.opencard.org/` for devices that are presently supported via OCF-compliant *CardTerminal* components. Check this list frequently as new components supporting different devices will become available.
4. Obtain OCF-compliant smart cards and the corresponding developers toolkit. Check the list at `http://www.opencard.org/` for cards that are presently supported via OCF-compliant *CardService* components. Check this list frequently as new components supporting different devices become available.

Writing your first sample application

Now that you've setup your system with both hardware and software, you are ready to code your first sample application: A program that reads the cardholder data from an OpenCard DEMOCARD[6].

If you do not own such a card, you can use any other smart card for which a `FileAccessCardService` is available. Use the tools provided with your smart card to create a transparent file in the master file (MF) of your smart card. The file must have the ID `0xc009` and should have access condition `ALWAYS` or `CHV` for read and write access. To initialize your card, you may prefer to start with the second sample application (see "Writing your second sample application" on page 8), which allows you to write data to that file.

First, we create a skeleton of the new application. The skeleton consists of import statements, a class definition, and a main method where the actual application code will be filled in:

```
import opencard.core.service.SmartCard;
import opencard.core.service.CardRequest;
import opencard.opt.iso.fs.FileAccessCardService;
```

4. `http://java.sun.com/products/jdk/`

5. `http://www.opencard.org/`

6. the card used for the OCF demo called "Internet Broker"

```
import opencard.opt.iso.fs.CardFile;
public class ReadFile {
  public static void main(String[] args)
  {
    System.out.println("reading smartcard file...");
    // here, the application code will be filled in
    System.exit(0);
  }
}
```

The SmartCard class provides an entry point to OCF. To wait for a smart card to be inserted, we will need the CardRequest class. FileAccessCardService is an interface which can be used to access data stored on file system based cards, and CardFile will turn out to be a useful helper class.

Before we can work with OCF, we have to initialize it. Afterwards, OCF has to be shut down to release system resources. Since errors may occur (for example because the user removes the smart card while it is accessed), we also have to implement some error handling. The code to do all that may look like this:

```
try {
  SmartCard.start();

  // here, OCF can be used

} catch OpenCard Exceptions {
  e.printStackTrace(System.err);

} finally { // even in case of an error...
  try {
    SmartCard.shutdown();
  } catch OpenCard Exceptions {
  e.printStackTrace(System.err);
  }
}
```

OCF uses exceptions to indicate errors. Therefore, the application code is put into a try/catch block. The first operation in this block initializes OCF. Any exceptions that are thrown during initialization, or while OCF is used, are caught and printed. Exception handling is discussed more thoroughly in “Handling exceptions” on page 21.

To make sure that OCF is shut down even in case of an error, the corresponding statement is located in the finally part of the try statement. At first glance, it may seem strange that shutting down OCF may cause another exception. However, the shutdown may cause dynamic libraries to be unloaded. An error in such an operation indicates serious problems of the operating system and should therefore not be ignored.

Now, we have dealt with the wrap-around code and can use OCF. The first thing we have to do is to wait for a smart card from which we can read. This is done by creating a CardRequest and waiting until it is satisfied:

```
// wait for a smartcard with file access support
CardRequest cr =
  new CardRequest(CardRequest.NEWCARD, null, FileAccessCardService.class);
SmartCard   sc = SmartCard.waitForCard(cr);

// here, we will read from the smart card

sc.close();
// printing the data can be done here
```

The argument for creating a CardRequest indicates that we are interested only in smart cards for which a FileAccessCardService can be instantiated. The waitForCard method will return when such a card has been inserted. It returns a SmartCard object that will be used by our application to refer to that card. The close method is used after the last card access to indicate that the application no longer needs the card.

After having obtained a SmartCard object, we have to create the card service we are going to use. We can then specify which file we want to access, and finally read its contents:

```
FileAccessCardService facs = (FileAccessCardService)
  sc.getCardService(FileAccessCardService.class, true);
CardFile root = new CardFile(facs);
CardFile file = new CardFile(root, ":c009");

byte[] data = facs.read(file.getPath(), 0,
                        file.getLength() );
```

Instances of the CardFile class are used to represent files or directories on the smart card, and to obtain information about them. In our example, we use two files. The first one, named root, is the master file, which can be compared to a root directory on a hard disk. Each file system-based smart card has such a unique master file. By passing only FileAccessCardService to the constructor of CardFile, we request a reference to the master file. The second CardFile instance is created by passing two arguments: a CardFile instance representing a directory and a relative path from that directory to the file in which we are interested. That file has the ID 0xc009 and resides immediately under the master file. The actual read operation shows how the information stored by CardFile can be used. getPath returns an absolute path to the file, while getLength returns its length in bytes. The second argument to the read operation is the index of the first byte, while the third one specifies the number of bytes to read. With this single method invocation, the full contents of the file are read and returned as a byte array. If you are using an Internet Broker demo card, you'll have to enter the password password when the read method is executed.

Now, the file contents are stored in a byte array named data. Since our application will no longer access the smart card, the SmartCard object can be closed as described above. We can then convert the byte array into a string, which will be printed to the standard output:

```
String entry = new String(data);
entry = entry.trim();
System.out.println(entry);
```

When creating a string from a byte array, the bytes are converted to characters using the JAVA platform's default encoding. The trim method removes leading and trailing whitespace from the string. This is required since only part of the file we read may contain actual data. The rest is padded with zeroes to prevent problems which might have otherwise been encountered in printing. Finally, the converted and trimmed contents of the file is printed. You may use the Internet Broker demo to verify that the output of your first sample application is correct.

Here is the complete listing of the first sample application. Although the detailed explanations may have confused you, it's not so long after all:

```
import opencard.core.service.SmartCard;
import opencard.core.service.CardRequest;
import opencard.opt.iso.fs.FileAccessCardService;
import opencard.opt.iso.fs.CardFile;
```

```
public class ReadFile {

  public static void main(String[] args)
  {
    System.out.println("reading smartcard file...");

    try {
      SmartCard.start();

      // wait for a smartcard with file access support
      CardRequest cr =
        new CardRequest(CardRequest.NEWCARD, null, FileAccessCardService.class);
      SmartCard   sc = SmartCard.waitForCard(cr);

      FileAccessCardService facs = (FileAccessCardService)
        sc.getCardService(FileAccessCardService.class, true);
      CardFile root = new CardFile(facs);
      CardFile file = new CardFile(root, ":c009");

      byte[] data = facs.read(file.getPath(), 0,
                              file.getLength() );
      sc.close();

      String entry = new String(data);
      entry = entry.trim();
      System.out.println(entry);

    } catch OpenCard Exceptions {
      e.printStackTrace(System.err);

    } finally { // even in case of an error...
      try {
        SmartCard.shutdown();
      } catch OpenCard Exceptions {
        e.printStackTrace(System.err);
      }
    }

    System.exit(0);
  }
}
```

Writing your second sample application

Now that you know how to read the cardholder data from an Internet Broker demo card, you may also want to change it. This will be the task of the second sample application. Assuming you are now familiar with initializing OCF, obtaining a SmartCard object, handling errors, and finally shutting down OCF again, the purpose of the following skeleton should be obvious. The class is called InitFile, since writing cardholder data is typically done during card initialization.

```
import opencard.core.service.SmartCard;
import opencard.core.service.CardRequest;
import opencard.opt.iso.fs.FileAccessCardService;
import opencard.opt.iso.fs.CardFile;

public class InitFile {

  public static void main(String[] args)
  {
    System.out.println("initializing file...");

    try {
      SmartCard.start();
      // wait for a smartcard with file access support
      CardRequest cr =
        new CardRequest(CardRequest.NEWCARD, null, FileAccessCardService.class);
      SmartCard   sc = SmartCard.waitForCard(cr);

      FileAccessCardService facs = (FileAccessCardService)
        sc.getCardService(FileAccessCardService.class, true);
      CardFile root = new CardFile(facs);
      CardFile file = new CardFile(root, ":c009");

      // here, we will write data to the smart card

    } catch OpenCard Exceptions {
      e.printStackTrace(System.err);
    } finally { // even in case of an error...
      try {
        SmartCard.shutdown();
      } catch OpenCard Exceptions {
        e.printStackTrace(System.err);
      }
    }

    System.exit(0);
  }
}
```

Let’s assume that the data to be written to the file has been passed as a command line argument. It is then available as the first string in the string array `args`. Since the Internet Broker demo stores multiple lines in the file, our application should be able to do the same. For the sake of simplicity, colons in the string are replaced with newlines. The application could then be invoked like this:

```
java InitFile "Klaus Mustermann:klaus@banana.com"
```

First, the string must be converted into a byte array. The `java.io` mechanism could be used for writing the string directly to the file, but that would not match the `read` operations in the Internet Broker demo and the first sample application. So this is how it’s done:

```
    String entry = args[0].replace(':', '\n');
    byte[] bytes = entry.getBytes();
    int   length = bytes.length;
```

In the first line, colons are replaced by newline characters. The second operation converts the string into a byte array, using the JAVA platform’s default character encoding. When taking a look at the first sample application, you will realize that the data has to be padded with zeroes to the full length of the file. Otherwise, data previously stored in the file would be only partially overwritten, resulting in an incorrect output. Note that the padding is a requirement of the sample applications, not of OCF. `FileAccessCardService` supports writing only parts of a file. The length of the file can be obtained from the `CardFile` instance named `file` using `getLength`. Since we cannot write more data than the file size specifies, the input is truncated if it was too long:

```
    int   length = bytes.length;

    byte[] data  = new byte [file.getLength()];
    if (data.length < length)
      length = data.length;
    System.arraycopy(bytes, 0, data, 0, length);
```

Now, the byte array named data holds the new contents of the file. It can be written by a single command, specifying an absolute path to the file to write, the target index in the file, and the data. If you are using the Internet Broker demo card, you’ll have to enter the password `password` when this method is invoked. For user convenience, we finally print what has been written to the file:

```
    // write the data to the file
    facs.write(file.getPath(), 0, data);

    System.out.println(entry);
```

Note that only one method invocation was required to write the data to the file. The rest has been done to prepare the data, which has nothing to do with OCF or smart cards. Here is the complete listing of the second sample application:

```
import opencard.core.service.SmartCard;
import opencard.core.service.CardRequest;
import opencard.opt.iso.fs.FileAccessCardService;
import opencard.opt.iso.fs.CardFile;

public class InitFile {

  public static void main(String[] args)
  {
    System.out.println("initializing file...");

    try {
      SmartCard.start();

      // wait for a smartcard with file access support
      CardRequest cr =
        new CardRequest(CardRequest.NEWCARD, null, FileAccessCardService.class);
      SmartCard   sc = SmartCard.waitForCard(cr);

      FileAccessCardService facs = (FileAccessCardService)
        sc.getCardService(FileAccessCardService.class, true);
      CardFile root = new CardFile(facs);
      CardFile file = new CardFile(root, ":c009");

      String entry = args[0].replace(':', '\n');
      byte[] bytes = entry.getBytes();
      int   length = bytes.length;

      byte[] data  = new byte [file.getLength()];
      if (data.length < length)
        length = data.length;
      System.arraycopy(bytes, 0, data, 0, length);

      // write the data to the file
      facs.write(file.getPath(), 0, data);

      System.out.println(entry);

    } catch OpenCard Exceptions {
      e.printStackTrace(System.err);

    } finally { // even in case of an error...
      try {
        SmartCard.shutdown();
      } catch OpenCard Exceptions {
        e.printStackTrace(System.err);
      }
    }

    System.exit(0);
  }
}
```

Chapter 3. OCF architectural concepts

This chapter provides an overview of the OpenCard architecture. We'll briefly explain the key concepts of the OpenCard Framework and put them in relation. We'll provide enough background to show how OCF brings together the various technology providers of the smart card industry, the application developers, the card terminal manufacturers, the card or card operating system manufacturers, and the card issuers. Finally, we'll introduce those concepts that are most relevant for application developers.

Architecture overview

We can divide the OpenCard Framework into two main parts: The *CardTerminal* layer and the *CardService* layer (see also Figure 1).

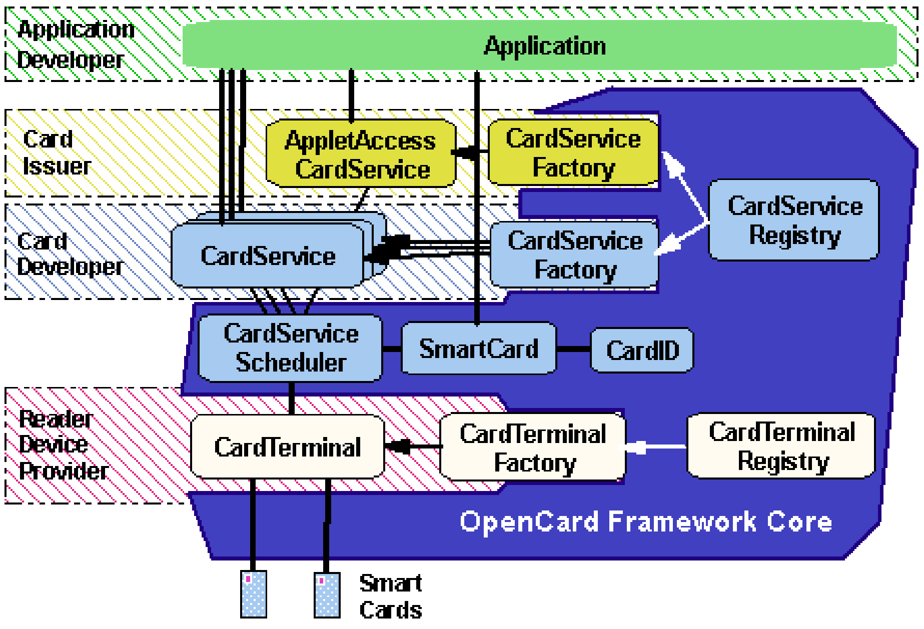

Figure 1. Main parts of the OpenCard Framework architecture.

The `CardTerminal` layer contains classes and interfaces that allow you, the application developer, to access card terminals and their slots. Using these classes you can, for example, find out whether a smart card is inserted in a card terminal.

The `CardService` layer defines the abstract `CardService` class: OCF represents smart card functions through card services. Each card service defines a particular (high-level) API that you can use to access some particular smart card function; for example, the file access card service gives you access to the file system of a smart card.

Both of OCF's layers — `CardTerminal` and `CardService` — are designed using the abstract factory pattern and the singleton pattern (as described in *Design Patterns — Elements of Reusable Object-Oriented Software*). The objects dealing with the manufacturer specific details are produced by a factory object which is supplied by the respective manufacturer. To determine which factory to use, OCF deploys a singleton called `registry`. A `registry` contains the configuration of an OCF component and creates the corresponding factory objects as needed.

We'll take a closer look at both layers, the `CardTerminal` layer and the `CardService` layer, in the next two sections.

The CardTerminal layer

There is a broad range of card readers on the market, with widely differing functions. Very simple card readers merely provide basic card input-output (I/O) functionality via a single slot to insert the card. More sophisticated card readers offer multiple slots or include a PIN pad and a display that can be used to perform cardholder verification. Card readers can attach to different I/O ports (e.g. serial ports and PC Card buses). Card readers come with the proper driver software for selected computer operating systems. OCF's `CardTerminal` layer provides adequate abstractions for the details of the various card terminals.

The classes of the `CardTerminal` layer serve a dual purpose: the most prominent function is to provide access to physical card terminals and inserted smart cards. This function is encapsulated in the `CardTerminal` class with its slots, and the `CardID` class.

In OCF, a physical card terminal is represented through the instances of the `CardTerminal` class. The *answer-to-reset* (ATR)[7] is represented in OCF through the `CardID` class. OCF allows for both static and dynamic configuration. In the case of static configuration, the configuration of card terminals is known *a priori* at system startup time. In the case of dynamic configuration, on the other hand, additional card terminals (e.g. PC Card-based terminals) can be added at run-time.

The `CardTerminal` class is an abstract superclass from which concrete implementations for particular card terminal types are derived. Each `CardTerminal` object contains one or more slots that represent the physical card slots of that card terminal. Access to a smart card that is inserted in a slot occurs through an exclusive gate object, the `SlotChannel` object: The `CardTerminal` class ensures that, at any given point in time, a maximum of one `SlotChannel` object per slot is instantiated. Thus, once an object has obtained a `SlotChannel` object, no other object can gain access to the associated smart card until that `SlotChannel` object has been released.

The `CardTerminal` class provides methods for checking card presence, obtaining a `SlotChannel` object, and sending and receiving APDUs. For card terminals offering additional functions — such as a display, a PIN pad, a finger print reader, or other input-output facilities — OCF provides additional interfaces that a `CardTerminal` can implement.

The `CardTerminal` layer also offers a mechanism to add and remove card terminals. The `CardTerminalFactory` class and the `CardTerminalRegistry` object implement this function. Each card reader manufacturer supporting OCF provides a `CardTerminalFactory` which "knows" about a particular family of card readers, and the respective `CardTerminal` classes. The system-wide unique

7. The ATR is the initial information emitted by a smart card upon being reset.

CardTerminalRegistry object keeps track of the installed card readers — as illustrated in Figure 2.

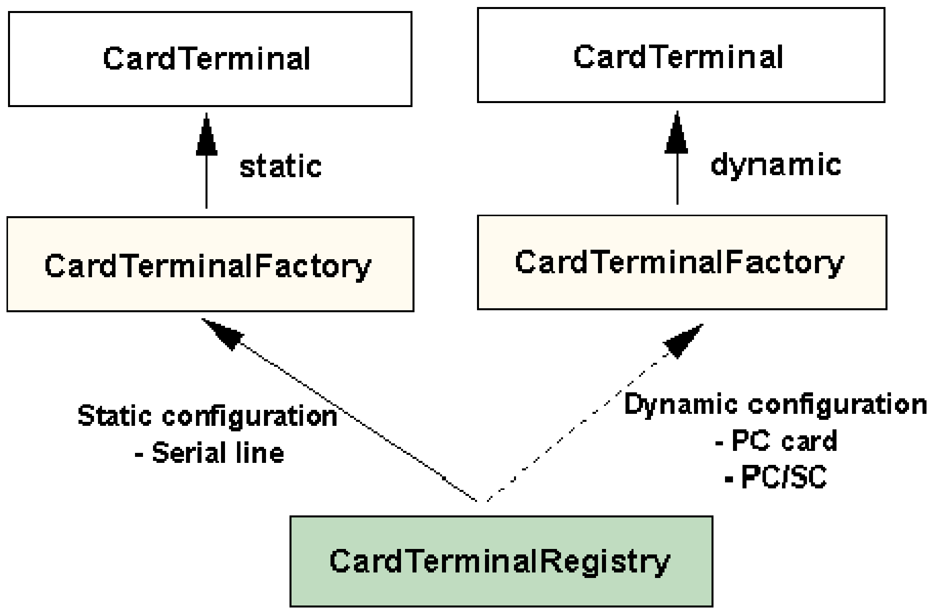

Figure 2. Classes of the `CardTerminal` *package.* OCF represents a real card terminal through the instances of the `CardTerminal` class with its slots. With OCF, both static and dynamic configurations are possible.

Furthermore, the CardTerminalRegistry object offers methods to register and unregister CardTerminal objects, and to enumerate all installed card terminals.

The CardTerminal classes generate events and notify the rest of the framework when a smart card is inserted into or removed from a card terminal (CardTerminal). These events are passed on to the EventGenerator by the CardTerminalRegistry (see Figure 3 on page 14).

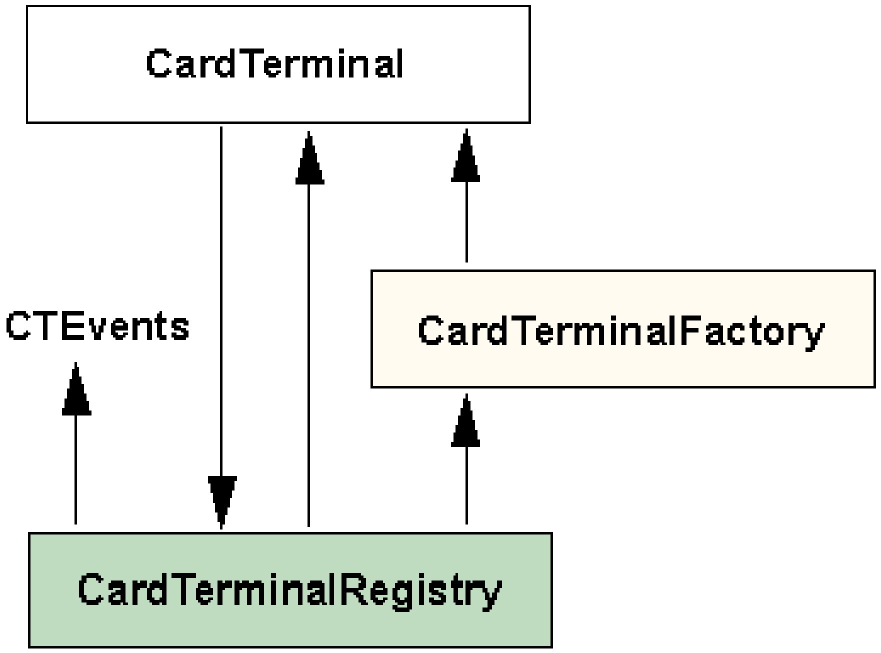

Figure 3. The `CardTerminal` layer. The `CardTerminalRegistry` keeps track of the available instances of `CardTerminalFactory` and `CardTerminal`. Each `CardTerminalFactory` keeps track of the `CardTerminal` it instantiated. Each `CardTerminal` manages its `slots` into which smart cards can be inserted. Insertion or removal of a smart card will trigger an event at the corresponding `CardTerminal`, which is passed on to the event package's `EventGenerator` by the `CardTerminalRegistry`. The dynamic installation of a `CardTerminal` does not trigger a special event. The events generated by the new terminal will be automatically passed to all listeners registered at the `EventGenerator`.

The CardService layer

Card services are the means by which the OCF makes smart card functions available to the application programmer. Therefore, they are the components that you probably will work with the most.

At the present time, the OpenCard Framework reference implementation defines only a few card service interfaces: `FileAccessCardService` and `SignatureCardService` are the most important ones. However, the number of standard card services will increase quickly, as it is the intention of the OpenCard Consortium to define card services for virtually all smart card functions.

FileAccessCardService

The electronic storage of information is often based on some sort of file system concept. Not surprisingly, most smart cards in use today offer tamper-proof storage via a file system abstraction[8].

8. *relational database* mechanisms or *object-based persistence* mechanisms are alternate abstractions for the storage of information and are considered for smart cards, too (see also the ISO standard *ISO 7816-7 Interindustry commands for Structured Card Query Language (SCQL)*.

`FileAccessCardService` is *OpenCard*'s abstraction for dealing with smart card based files. It offers a high-level interface modelled after JAVA's `java.io` package, making it virtually unnecessary for you to understand the nitty-gritty details of those standards.

SignatureCardService

Electronic signatures of data rely on cryptographic mechanisms that make use of cryptographic key information to perform their operation. A common scheme for signing data is the use of public key algorithms like RSA, which involves a *key pair* consisting of a *private* and a *public* key. Probably the most important application of smart cards is and will be their ability to serve both as a container for key (in particular, private key) information and as a processor for performing the cryptographic signing operation. In other words, the signature generation can be performed without the sensitive key information ever leaving the tamper-proof smart card.

`SignatureCardService` is a simple abstraction for the import, verification, and export of key information as well as for the generation and verification of digital signatures generated using a public key algorithm like RSA.

AppletAccessCardService and AppletManagerCardService

Smart cards can offer multi-function capabilities, i.e. a single card can support several applications. Accordingly, smart card applications will have to face situations where cards of the same or different type will host diverse sets of applets. There might even be smart card applets intended for the installation of new or removal of existing card-resident applets. It might also be necessary to suspend certain card-resident applets for a period of time. At the least, a smart card aware card-external application should be capable of finding out what card-resident applets a particular card hosts and presenting the card-holder a choice from which to select.

In order to address these requirements, card issuers who decide what applications are deployed or planned to be deployed with their cards put additional *meta-information* in the card that card-external applications can access to find out about - and, if necessary, change - the situation in any particular card.

These functions are handled by two card services: `AppletAccessCardService` and `AppletManagerCardService`. `AppletAccessCardService` is capable of listing applications, while `AppletManagerCardService` defines a high-level API through which applications can install and remove, card-resident applets in an issuer independent manner.

Chapter 4. Programming with OCF

Configuring the OpenCard Framework

The OpenCard Framework reference implementation obtains configuration information via the Java system properties. Java system properties are a platform-independent mechanism to make operating system and run-time environment variables available to programs. The Java run-time environment defines a set of default properties that are always part of the system properties. Applications can extend this set of properties by *loading* additional properties and merging them with the system properties. In this way, OpenCard Framework configuration parameters are added to the system properties.

Configuring the registries

The system-wide `CardTerminalRegistry` and `CardServiceRegistry` keep track of the instances of `CardTerminalFactory` and `CardServiceFactory`, respectively. In order for the OpenCard Framework reference implementation to be meaningful, each registry must know at least one factory. When the framework starts up, the `Card[Terminal|Service]Registry`s are initialized by the system based on properties that the user defined and added to the system properties.

You can configure `CardTerminalRegistry` via the `OpenCard.terminals` property and `CardServiceRegistry` via the `OpenCard.services` property.

The syntax of the property string for either property is as follows:

```
<record-0> <record-1> ... <record-N>
```

where records are separated by a white-space and each record consists of a class name and optional string parameters separated by a ″|″, i.e.

```
class-name|param-1|param-2| ... |param-N
```

The following example illustrates how the property string might look:

Property name:
`OpenCard.services`

Property string:
`com.ibm.opencard.factory.MFCCardServiceFactory`

Property name:
`OpenCard.terminals`

Property string:

```
com.ibm.opencard.terminal.ibm5948.IBMCardTerminalFactory|IBM5948-1|IBM5948-B02|1\
com.ibm.opencard.terminal.pcsc10.Pcsc10CardTerminalFactory
```

Configuring tracing output

The OpenCard Framework reference implementation is comprehensively instrumented to produce detailed tracing information at run-time. This provides valuable feedback about the operation of the framework and helps to locate problems should they occur. You can flexibly adjust the amount of tracing information produced by setting the `OpenCard.trace` system property.

The utility class `opencard.core.util.Tracer` distinguishes between the following trace levels:

EMERGENCY	**System is unusable; numeric value is 0**
ALERT	Action must be taken immediately; numeric value is 1
CRITICAL	Critical condition; numeric value is 2
ERROR	Error condition; numeric value is 3
WARNING	Warning condition; numeric value is 4
NOTICE	Normal but significant condition; numeric value is 5
INFO	Informational; numeric value is 6
DEBUG	Debugging information; numeric value is 7
LOWEST	Even more details; numeric value is 8

The syntax of the property string is as follows:

```
<package-prefix:trace-level> <package-prefix:trace-level> ... <package-prefix:trace-level>
```

where `package-prefix` specifies the prefix of a package or class name and `level` specifies a number between 0 and 8 corresponding to the desired trace level.

The following example illustrates how the property strings might look:

Property name:
`OpenCard.trace`

Property string:

```
opencard.core:6 opencard.opt:0 com.ibm.opencard:3 \
com.ibm.opencard.terminal.ibm5948.IBM5948CardTerminal:8
```

OpenCardPropertyFileLoader

The OpenCard architecture does not define the mechanism by which the system properties are extended because there is no single mechanism that is guaranteed to be available on all platforms on which the OpenCard Framework reference implementation will run. However, the OpenCard Framework reference implementation does provide a utility class called `opencard.opt.util.OpenCardPropertyFileLoader` to load properties from a file. This file-based properties loading mechanism is the default used to extend the system properties. Details on how to change this mechanism are described in "Loading properties with your own mechanism" on page 19.

Now let's take a look at how the default properties loader works. `OpenCardPropertyFileLoader` looks in the following places for property files (in the order given):

1. `[java.home]/lib/opencard.properties`
2. `[user.home]/.opencard.properties`
3. `[user.dir]/opencard.properties`
4. `[user.dir]/.opencard.properties`

where `[xxx.xxx]` are the respective path variables as defined in the default system properties. It loads the properties from each of these files, in turn merging them with the system properties. If the properties file being read contains a property name that has already been defined in the properties set, the new definition will be ignored by default.

In case you wish to override a property that is already defined in the properties set, you can do this by defining the property name `name` anew and adding an additional property with the name `name.override` that has the value `true` to your property file.

The following example will set the property `OpenCard.property` to the value `new_value` no matter what its value was before:

```
# Overriding a possibly already defined OpenCard property
# with a new value
    OpenCard.property = new_value
    OpenCard.property.override = true
```

Using the same properties strings as in the examples above, your properties file might look like this:

```
# Configuring the CardServiceRegistry
OpenCard.services = com.ibm.opencard.service.MFCCardServiceFactory

# Configuring the CardTerminalRegistry
# The parameters after the class name are:
#       name, type, and address of the card terminal
OpenCard.terminals = \
  com.ibm.opencard.terminal.ibm5948.IBMCardTerminalFactory|IBM5948-1|IBM5948-B02|1 \
  com.ibm.opencard.terminal.pcsc10.Pcsc10CardTerminalFactory

# Configuring Tracing
OpenCard.trace = opencard.core:6 opencard.opt:0 com.ibm.opencard:3 \
                 com.ibm.opencard.terminal.ibm5948.IBM5948CardTerminal:8

# Overriding OpenCard.property
OpenCard.property = new_value
OpenCard.property.override = true
```

Loading properties with your own mechanism

The default property loading mechanism implemented by the `opencard.opt.util.OpenCardPropertyFileLoader` class may not be adequate in all environments. For instance, it cannot be used on platforms that do not have a file system.

You can implement a different property loading that fits your purpose or platform better and have it replace the default mechanism. In order for your property loader to integrate with the framework, it must implement the `opencard.core.util.OpenCardConfigurationProvider` interface.

You must tell the framework to use your class instead of the default loader. You do that via the following system property:

Property name:
: `OpenCard.loaderClassName`

Property string:
: `fully-qualified/class/name`

Note:
Obviously, you must set this property *prior* to the OpenCard properties loading mechanism being invoked, for instance via a command line flag of the Java interpreter or, if applicable, in the system-wide `.properties` file.

Initializing the OpenCard Framework

The OpenCard Framework reference implementation must be initialized at startup time. You must therefore invoke the start() method of the SmartCard class in any application that uses OCF. Before the application finishes, the shutdown method of the SmartCard class should be invoked, too. The code to do has the following appearance:

```
try {
  SmartCard.start();

  // use OCF...

} catch OpenCard Exceptions {
  // handle error...

} finally {
  try {
    SmartCard.shutdown();
  } catch OpenCard Exceptions {
    // handle error...
  }
}
```

Waiting for a smart card

The SmartCard object is the pivotal object for interacting with a physical smart card. So now that you have initialized the framework, you probably may want to obtain a SmartCard object. One way to achieve this is to call the waitForCard method on the SmartCard class, which returns a SmartCard object.

The waitForCard method takes as an argument a CardRequest object that lets you further specify the details regarding what card you are looking for and how long you intend to wait. In particular, you can specify that

- *any* smart card is acceptable, including a card that has already been inserted
- only a *newly* inserted smart card is acceptable
- the smart card's *Answer-to-Reset* response must satisfy an arbitrary condition, specified by an instance of CardIDFilter
- the smart card must support a given CardService class or interface
- the smart card must be inserted into a particular card terminal
- the call should return after a given time-out value

These details can be specified via parameters passed to the CardRequest class constructors and/or by invoking the corresponding CardRequest class methods.

Let's assume that we're interested in a smart card supporting FileAccessCardService. We could then write the following code:

```
SmartCard card = SmartCard.waitForCard(
  new CardRequest(CardRequest.NEWCARD, null, opencard.opt.iso.fs.FileAccessCardService.class));
```

The waitForCard method is a synchronous, *application-driven* model of gaining access to a SmartCard object. Alternately, you may prefer an *event-driven* model where your application receives a notification from the framework at the occasion of card insertion. This model is more suited for graphical applications. It is explained in "Getting a smart card via event-notification" on page 32.

Obtaining a card service

Card services are the means by which smart card functions are made available to the application. Thus, the next step is to obtain the proper instance of a given card service.

The actions you have to take depend on the nature of the application. We distinguish between two types of applications: (1) the ones that are prepared to work with only one particular card-resident application, and (2) the ones that can interact with a variety of different (at least two) card-resident applications.

In the first situation, you have probably specified the CardService class that you already required at the time you issued the waitForCard call. The code fragment to obtain e.g. FileAccessCardService looks like this:

```
FileAccessCardService facs = null;
try {
  facs = (FileAccessCardService)
    card.getCardService(FileAccessCardService, true);
} catch OpenCard Exceptions {
  // handle error...
}
```

You're now ready to start working with the requested card service. We'll show you how to work with some of the available card services in subsequent sections.

In the situation where your application can interact with several different card-resident applications, you will have to find out what card-resident applications are present. For that purpose, you must obtain AppletAccessCardService, which has a list method that lets you obtain the list of applications on the card. In this case, the code looks something like this:

```
AppletAccessCardService aacs = null;
try {
  aacs = (AppletAccessCardService)
    card.getCardService(AppletAccessCardService, true);
} catch OpenCard Exceptions {
  // handle error...
}
```

You're now ready to write the code to select a particular application in the card. We'll show you how to do that in the next section.

Handling exceptions

It is important that provisions be made in every code for dealing with exceptional circumstances, and OpenCard Framework features a number of corresponding methods for responding appropriately to unusual situations.

At the root of OCF's exception hierarchy are two classes: opencard.core.OpenCardException and opencard.core.OpenCardRuntimeException. The first is a subclass of Java's IOException and is therefore checked, while the second is a subclass of RuntimeException and is thus unchecked. The distinction is that the latter need not be caught or declared to be thrown (i.e. the compiler does not force you to handle these exceptions - hence the term 'unchecked exception'). The rationale behind this concept is that exceptions of this kind disrupt the normal program flow so severely that continuing would not make much sense. Further, from a pragmatic point of view, it would often be impractical to check for these conditions (an example of this is OutOfMemoryError, which can be thrown practically anywhere in code where objects are created dynamically). Nevertheless, all other exceptions should be dealt with in order to ensure program robustness.

The OCF exception hierarchy branches into two major trunks in the `terminal` and `service` sub-packages. The exception classes in the `terminal` package all extend `CardTerminalException`, which is itself a subclass of `OpenCardException` (and therefore checked). In the `service` package, we have two base exception classes: the `CardServiceException` class and the `CardServiceRuntimeException` class. The `CardServiceException` class extends `OpenCardException`, while the `CardServiceRuntimeException` class extends `OpenCardRuntimeException`. The latter has only two subclasses, which means that the application developer must handle the vast majority of OpenCard's exceptions.

Throughout this guide, for the sake of brevity, a 'catch-all' exception handler has been used in the code samples. However, this is not a style to be recommended in real-world code. So if specific exceptions are thrown in your code, catch them using a specific handler. The most general OCF sample in this guide looks like this:

```
try {
  SmartCard.start ();      // get OpenCard up and running
                           // here comes your client code
  SmartCard.shutdown ()    // cleanup OpenCard
}
catch (Exception e) {
// catch-all handler
// ...
}
```

When looking at the declaration of `SmartCard.start`, it becomes apparent that this method declares four exceptions that may be thrown:

- `OpenCardPropertyLoadingException`
- `ClassNotFoundException`
- `CardServiceException`
- `CardTerminalException`

As you now know, the complete sample above should be written with specific exception handlers as in the following:

```
try {
  SmartCard.start ();      // get OpenCard up and running
                           // here comes your client code
  SmartCard.shutdown ();
  // cleanup OpenCard
}
catch (OpenCardPropertyLoadingException plfe) {
  // ...
}
catch (ClassNotFoundException cnfe) {
  // ...
}
catch (CardServiceException cse) {
  // ...
}
catch (CardTerminalException cte) {
  // ...
}
```

This code is now less concise than the first sample presented above. Further, it takes longer to write (and to compile). But it is precise (it is obvious what this piece of software does and where it may fail) and specific. When an exception is thrown, the user will know exactly what kind of exception it was. This is hence the proper way to deal with specific exceptions.

Listing applications

Card-resident applets are described by *meta-information* encapsulated in AppletInfo objects. This meta-information includes the user-friendly *label* and the *AppletID*.

The term "applet" as we use it in this documentation can be either a program, which is executed on the card itself (like a typical Java Card applet). Or it can be an application specific set of files and directory structures (as we will find on ISO compliant file system oriented cards). In both cases the AppletID is used to identify the applet in a world-wide unique way. There are standards which define how each smart card should hold a directory of AppletIDs, allowing any card-external application to decide, if it supports a particular card or not

You can obtain the AppletInfos from your SmartCard object using AppletAccessCardService's list method.

```
AppletInfo [] templates = null;
try {
  AppletAccessCardService aacs = null;
  aacs = (AppletAccessCardService)
       card.getCardService(AppletAccessCardService, true);
  templates = aacs.list();
  // evaluate templates, e.g., present the user a list
  // of applet names from which to select
  // ...

} catch OpenCard Exceptions {
  // handle exception
  ...
}
```

Given the array of AppletInfo objects, you can prepare a list with applet names from which the user or the card-externmal application selects the applet he wants to work with. That application will then use the respective AppletInfo object to locate itself on the smart card. How applets defined by the AppletInfo objects are mapped to the corresponding card-external applications is beyond OCF's scope.

Using the AppletInfo retrieved by the the AppletAccessCardService the correct Applet can now be selected. For example an ISO compliant card can use the AppletID within the select-by-AID command to select the directory which corresponds to a particular application. A Java Card can use the AppletID to select a appropriate Java Applet, which can be executed on the card.

So your code might look something like this:

```
try {
  // ...
  AppletID aid = template[0].getAid());
  CardFilePath filePath = new CardFilePath(aid.toString());
  // ...
} catch OpenCard Exceptions {
  // handle exception
  ...
}
```

Working with files

Files in the smart card are accessed via FileAccessCardService. We have already explained how an application obtains an instance of this service in a previous section. In this section, we'll take a look at how FileAccessCardService works and explain how to use the additional classes that come along with it.

FileAccessCardService

FileAccessCardService provides a variety of methods to obtain information about the files in the smart card and to read and update them. The service operates stateless. This means that there is no such thing as a current directory or an open file. Instead, all methods that operate on a particular file expect a CardFilePath argument that identifies the target file. Information about a file in the smart card is provided by an instance of the CardFileInfo interface. This interface defines methods to query the kind of file, for example whether it is a directory, a transparent data file, or a structured data file. Other methods provide more detailed information, like the size of the file. Here is a code fragment that shows how to obtain file information:

```
try {
  CardFilePath rootpath = facs.getRoot()
  CardFilePath filepath = new CardFilePath(rootpath);
  filepath.append(new CardFilePath(":c009"));

  CardFileInfo rootinfo = facs.getFileInfo(rootpath);
  CardFileInfo fileinfo = facs.getFileInfo(filepath);
} catch OpenCard Exceptions {
  // handle error...
}
```

First, FileAccessCardService is used to obtain the path to the master file on the smart card, which is stored in rootpath. Then, a copy of that path is extended by a relative path to the file with an ID of 0xc009. The result is an absolute path to that file, which is stored in filepath. The following two invocations of FileAccessCardService return the information about these two files.

The CardFilePath class

Dealing with CardFilePath is tricky and error-prone, since CardFilePath objects are mutable and have to be copied frequently to ensure proper operation of the application. Aside from that, it is inconvenient to use two attributes to store the path to and the information about the same file. The CardFile class combines these two attributes, as well as providing several constructors that simplify path composition. Using the CardFile class, the code fragment of the previous section would look like this:

```
try {
  CardFile root = new CardFile(facs);
  CardFile file = new CardFile(root, ":c009");
} catch OpenCard Exceptions {
  // handle error...
}
```

The first CardFile object is created by simply passing FileAccessCardService to the constructor. The constructor will use getRoot to obtain the path to the master file, and invoke getFileInfo to obtain the information about this file. Apart from the path and information about the file, the CardFile object also stores the instance of FileAccessCardService that was used to create it.

The second CardFile object is created by passing a parent directory in form of another CardFile object, and a string that contains a relative path from the parent directory to the file to represent with the new CardFile object. The constructor will copy the path of the parent directory, append a relative path created from the string, and invoke getFileInfo to obtain the information about this file. FileAccessCardService used is the one that is stored in the parent CardFile.

Having created a CardFile object, the path to the represented file can be obtained using getPath. Since CardFile implements the CardFileInfo interface, the methods defined there can be invoked directly at the CardFile object. If there is a need to have the CardFileInfo object that was returned by FileAccessCardService —for example to downcast it to an implementation specific type —it can be obtained by invoking getFileInfo at the CardFile object.

The CardFileInput/OutputStream classes

The java.io package defines classes and interfaces to access files in a traditional file system, for example on a hard disk. File system-based smart cards typically support various file types. One of these are transparent files, which are comparable to files in a traditional file system. The CardFileInputStream and CardFileOutputStream classes extend the java.io classes named InputStream and OutputStream, respectively. They can be used to access transparent files in a smart card through the java.io mechanisms. Their constructors expect a CardFile object that specifies the transparent file to access as well as the instance of FileAccessCardService to use.

In a similar way, the CardFileReader and CardFileWriter classes extend the java.io classes named InputStreamReader and OutputStreamWriter, respectively. These classes provide file access on a character basis instead of on a byte basis.

The CardRandomRecordAccess class

The CardRandomRecordAccess and CardRecord classes provide a high level mechanism for accessing linear structured files of either fixed or variable record size. CardRecord is used to encapsulate the data stored in the records. CardRandomRecordAccess defines a file pointer to a record in the structured file and provides methods to read and write single records or arrays of records. Its constructor expects a CardFile object that represents the linear structured file and specifies the particular instance of FileAccessCardService to use for accessing it.

Note:
CardRandomRecordAccess can be used only for files with a linear structure. Transparent files are not split into records; they can be accessed using the classes described in the previous section. On the other hand, files with a cyclic structure do not support random write access and may use a different way of addressing records when reading. Currently, there is no high level mechanism for accessing cyclic files.

Signature generation and verification

The generation and verification of signatures in the smart card is performed via the following signature card services: SignatureCardService, KeyImportCardService, and KeyGenerationCardService. The functionality is distributed into these three interfaces to allow a card service implementer to implement only that subset of signature functionality which is supported by the specific card. We have already explained how an application obtains a particular instance of CardService in a previous section. In this section, we'll take a look at the abstractions that the signature card services provide and explain how they're used.

Generating signatures

To generate/verify a signature, a card service implementing interface called SignatureCardService is needed. Signature generation on a message is performed by computing a hash value on the message and then encrypting the hash value using the private key of a public key algorithm. In the example below, the RSA public key algorithm is used.

SignatureCardService offers the signData method to perform this operation. In addition to the message to be signed, the method needs to know which hash algorithm and public key algorithm to use and which key to use for signing. The private key reference for file-oriented cards consists of a directory CardFilePath object and an integer specifying the number of the key relative to the directory.

The code you have to write looks something like this

```
byte[] message = ...;
byte[] signature = null;
SignatureCardService scs = null;
try {
  // allocate card service
  scs = (SignatureCardService)
    card.getCardService(SignatureCardService, true);
  // create the private key reference
  CardFilePath path = new CardFilePath(":3F00:C200");
  PrivateKeyFile keyRef = new PrivateKeyFile(path,0);
  // generate signature
  signature = scs.signData(keyRef,"SHA1withRSA",message);
  } catch OpenCard Exceptions {
  // handle exception
  ...
}
```

Verifying signatures

Signature verification on a message is performed by computing the decryption on the given signature using the public key, computing a hash value on the plain message, and comparing the hash value to the decrypted signature. SignatureCardService offers the verifySignedData method to perform this operation. In addition to the plain and cipher message to be verified, the method needs to know the hash algorithm and public key algorithm to be used and which key to use for signing. The public key reference for file-oriented cards consists of a directory CardFilePath object and an integer specifying the number of the key relative to the directory. The code you have to write looks something like this:

```
byte[] message = ...;
byte[] signature = ...;
SignatureCardService scs = null;
try {
  // wait for card and obtain the card service
  // ...
  // create the public key reference
  CardFilePath path = new CardFilePath(":3F00:C200");
  PublicKeyFile keyRef = new PublicKeyFile(path,1);
  // verify signature
  boolean result = scs.verifySignedData(keyRef,"SHA1withRSA", message, signature);
  } catch OpenCard Exceptions {
  // handle exception
  ...
}
```

Instead of performing both of these operations (the hashing operation and the public key algorithm) on the card, it is possible to perform the time-consuming hashing of long messages outside the OpenCard Framework. To do this, SignatureCardService offers two methods: the SignHash() method and the VerifySignedHash() method, respectively. Computing the hash value is then the responsibility of the application, and can be performed using classes like Sun's MessageDigest in the java.security package or subpackages. It may also make sense to perform signature verification (which does not involve any private keys) off card using the java.security package. Wherever the private key is needed (for example when performing signature generation), a smart card offers tamper-proof storage to protect the private key.

Importing keys

In addition to generating and verifying signatures, KeyImportCardService (which extends the SignatureCardService interface) offers methods for the importation and subsequent in-card verification of keys for asymmetric key algorithms. The non-validating methods to import keys are importPrivateKey and importPublicKey. In addition to the actual key data and key information, these methods take as parameters a key reference to identify the storage location of the imported key. The key reference for file-oriented cards consists of a directory CardFilePath object and an integer specifying the number of the key relative to the directory. The corresponding validating methods are importAndValidatePrivateKey and importAndValidatePublicKey. These methods make the smart card check the integrity of the imported key by validating a signature of the key, which is passed as an additional parameter, using (another) indicated key for signature verification. The smart card accepts the imported keys subject to successful verification. The code to import and validate a private key might look something like this:

```
RSACRTKey rsaPrivate = ...;
byte[] signature = ...;
byte[] keyInfo = ...;
SignatureCardService scs = null;
try {
  // wait for card and obtain the two card services
  // ...
// create the private key reference for the key to be imported
  CardFilePath path1 = new CardFilePath(":3F00:C110");
  PrivateKeyFile targetKey = new PrivateKeyFile(path1,0);
  // create the public key reference for the key to verify the signature
  CardFilePath path2 = new CardFilePath(":3F00:C200");
  PublicKeyFile valKey = new PublicKeyFile(path2,1);
  boolean result = scs.importAndValidatePrivateKey(targetKey,
                  rsaPrivate, keyInfo, signature, valKey);
  // ...
} catch OpenCard Exceptions {
  // handle exception
  ...
}
```

Generating keys

In addition to working with signatures, KeyGenerationCardService — a further extension of the SignatureCardService interface — also offers methods for generating a key pair for a public key algorithm on the card and for reading the public key part of the pair for usage outside of the card.

The method for generating keys is called generateKeyPair. In addition to the storage destinations for the private and public keys, this method takes as parameters the required key strength and key algorithm. A specific card may not support the required strength and algorithm, and the card service representing the card may throw an appropriate exception.

The method for reading the public key is called readPublicKey. In addition to the key reference for the key to be read, this method takes the required key algorithm as a parameter. No method is available for reading the private key since the private key should never leave the card.

The code for generating a RSA key pair and reading the public key might look like this:

```
RSAPublicKey rsaPublic = null;
KeyGenerationCardService kgs = null;
try {
  // wait for card and obtain the card service
  // ...
  // create the key references for the keys to be generated
  CardFilePath path = new CardFilePath(":3F00:C110");
  PrivateKeyFile privKey = new PrivateKeyFile(path,0);
  PublicKeyFile pubKey = new PublicKeyFile(path,1);
  kgs.generateKeyPair(privKey, pubKey, 512, "RSA");
  // read the public key from the card
  rsaPublic = (RSAPublicKey) kgs.readPublicKey(pubKey, "RSA");
  // ...
  } catch OpenCard Exceptions {
  // handle exception
  ...
}
```

Chapter 5. Advanced programming with OCF

In this chapter, we'll show you some more advanced programming concepts of the OpenCard Framework.

Working with OpenCard events

OpenCard uses the Observer design pattern to inform listeners about events that happen in the framework.

In the OpenCard Framework reference implementation, there are a number of objects that can be the source of `OpenCardEvent` objects that other framework components or the application itself might be interested in. The implementation of the event notification is based on `EventObject` class and `EventListener` interface of the `java.util` package.

Objects that are interested in receiving event notifications must implement some extension of the `EventListener` interface and register with the proper object by adding themselves as listeners.

As an application programmer, you may be interested in the following types of OpenCard Framework events:

- events signalling that a card was inserted or removed into a card terminal,
- events signalling that a tracer issued a tracing message.

CardTerminalEvent

In previous versions of OCF the `CardTerminalRegistry` was responsible to emit card insertion / removal events. This function is now moved to class `EventGenerator` in the events package to have a clearer separation between the events and terminal package. The EventGenerator continuously monitors the state of physical card reader devices and emits `CardTerminalEvents` if the state changes.

The `CTListener` interface defines two methods, `cardInserted()` and `cardRemoved()`, through which `CardTerminalEvents` can be communicated to objects interested in these events.

You can register objects that are interested in receiving `CardTerminalEvents` provided they implement the `CTListener` interface. You can register your `CTListener` with the system-wide `EventGenerator` instance using the `addListener()` and `removeListener()` methods. The registered objects will then receive all `CardTerminalEvents` produced by all currently-registered `CardTerminal` objects.

Locking a CardTerminal

Once you hava a SmartCard object you can gain exclusive access using the `SmartCard.beginMutex()` methods. This is described under "Gaining exclusive access to the smart card" on page 35

There are other scenarios like transferring money from one smartcard to another one where you need to maintain a lock on an individual slot or the whole CardTerminal across card insertions and card removals.

To lock a terminal you can use the methods declared in the `opencard.opt.terminal.Lockable` interface, as demonstrated in the following example:

```
// lock terminal
Enumeration terminals =CardTerminalRegistry.getRegistry().getCardTerminals();
CardTerminal t = (CardTerminal)terminals.nextElement();
Lockable terminal = null;
Object handle = null;
if (t instanceof Lockable) {
   terminal = (Lockable)t;
   System.out.println("Locking terminal "+terminal);
   handle = terminal.lock();
}
// now request SmartCard objects and communicate with cards
```

The lock on a CardTerminal must be obtained before requesting any SmartCard object for the terminal. It will throw an exception, if another process (outside of the JVM running the OCF application) or another thread using OCF already uses the CardTerminal.

To release a lock on a locked CardTerminal first destroy all SmartCard objects before calling the unlock function as the following example shows

```
SmartCard sc = ...
// close down the SmartCard object and unlock the terminal
sc.close();
sc = null;
if (handle!=null) {
   terminal.unlock(handle);
}
```

The OpenCard Reference implementation comes with a lockable CardTerminal implementation for PCSC card readers (see `com.ibm.opencard.terminal.pcsc10.Pcsc10LockableTerminal`. Since the Lockable interface is an optional interface, not all CardTerminals will be lockable. To see how to implement a lockable CardTerminal see “Implementing the Lockable interface” on page 57

For more details on CardTerminal Locking see `RFC 17-1 OCF Terminal Locking Mechanism` on the OpenCard website.

TracerEvents

For an explanation of `TracerEvents`, please see “Using the trace utility” on page 40.

Using optional CardTerminal functions

For many applications, it will be sufficient to work with the `SmartCard` object and `CardService` objects to interact with the smart card. However, some applications may want to take advantage of special features that some card reader devices offer, such as using the display or the PIN pad integrated with the reader.

`CardTerminal` objects expose additional features by implementing corresponding interfaces. The OpenCard Framework reference implementation defines a number of these interfaces in the `opencard.opt.terminal` package. They include:

`UserInteraction`
This interface comprises the `display()` method to present a message to the cardholder, the `clearDisplay()` method to erase the message from the display, the `keyboardInput()` method to collect a string entered by the

cardholder, and the promptUser() convenience method, which combines displaying the message to and collecting input from the cardholder into a single call.

PowerManagementInterface
: This interface comprises the powerUpCard() and powerDownCard() methods, both of which take the slot number as a parameter and allow the supply of power to the smart card in the designated slot to be controlled. This function is useful with long lasting applications that only sporadically interact with the smart card in environments where low power consumption matters, e.g. PDAs and hand-held computers.

TerminalCommand
: This interface comprises the sendTerminalCommand() method, which takes a byte array and sends it to the card reader. Assuming an application knows the set of commands that a given card reader supports, it can use this interface to control the card reader.

The CardTerminal class offers the features() method, which returns a Properties object containing descriptive property strings for a particular terminal. These properties may be used to indicate to the application what optional features a particular CardTerminal object supports.[9]Alternately, an application can use the introspection API to find out whether the particular CardTerminal supports a given interface.

Let's assume that you would like to make use of the UserInteraction function in your application. This is the code you would use:

```
try {
  // initialize framework
  SmartCard.start();
  // wait for a card to be inserted
  // any new card will do; any reader will do
  SmartCard card = SmartCard.waitForCard(new CardRequest());

  // get the card terminal in which the card was inserted
  CardTerminal terminal = card.getCardID().getSlot().getCardTerminal();

  String name = null;

  // check whether it supports UserInteraction
  if (terminal instanceof UserInteraction.class) {
    // we can prompt user using the terminals I/O capabilities
    // assuming we accept all letters, allLetters might be a string containing
    //     all characters in upper and lower case plus blank
    // assuming a LF or CR terminate the input, terminator might be a
    //     string containing "\n\r"
    name = terminal.promptUser("Please enter your name:",
                 new CardTerminalIOControl(0, 60, allLetters, terminator));
  } else {
    // put code here to prompt user in a different way, e.g. through a GUI
    // ...
}

  // do something with the name
  if (name != null) {
  // put your code here
  }
```

9. At present, we still lack a standard set of properties to characterize card terminals beyond the pure terminal configuration data. This is needed if an application wants to find out what optional functions a card reader supports.

```
} catch OpenCard Exceptions {
  // exception handling
  // ..
}
```

Similar code can be used to make use of the other interfaces (`PowerManagementInterface` and `TerminalCommand`).

Getting a smart card via event-notification

In “Waiting for a smart card” on page 20, we showed you how you gain access to a `SmartCard` object using the application-driven programming model based on the synchronous `waitForCard` call. In this section, we will introduce an event-driven programming model to accomplish the same based on the event notification mechanism introduced in “Working with OpenCard events” on page 29.

In the event-driven programming model, an application suspends its execution after initialization. The execution resumes when the application receives an event that has to be processed. Events can be received from various sources. For example, there may be timer events that trigger regular actions, like auto-saving in an editor. Other events can originate from network connections, indicating external request. The kind of events we are interested here are the result of user input. The JAVA ABSTRACT WINDOW TOOLKIT (AWT) sends events when the user moves the mouse, or presses a key on the keyboard. Likewise, the OPENCARD FRAMEWORK sends events when the user inserts or removes a smart card. The key advantage of the event-driven paradigm is that a single application thread is sufficient to process events from various sources. In the application-driven paradigm, a thread has to block on a single source of input, for example the `waitForCard` call that is used to wait for insertion of a smart card.

In order to receive events, an application has to register at a source of events. In OCF, the central source for events is `EventGenerator`. An object implementing the `CTListener` interface has to be passed to the registry, which will invoke the methods of that object at the appropriate time. The argument to each invocation is a `CardTerminalEvent` that includes details about what happened where. Take a look at this example:

```
import opencard.core.event.CTListener;
import opencard.core.event.CardTerminalEvent;
import opencard.core.event.EventGenerator;
import opencard.core.terminal.Slot;
import opencard.core.terminal.CardTerminal;
import opencard.core.service.SmartCard;

public class Listener implements CTListener {
  private SmartCard smartcard = null;
  private CardTerminal terminal=null;
  private int slotID = 0;


  public void register()
  {
    EventGenerator.getGenerator().addCTListener(this);
    try {
      EventGenerator.getGenerator().createEventsForPresentCards(this);
    } catch (Exception e) {
      e.printStackTrace(System.err);
    }
  }

  public void unregister()
```

```
    {
      EventGenerator.getGenerator().removeCTListener(this);
    }

    public void cardInserted(CardTerminalEvent event)
    {
      if (smartcard == null) {
        try {
          smartcard = SmartCard.getSmartCard(event);
          terminal = event.getCardTerminal();
          slotID = event.getSlotID();
        } catch OpenCard Exceptions {
          // handle error...
        }
      }
    }

    public void cardRemoved(CardTerminalEvent event)
    {
      if ((event.getSlotID() == slotID) &&(event.getCardTerminal()==terminal)) {
        smartcard = null;
        terminal = null;
        slotID = null;
      }
    }
  }
```

The class implements the interface CTListener, which requires the cardInserted and cardRemoved methods. The cardInserted method is invoked when a smart card is inserted into one of the slots of one of the terminals. In the example, this method first checks whether there is already a SmartCard object. If there is none, one is created by invoking getSmartCard. This is the non-blocking counterpart of waitForCard. Optionally, CardRequest can be passed as a second argument. If that request is not satisfied by the smart card that has just been inserted, getSmartCard returns null. After creating the SmartCard object, the terminal and slot where the card is located is stored. The cardRemoved method is invoked when a smart card is removed from one of the slots. In the example, it is checked whether the card has been removed from the slot for which a SmartCard object has been created by means of the cardInserted method. If so, that SmartCard object is thrown away, since it cannot be used anymore. If not, the event is simply ignored.

Now that we know what happens when an event occurs, how can we make sure that the events are passed to our Listener object? That's what register does. It registers Listener at CardTerminalRegistry by invoking addCTListener. From then on, every card insertion or removal results in an invocation of the cardInserted and cardRemoved methods. There remains one problem: What about smart cards that have been inserted before our listener was registered? We may want to know about them, too. This is achieved by invoking the createEventsForPresentCards method. This method scans all terminals and slots, and creates events for any cards present in the slots. These events are not sent to just any event listener, but instead only to the one that is passed as an argument. That way, other applications will not receive card insertion events for cards they already know of. Since scanning the slots may result in exceptions being thrown, some error handling has to be done at this point. If a card is inserted between registering and the following invocation of createEventsForPresentCards, the listener may get two events for that card. Double events can be filtered out by checking the slot from which they originate.

The counterpart to the register method is unregister. It removes a listener from the set of objects that will be notified about new events. Some spurious events may still be reported to the removed listener if an event broadcast operation is already in progress.

Note:
The event handling methods are invoked by a thread that has to notify other listeners, too. Therefore, event handling methods have to execute fast, and they cannot invoke blocking methods like waitForCard. Otherwise, the system may run into a deadlock.

The creation of a SmartCard object like in the example above is at the outer limit of what should be done in the event handler itself. If more complex operations have to be performed to process an event, the application thread should be notified. This may require event queuing in order to prevent events from going unnoticed because the application thread is busy processing a previous event. You will find more information about event handling in books about JAVA'S ABSTRACT WINDOW TOOLKIT (AWT).

Obtaining a smart card from a particular card terminal

Let's assume that a system has configured multiple card readers or multi-slotted card readers. Applications may have a need for a particular card reader or slot. The reason may be that only this card reader supports prompting the cardholder for the *cardholder verification* (CHV) data using the built-in display and PIN pad[10], or that only this slot supports *capturing* the physical smart card for the duration of the interaction.

Correspondingly, you have to write your application so that it looks for a particular card terminal or slot and gets a SmartCard object once a physical smart card has been inserted. The system-wide CardTerminalRegistry instance offers the getCardTerminals() method, which returns an Enumeration of all CardTerminal objects presently registered. You get a reference to the CardTerminalRegistry instance through the getRegistry() class method. Once you have obtained the Enumeration of CardTerminal objects, you can walk through it and determine the one you are looking for.

Your code to identify the proper terminal might look something like this:

```
try {
  // initialize framework
  SmartCard.start();

  // get the enumeration of presently registered card terminals
  Enumeration terminals = CardTerminalRegistry.getRegistry().getCardTerminals();

  // iterate over terminals to get the first instance of proper type
  String wantedType = new String("IBM5948-B02");
  CardTerminal terminal = null;
  while (terminals.hasMoreElements()) {
    terminal = (CardTerminal) terminals.nextElement();
    if (terminal.getType() .equals(wantedType)) {
      wanted = terminal;
      break;
```

10. Performing CHV without exposing the entered PIN code or password outside of the card reader device's security perimeter enhances security.

```
    }
  }

  // do something with the wanted terminal
  // ...
} catch OpenCard Exceptions {
  // handle exception
  // ...
}
```

Once you have the reference to the wanted terminal, you can either use the application-driven model and *wait* for a card using the SmartCard's waitForCard() method to pass it a CardRequest object with the card terminal set (by means of the setCardTerminal() method), or you can use the event-driven model and *get* the card using the SmartCard's getSmartCard() method to pass it the CardTerminalEvent object you received via the cardInserted() notification.

Gaining exclusive access to the smart card

Exclusive access to smart cards can be required in situations where an application needs to perform a semantic function on the card that requires an uninterrupted sequence of smart card operations. A good example for this requirement is a payment application such as running in an automatic teller machine (ATM)[11] that has to debit the card and credit some account (or vice versa) by a certain amount within a transaction context.

In your applications, you can acquire a *mutex context* by calling the beginMutex() method and release it by calling endMutex() on the SmartCard object. For the application to be well-behaved, you should claim the mutex context for as short a period as possible to avoid unnecessarily blocking other applications (or threads) from accessing the card.

Referencing files

ISO-1716-4 defines four different ways of referencing files within a smart card. They are:

Referencing by file identifier:
Files may be referenced by a file identifier coded in two bytes. For uniqueness, EFs (files) and DFs (directories) under a common DF shall have different file identifiers. The MF (root directory) shall have a file ID of hexadecimal value 0x3f00.

Referencing by path:
Files may referenced by a path, i.e. a concatenation of file identifiers. The path begins with the identifier of the MF or of the current DF, the order of the identifiers is parent to child. In case the identifier of the current directory is unknown, a hexadecimal value of 0x3fff can be used instead (reserved value).

Referencing by short EF identifier:
EFs can be referenced by a short EF identifier coded on five bits with values in the range from 1 to 30. The value 0 is used to reference the currently selected EF. Short EF identifiers cannot be used in a path or as a file ID.

11. ATM's typically take in the cards for the period of interaction to prevent early removal of the card by the user.

Referencing by DF name:

DFs can referenced by a DF name coded in one to sixteen bytes. For uniqueness, DFs within a card shall have different DF names (see also the specification of application identifiers).

In the OpenCard Framework reference implementation, references to files in the smart card are abstracted by the `CardFilePath` class. Most of what you want to do as an application programmer can be performed via this class. For convenience, the `CardFilePath` class offers two constructors, one taking a string and one taking a byte array as the argument. You use the string-based constructor when your application obtains the reference from a file or system property or when you hard-code file IDs for improved readability. You use the byte array constructor when your application obtains the file reference directly as a byte value, for instance, by reading a *DIR*[12] file in the card itself.

String representations of file references have to conform to the following syntax:

- A single file ID starts with a colon (″`:`″) followed by four hexadecimal digits[13]. Example: The master file (MF) of an ISO file system card is represented by the strings ″`:3f00`″ or ″`:3F00`″.
- A short file ID starts with a colon (″`:`″) followed by two hexadecimal digits representing the 5–bit value of the identifier. Example: the short file ID `20` (decimal) is represented by the string: ″`:14`″.
- An application identifier (or DF name) starts with a hash (″`#`″) followed by an even number of hexadecimal digits specifying the byte values of the application identifier.
- An application identifier (or DF name) in string notation does not start with a special character. It consists of a sequence of ASCII characters which specify the byte values of the application identifier.

Given these definitions, the following strings represent valid card file paths:

File reference	**Meaning**
`:3f00`	The `MF` (root directory)
`:3f00:c200:f5`	The `EF` (file) having a short EF ID of `21</ xph> contained inDF` (directory) having a file ID of `0xc200`

In addition to supporting certain file referencing methods, the OpenCard Framework reference implementation offers referencing files via symbolic file names.

File reference	**Meaning**
`/MF/LOYALTY/BONUS`	The `EF` (file) named `BONUS` in the `DF` (directory) named `LOYALTY` under the `MF` (root directory)

Smart cards do not commonly support symbolic names directly, i.e. the symbolic names specified in the card-external application must eventually be resolved into one of the ISO-defined file references. Typically, this resolution is done by the `CardService` layer and is not something the application programmer is concerned with directly. However, since not all implementations will support symbolic names, that the documentation accompanying the particular card services in use be checked.

12. DIR files are EFs containing a list of applications that a card contains along with optional information (e.g. the path to the DF application).

13. A hexadecimal digit is a character from the set 0-9, a-f, A-F.

Apart from simply improving the readability of applications, symbolic path names imply an additional redirection when referencing files in the card. This redirection can be used by card issuers to their advantage by letting them allocate ISO-based file IDs independently of the application logical file structure. The net benefit to application programmers is that the application becomes more portable by being independent of the details of file reference usage.

Working with credentials

At the start of a smart-card aware application, the card-external application and the card-resident application usually engage in a procedure to establish a trusted relationship prior to exchanging any sensitive information. This procedure may involve one or more of the following measures:

1. The card-external application authenticates itself to the card-resident application. In smart card parlance, this is called *External Authentication*.
2. The card-resident application authenticates itself to the card-external application. In smart card parlance, this is called *Internal Authentication*.
3. The cardholder authenticates him or herself to the card. In smart card parlance, this is called *Cardholder Verification (CHV)*.

Steps 1 and 2 combined are sometimes called *mutual authentication*.

Mutual authentication typically relies on a challenge-response protocol involving the computation of a response to a challenge using some cryptographic keys shared between challenger and responder. Cardholder verification typically relies on the cardholder proving his identity by revealing a shared secret (such as a password or PIN code) or by presenting a biometric characteristic such as a finger print or retina pattern scan. The card-resident application usually occupies a directory (DF) and all the files underneath it.[14]

Accordingly, granting access to an application can be equated with granting access to a directory. Access conditions to files in the file system of a smart card are typically defined during layout specification (see also “Developing applications using OpenCard Framework” on page 2.)[15]To abstract from file-system oriented cards, we call this a security domain. The access conditions for internal authentication rely on cryptographic key(s) stored in the card as part of the card-resident application, while the access condition for external authentication relies on cryptographic key(s) provided by the card-external application. For access conditions involving CHV, see “Performing cardholder verification” on page 39.

The cryptographic algorithms used in the computations of access conditions can vary between different types of smart cards. Accordingly, in situations where a card-external application has to interact with card-resident applications protected via different cryptographic mechanisms, the card-external application must be able to provide cryptographic keys for all these mechanisms.

Now what does this all mean to application programmers? It means that they must have a way to make available in their (card-external) application proper credentials (most often cryptographic keys or certificates) that fit the access conditions of the files which the given application is supposed to interact with.

14. In the case of JavaCards, the model will be somewhat different, but the essence of what is said here remains valid.

15. This may be different in the case of emerging multi-function smart cards whose operating system allows for the post-issuance deployment of applications.

The OpenCard Framework has hooks in the package called opencard.opt.security that allow the application programmer to do just that: Interface SecureService has a method called provideCredentials which allows the application to pass a bag of credentials for a security domain. Sample card services that implement this interface are FileAccessCardService or SignatureCardService.

Within CredentialBag, the credentials are organized as follows: CredentialBag contains a CredentialStore for each type of card that is supported by the application called /SecurityDomain. Each CredentialStore conceptionally is a dictionary of key/value pairs that associate a key with a Credential. A Credential usually represents a key and the associated algorithm needed to perform a cryptographic function.

Since some of these concepts are implemented differently for specific cards, the OpenCard Framework provides only such abstract classes as CredentialStore or (tag) interfaces like SecureService, Credential, SignCredential, SymmetricCredential, and PKACredential where standardization is not yet achieved. The card service implementer for a specific card must implement these interfaces or subclass the abstract classes.

In those cases in which we see enough commonality between different cards, the OpenCard Framework provides concrete implementations of Credentials like RSASignCredential and DSASignCredential used to perform external authentication using public key algorithms. Other concrete credential implementations may be added over time.

As an application programmer using a specific card service, you thus have to instantiate the appropriate subclasses of the Credentials interfaces matching the used card services. You then put the credentials in the CredentialStore subclasses provided by the card service and add them to a CredentialBag. Finally, you make the credentials available by invoking the provideCredentials method of the card service needing the credentials.

In the example pseudo code below, we assume the application has to interact with two types of smart cards with cryptographic algorithms relying on DES 18 and RSA 19, respectively. The corresponding DESSignCredential class and VendorXCredentialStore classes are provided by the respective card vendors (who also provide the FileAccessCardService implementations).

```
// instantiate DES keys
DESSignCredential desCred0 = new DESSignCredential(...);
DESSignCredential desCred1 = new DESSignCredential(...);
// instantiate key store for DES type smart cards
VendorXCredentialStore desStore = new VendorXCredentialStore();
// add credentials to store under a lookup number
desStore.storeCredential(new Integer(0), desCred0);
desStore.storeCredential(new Integer(1), desCred1);
// instantiate private RSA keys
RSASignCredential rsaCred0 = new RSASignCredential(...);
RSASignCredential rsaCred1 = new RSASignCredential(...);
// instantiate key store for RSA type smart cards
VendorYCredentialStore rsaStore = new VendorYCredentialStore();
// add credentials to store under a lookup number
rsaStore.storeCredential( new Integer(0), rsaCred0);
rsaStore.storeCredential( new Integer(0), rsaCred1);
// instantiate credential bag for the application's security domain
CredentialBag credBag = new CredentialBag()
// add stores to bag
credBag.addCredentialStore(desStore);
credBag.addCredentialStore(rsaStore);
```

```
  // work with smart card
  try {

    // wait for smart card
    SmartCard sc = SmartCard.waitForCard(
    new CardRequest(opencard.opt.iso.fs.FileAccessCardService.class));
    // get file system card service
    FileAccessCardService fscs =
    card.getCardService(opencard.opt.iso.fs.FileAccessCardService.class, false);
    // for file oriented card use a path as security domain
    CardFilePath domain = new CardFilePath(":3F00:C110");
    // add root key bag
    fscs.provideCredentials(domain, credBag);
    // keep going
    // ...

  } catch OpenCard Exceptions {
  // ...
}
```

Observe that the Key and KeyStore classes in the code sample are fictitious; look into the documentations provided by your card service provider to find out the proper classes to use with your card services.

Performing cardholder verification

The card-resident component of a smart card application typically resides within a directory (DF) of the smart card file system. Access to this application directory is usually granted only if the cardholder's identity can be successfully verified. This verification process is called cardholder verification (CHV) and is commonly based on a secret shared between the card-resident application and the cardholder. The secret is called CHV data and can be a PIN code or password that the cardholder must provide after being prompted for input.

The prompting of the cardholder and the acquisition of the CHV data can be performed in one of two ways:

GUI-based
: The cardholder is presented with a GUI that prompts for input of the CHV data. The assumption here is that the application is running in a general purpose computing device with display and keyboard or pen input and a smart card reader attached to it.

Terminal-based
: The cardholder is prompted for input by the card terminal. The assumption here is that the card terminal has a built-in (simple) display and keyboard or PIN pad.

Both approaches have their pluses and minuses. The GUI-based CHV acquisition gives the application programmer complete freedom in forming the visual appearance of the dialog with the user; this may be desirable if company branding is important for all interactions with a cardholder. The disadvantage is that the sensitive CHV data is entered via a normal keyboard and is maintained in the computer's memory and, therefore, is subject to eavesdropping attacks. In contrast, the terminal-based approach keeps the sensitive data within the security perimeter of card terminal devices, which are often designed and manufactured to meet certain security criteria. The drawback of these devices is that they typically offer only simple text-based dialog capability.

Provided a card terminal supports terminal-based CHV, which is the case if the corresponding `CardTerminal` subclass implements the `VerifiedAPDUInterface` interface, the OpenCard Framework reference implementation will use this terminal capability for increased security. In other words, you as an application programmer cannot enforce the GUI-based approach. At present, the OpenCard Framework has not architected a way for you to specify the prompt statement that gets presented to the user. Check the documentation of your card service implementation for a possibility to specify this string.

In case a card terminal does not support terminal-based CHV, the OpenCard Framework reference implementation will use the GUI-based approach. In this case, you have the option of either accepting the default GUI dialog implemented in the `DefaultCHVDialog` class or implementing your own GUI dialog and registering it with the `CardService` object. Your GUI component must implement the `CHVDialog` interface. You register it via the `CardService` object's `setCHVDialog()` method, which takes your GUI instance as the argument.

The fact that the OpenCard Framework reference implementation does not give the programmer the choice between GUI-based and terminal-based CHV acquisition may at first seem arbitrarily restrictive. However, the impact on security in return for freely providing a choice contradicts the design objectives of the OpenCard Framework and, in most cases, would contradict the original intent that motivated the use of smart cards to secure an application in the first place.

Using the trace utility

Tracing is a widely practiced programming technique; during program development it is convenient for debugging and, once an application is deployed, it can serve to produce an operations log that can be used for diagnostics. The amount of tracing output to be produced depends on the purpose and often is configurable at load-time or even run-time of the application.

The OpenCard Framework reference implementation internally makes use of a tracing utility implemented in the `Tracer` class. You can use this utility for your own purposes when developing your application. The `Tracer` class distinguishes a total of eight different trace levels and can be configured via the `OpenCard.trace` system property (see "Configuring tracing output" on page 17). For each trace level, the `Tracer` class offers two methods to issue a trace message, one taking a `String` object as a message parameter, the other taking a `Throwable` object instead.

The code of a class of yours using the `Tracer` class might look something like this:

```
public class MyClass {

  // instance tracer
  private Tracer itracer = new Tracer(this, MyClass.class);

  // class tracer
  private static Tracer ctracer = new Tracer(MyClass.class);

  public MyClass() {
    // ...
    ctracer.info("<init>", "initializing");

    // ...
}

public void someMethod() {
  // ...
  itracer.debug("someMethod", "something's fishy here");
  // ...
  try {
    // ...
  } catch OpenCard Exceptions {
    // ...
    itracer.critical("someMethod", e);

    // ...
    }
}
```

In addition to just creating tracing output, the `Tracer` class offers a trace event-notification service. This service notifies all components that have previously registered with a `Tracer` object about the occurrence of a trace event. You can register components via the `addTracerListener()` and `removeTracerListener()` methods of the `Tracer` class. The components you register must implement the `TracerListener` interface, which specifies the single `traceEvent()` method taking a `TracerEvent` object as a parameter.

Chapter 6. Using the OpenCard Framework in applets

Using the OCF functionality in browsers brings additional advantages, but also requires some extra work to be done. This is due mostly to the security concerns raised by loading executable content from the INTERNET. In the old days of the Sandbox model, operations considered potentially harmful were simply not allowed for applets loaded from the INTERNET. OCF performs a few operations which would have been included in this category. The security model of the JAVA PLATFORM 1.1 and even more so of the upcoming version 1.2 do allow such operations to be performed under certain conditions. It should be noted though that the security models differ substantially in versions 1.1 and 1.2 and that different tools are being used. What complicates matters even more is the fact that the two most prominent INTERNET browsers used —i.e. NETSCAPE NAVIGATOR (COMMUNICATOR) and MICROSOFT INTERNET EXPLORER — still use their proprietary security model. To remedy this situation, SUN (JAVASOFT) introduced the JAVA PLUG-IN (formerly named ACTIVATOR) which allows users to specify an alternative JVM which can then be used instead of the browser's built-in virtual machine.

The following chapter describes both alternative approaches:

- using the more portable pure java approach of the Java Plug-IN
- using the proprietary security mechanisms of the browsers

Usage scenario

The easiest thing is to write an applet that uses OCF, bundle all that together, and download all this stuff in its entirety. But although OCF is not too fat (and we'll strive to make it even leaner), we did not want users having to download software they already have (the OCF itself that is) sitting on their local machine. Our usage scenario therefore consists of the OCF being installed on the local user machine and an applet being downloaded from the INTERNET using that local OCF.

The security model of the Java Platform 1.1

Version 1.1 basically extends the JAVA PLATFORM 1.0 model such that digitally signed applets are granted the same privileges as code loaded locally (provided that the certificate used for creating the signature is known). So that's still the old ″all or nothing″ approach only ″improved″ by the ability to allow applets to perform potentially risky operations at all. Local code is trusted generally.

The security model of the Java Platform 1.2

This version will substantially improve security management as it will be possible to grant privileges in a very fine-grained matter. Within the range of ″all″ or ″nothing,″ one may state the operations allowed to be carried out by code coming from a certain location and signed by one or several signers. One notable difference to the model of the PLATFORM 1.1 is that code stored locally is conceptually regarded not to be more trustworthy than code downloaded from the INTERNET.

The Java Plug-In

This is basically a piece of code (technically an Active/X control for MS Internet Explorer or a plug-in for the Netscape Navigator) that circumvents the browser's built-in virtual machine and uses another one. This may be either a JRE coming with the Plug-in software or any other version you wish to use, e.g. a beta version of the upcoming Java Platform 1.2. HTML documents have to be instrumented with some special tags in order for the browsers to detect the request for the Plug-in to be executed. If it has not been installed before, the browser offers to download it on the fly. Once downloaded and installed, the browser will use the Plug-in software every time it encounters such an instrumented HTML document. This in turn will cause the desired JVM to be used instead of the browser's own version. The Plug-in comes with a control panel which allows you for instance to determine which JVM and which proxies to use.

Note:
This software does not currently support SOCKS servers. In order to instrument your HTML documents with the required tags, you can use a conversion tool which is offered by Sun free of charge and is pretty easy to use.

Writing secure applets with OCF and the Java Plug-In

As the developer, you'll have to acquire two things: the Plug-in and a digital certificate. The easiest way to get the Plug-in is to visit the Plug-in product site at

```
http://www.javasoft.com/products/plugin/
```

and select the appropriate download package for your platform.

As these HTML documents are all instrumented with the special tags, your browser will note that the Plug-in is not installed yet on your local machine and offer to download it. Well, there's no reason to turn down this offer here;-) so go ahead and get the chunk of about 5 MB. Install the thing and start your browser anew. After that, visit the above mentioned demo site and try one of the samples again. This should now bring up the Plug-in console window, and the demos should work. If this is not the case, check the Plug-in's FAQ (chances are the problems are caused by proxies).

There are basically two ways of getting a certificate: buying one or making one. The latter case is acceptable only for testing purposes because everyone can do that (including some not-so well behaved hacker or criminal). The bottom line is that everyone should be very suspicious of code that may have been signed while the certificate coming with it was not issued by an official Certification Authority. These are companies (e.g. VeriSign, Thawte) that officially confirm that individuals or organizations/companies are who they claim to be. To this end, they'll give the certificate requester a digital certificate signed by them. The certificate can then be used by developers (that means you;-) to digitally sign their code (e.g. applets). The recipient of the code is then able to check whether or not the code was changed during transmission and that it indeed does come from the one who claims he sent/provided it.

Note:
Please bear in mind that nothing else can be concluded from a digital signature, e.g. that the code does not carry out harmful operations on your machine, is bug-free, generally usable, or well-engineered.

The Certification Authorities have digital certificates themselves which are stored by default in the browsers which are thus able to check certificates coming with digital signatures in some code downloaded from the INTERNET. If you nonetheless want to create a so called ″self-signed″ certificate for testing purposes, we'll show you how to do that in the sections below which are specific to the JAVA PLATFORM 1.1 and 1.2.

Once you have a certificate, you can use it to sign your code. To do that, the code must be packaged into a Jar archive. Signing the archive will cause it to increase in size slightly as some data will be added to it. After you have written the HTML document using your applet (and tested it locally with the Appletviewer), you must adapt it to use the special tags required for the PLUG-IN. Using the HTML converter for this chore is easiest.

Now you are ready to deploy the converted HTML document along with the applet archived in a Jar file. Move it to your web server and test it.

Differences in the procedure between Java Platform 1.1 and 1.2

As mentioned above the security models have evolved quite a bit from the early JAVA versions, over PLATFORM 1.1, up to the upcoming PLATFORM 1.2. Concepts, terms, capabilities and tools are different affecting the procedure in the following areas:

- creating and storing a self-signed certificate
- signing an archive
- importing a certificate.

These procedures are described in more detail below for the respective platform.

Writing secure applets with OCF for Java Platform 1.1

1. Creating and storing a self-signed certificate (if not using an official certificate).
   - Create an identity in the identity database:

     ```
     javakey -cs <identityName> true
     // e.g. javakey -cs "Mike" true
     ```

   - Create a certificate directive file, which basically is a simple ASCII text file e.g.

     ```
     issuer.name=Mike
     issuer.cert=1
     subject.name=Mike
     subject.real.name=Mike Wendler
     subject.org.unit=SC
     subject.org=IBM
     subject.country=GE
     start.date=18 Jun 1997
     end.date=15 Sep 1998
     serial.number=1001
     out.file=MikesCert4J11.cer
     ```

- Generate a certificate:

  ```
  javakey -gc <certification_directive_file>
  // e.g. javakey -gc MikesCertDirective
  ```

2. Digitally sign an archive.
   - Create a signing directive file, which basically is a simple ASCII text file as well, e.g.

     ```
     signer=Mike
     cert=1
     chain=0
     signature.file=MIKESIGN
     out.file=SignedApplet.jar
     ```

   - Sign the archive:

     ```
     javakey -gs <sign_directive_file>
     // e.g. javakey -gs MikesSignDirective
     ```

For more information about javakey please have a look at the tool documentation accompanying each JDK.

Using secure applets with OCF for the Java Platform 1.1

1. Create a new trusted identity in your keystore (which is being created if it does not already exist)

   ```
   javakey -cs <identityName> true
   // e.g. javakey -cs "Mike" true
   ```

2. Import the certificate of the new trusted identity into the local keystore:

   ```
   javakey -ic <identityName> <certificateFile>
   // e.g. javakey -cs "Mike" MikesCert4J11.cer
   ```

3. Download the HTML document with the desired applet from the web server.

Steps 4 - 7 are only required if you haven't installed the JAVA PLUG-IN before.

4. Download the PLUG-IN when prompted and install it.
5. After installation, activate the JAVA PLUG-IN control panel and select the ″Advanced″ tab. In the JAVA RUNTIME ENVIRONMENT section, select one of the one or more versions of JDK 1.1 that should be listed there.
6. If the only proxy you use is a SOCKS, you'll have to set an alternative http proxy in the ″Proxies″ section of the JAVA PLUG-IN control panel. (The JAVA PLUG-IN cannot handle SOCKS servers).
7. Start your browser anew and fetch the corresponding HTML document again (step 3).

Writing secure applets with OCF for the Java Platform 1.2

1. Create and store a self-signed certificate (if not using an official certificate):
   - Create a key pair in the identity database:

     ```
     keytool -genkey -alias <aliasName> -keystore
     <keystoreName> -keypass <keyPassword>
     -dname <x500Name> -storepass <keystorePassword>
     // e.g. keytool -genkey -alias Mike -keystore .keystore
     // -keypass Mikeskeypass -dname "cn=Mike" -storepass Mikesstorepass
     ```

     Export the self-signed certificate:

     ```
     keytool -export -alias <aliasName> -file <certificateFileName>
     // e.g. keytool -export -alias Mike -file MikesCert4J12.cer
     ```

2. Digitally sign an archive:

```
jarsigner -storepass <keystorePassword> -keystore <keystoreName>
-keypass <keyPassword> <JarArchive> <aliasName>
// e.g. jarsigner -storepass Mikesstorepass -keystore .keystore
-keypass Mikeskeypass SignedApplet.jar Mike
```

Using secure applets with OCF for the Java Platform 1.2

1. If you have not already done so, install the latest version of JDK 1.2 and update your environment accordingly.
2. If you do not already have a `.java.policy` file in your `user.home` directory, provide one. This file determines what actions foreign applets are allowed to carry out on your local machine. A possible policy file (setting all required privileges for the Internet Broker demo) may look like this:

```
// this keystore is to store our certificates
keystore ".keystore";

// a grant entry suitable for the Internet Broker Demo
// allows ALL applets that were signed by "Mike" to carry out the following actions
grant signedBy "Mike" {
    // read and write arbitrary (including sensitive) system properties
    permission java.util.PropertyPermission "*", "read,write";

    // read the 'opencard.properties' file in the standard locations
    permission java.io.FilePermission "${java.home}/lib/opencard.properties", "read";
    permission java.io.FilePermission "${user.home}/.opencard.properties", "read";
    permission java.io.FilePermission "${user.dir}/opencard.properties", "read";
    permission java.io.FilePermission "${user.dir}/.opencard.properties", "read";

    // dynamically load native libraries
    permission java.lang.RuntimePermission "loadLibrary.*";

    // get access to declared constructors/methods/fields via reflection API
    permission java.lang.RuntimePermission "reflect.declared.*";
};
```

3. Import the certificate into your keystore (which is being created if it does not already exist):

```
keytool -import -alias <aliasName> -file <certificateFile>
    -storepass <yourKeyStorePasswordHere>
    // e.g. keytool -import -alias Mike -file MikesCert.cer
    -storepass <yourKeyStorePasswordHere>
```

4. If you don't already have one, create a Jar archive of OCF on the local machine and sign it with either a self-signed or an ″official″ certificate.
5. Download the HTML document with the desired applet from the web server.

Steps 6 - 9 are required only if you haven't installed the JAVA PLUG-IN before.

6. Download the PLUG-IN when prompted and install it.
7. After installation, activate the JAVA PLUG-IN control panel and select the ″Advanced″ tab. In the JAVA RUNTIME ENVIRONMENT section, select one of the one or more versions of JDK 1.1 that should be listed there.
8. If the only proxy you use is a SOCKS, you'll have to set an alternative http proxy in the 'Proxies' section of the Java Plug-in control panel (e.g. `proxy.de.ibm.com/` on port 80). (The JAVA PLUG-IN cannot handle SOCKS servers).
9. Start your browser anew and fetch the according HTML document again (step 3).

For more information about `keytool/jarsigner`, please have a look at the tool documentation accompanying each JDK.

Native Browser support

By native browser support we understand the deployment of OCF under the proprietary JVM and security model of a specific vendor's browser product. The most commonly used products at the time of writing, `Netscape Navigator` and `Microsoft Explorer` both allow applets to escape the Java 1.0 sandbox and use system resources like native calls, file system and properties from within a downloaded applet if

- the applet is signed using the vendor specific signing process
- the applet calls proprietary APIs to request the required privileges

.

The following instructions outline the procedures to get signed applets working under some browsers' proprietary Java virtual machines.

The opencard.core.util.SystemAccess class

OCF needs access to system resources. Native calls (JNI) are needed in some terminal implementations, access to properties is needed in the initalization step to configure the registries, access to the file system is needed when reading the `opencard.properties` file. Access to those resources from an applet in a browser environment requires the caller to request the privilege to do so from the browser using proprietary APIs.

To avoid vendor specific calls in the OCF pure java code base this browser specific code has been isolated in the singleton class `SystemAccess. By default OCF is configured with classopencard.core.util.SystemAccess which does not contain browser specific code. This works fine when using the Java Plug-In JVM or when using Java applications. When running OCF under a browser's VM a browser specific SystemAccess class should be configured as follows before the SmartCard.start() method is invoked:`

```
public void init() {
...
opencard.core.util.SystemAccess sys =
  new opencard.opt.vendorX.VendorXSystemAccess();
opencard.core.util.SystemAccess.setSystemAccess(sys);
...
SmartCard.start();
...
}
```

OCF will then use the configured SystemAccess class instead.

To make sure the browser specific SystemAccess class is considered part of the signed applet the class file must be packaged as part of the signed applet and must not be contained in the system class path of the browser environment.

Writing secure applets with OpenCard and Netscape Communicator

The following instructions use Netscape Communicator 4.07 under Windows NT 4.0 but the concepts presented can be applied to other platforms as well.

Install the OpenCard base components in a local directory. The `CLASSPATH` environment variable should point to the OCF class or jar files in the environment where the browser is invoked.

If the configured card terminals need native libraries, the DLLs should be placed in the \program\java\bin directory of the Netscape product tree.

OpenCard searches in java.home, user.home and user.dir directories for its properties file. ″java.home″ is unset under Netscape, ″user.home″ is set to \Netscape\Users\<userid>\ and ″user.dir″ is set to the current directory of the browser. The recommendation is to install a <netscape-product-path>\Netscape\Users\<youruserid>\.opencard.properties file.

Add the following code to the `init()` method of your applet:

```
opencard.core.util.SystemAccess sys =
  new opencard.opt.netscape.NetscapeSystemAccess();
opencard.core.util.SystemAccess.setSystemAccess(sys);
```

Package your applet, the opencard.opt.netscape.NetscapeSystemAccess.class and other classes needed (but not contained in the locally installed OCF) into one jar file by using Netscape's `signtool`. The signtool plus documentation is available from http://developer.netscape.com/docs/manuals/signedobj/signtool/index.htm. The obtained jar file must be signed using a valid signer certificate. The certificate necessary is Netscape specific. A test key pair and certificate can be created by using the signtool.

Writing secure applets with OpenCard and Microsoft Explorer

The following instructions use Microsoft Explorer 4.0 under Windows NT 4.0 but the concepts presented can be applied to other platforms as well. To successfully run native card terminals the JNI update to the Microsoft JVM needs to be installed. It can be downloaded from http://www.microsoft.com/java.

Install the OpenCard base components in a local directory. The `CLASSPATH` environment variable should point to the OCF class or jar files in the environment where the browser is invoked.

If the configured card terminals need native libraries, the DLLs should be placed in a directory contained in the `PATH` environment variable.

OpenCard searches the following paths under Microsofts JVM:

```
java.home:
C:\WINNT\Java\lib\opencard.properties,
C:\WINNT\Java\.opencard.properties
user.home:
C:\WINNT\Profiles\<youruserid>\Desktop\opencard.properties,
C:\WINNT\Profiles\<youruserid>\Desktop\.opencard.properties
```

The recommendation is to put the opencard.properties into \<system>\profiles\<userid>\Desktop\ directory.

Add the following code to the `init` method of your applet:

```
opencard.core.util.SystemAccess sys =
  new opencard.opt.ms.MicrosoftSystemAccess();
opencard.core.util.SystemAccess.setSystemAccess(sys);
```

Package your applet, the opencard.opt.ms.MicrosoftSystemAccess.class and other classes needed (but not contained in the locally installed OCF) into one cab file by using Microsoft's SDK for Java `cabarc and signcode` tools. The tools plus documentation is available from http://www.microsoft.com/java. The obtained cab file must be signed using a valid signer certificate. The certificate necessary is Microsoft specific. A test key pair and certificate can be created by using the Microsoft SDK for Java.

Chapter 7. Writing a CardTerminal

Writing a `CardTerminal` isn't particularly difficult — provided you know everything you need to know about your smart card reader. In this section, we'll show you how to implement a `CardTerminal`. There are many possible ways of obtaining a working `CardTerminal`, so we'll limit ourselves to just a few of the more important kinds of implementation.

Implementing a CardTerminal

`Note:`All timeout parameters should be ignored in terminal implementations. The timeout parameter still exists as a placeholder from previous releases of OCF where it was used to avoid breaking binary compatibility of existing CardTerminals. Implementers of new CardTerminals should simply ignore this parameter!

It is convenient to divide `CardTerminal` implementation into three layers. Depending upon the specific circumstances of your case, you may have to implement all three layers or you may be able to improvise using pre-existent parts.

interface layer
This layer is derived from the abstract `CardTerminal` class. It provides the methods for interfacing with the upper part of the framework.

function layer
This layer encapsulates the detailed knowledge about your reader.

communication layer
This layer provides methods for handling the transfer of the data between the host (your computer) and the card reader.

The interface layer

This layer provides the standard functions needed by OCF to access the smart card. The top-level `CardTerminal` implementation must inherit from the `opencard.core.terminal.CardTerminal` class to provide these functions.

Important methods

In order to write a `CardTerminal`, you will have to implement or overwrite at least the following methods:

- `Constructor`
- `open()` and `close()`
- `isCardPresent()`
- `getCardID()`
- `powerUpCard()`
- `internalFeatures()`
- `internalReset()`
- `internalSendAPDU()`

The Constructor: First, you should overwrite the default Constructor. It is important to invoke the Constructor of the parent class (see the sample code in the following example). After doing this, you must add the Slot objects to your terminal.

The following is a sample Constructor for a card reader possessing a single slot:

```
protected MyCardTerminal(String name, String type, String device)
    throws CardTerminalException {

  super(name, type, device);
  addSlots(1);
}
```

The open() and close() methods: The open() and close() methods should implement the initialization and cleanup of the next-lower layer (the **function layer**). The open() method must make sure that successive calls to the isCardPresent() method are not using unitialized data. In the case of an unrecoverable error, you should abort and throw a CardTerminalException with a plain-language message to assist the application programmer in localizing the problem.

The isCardPresent() method: The task of the isCardPresent() method is to recognize whether or not a smart card has been inserted in a given slot. The way to get this information from your terminal depends on your concrete implementation. Perhaps the best way is to poll the terminal and hold the slot status within a cache object. From the point-of-view of performance, the isCardPresent() method should analyze the content of the cache object and should not always push a status request to the reader (see also "Polling" on page 54)

The getCardID() method: getCardID() is responsible for returning the ATR of a smart card in a given slot encapsulated in a CardID object.

As a rule, you can't return the ATR without explicitly powering up the card. But you should determine whether you've already powered up the card before. Otherwise, you might cause a repower — equivalent to resetting the smart card — which can prove very confusing for your application. Luckily, it's quite easy to check whether the card has been powered up before if you cache the ATR on every power-up and reset this cache (set to null) every time a card is removed.

There are two getCardID methods. One has an additional parameter: int ms. This timeout was used in previous releases and should now be ignored. It is not meant to wait for a card insertion if no card is available in the requested slot. In this case, you should throw a CardTerminalException with a message in plain language (e.g.: "no card inserted").

You should write your implementation code using the getCardID(int slotID) method. The other getCardID(int slotID, int timeout) method with the timeout parameter only invokes the first method without passing on the timeout value (which is to be ignored anyway).

Sample code for a single-slot reader:

```
private byte[] cachedATR = null;

public CardID getCardID(int slotID)
                    throws CardTerminalException {
  CardID cardID = null;

  if (isCardPresent(slotID)) {
    // check if ATR was already read
    // (if card is already powered)
    if (cachedATR == null)
      // card must be previously powered
      cardID = new CardID(getSlot(slotID),
                          powerUpCard(slotID));

    else
      // card was already powered
      cardID = new CardID(getSlot(slotID), cachedATR);
  } else
    // invalidate the cached ATR
    cachedATR = null;

  return cardID;
}

public CardID getCardID(int slotID, int timeout)
          throws CardTerminalException {

    return getCardID(slotID);
}
```

The powerUpCard() method: The powerUpCard() method employed in the getCardID() example is a helper method which returns a byte array with the ATR of the newly powered card. Furthermore, it encapsulates methods from the **Function Layer** for powering up the smart card and obtaining the ATR from it. The latter method is often coupled with the reader's power-up mechanism.

The internalFeatures() method: The next thing you've got to do is to implement the internalFeatures() method. This method returns possible additional features like information about the terminal's PIN pad or display. This information is appended to the basic, internal information on the terminal (i.e. its name, type, and address). The sample code in the following example illustrates this:

```
protected Properties internalFeatures(Properties features) {
  features.put("FEATURE", "VALUE");
  return features;
}
```

The internalReset() method: The next method you have to implement is referred to as the internalReset() method. This method should reset and power up the smart card. Consequently, it must return the newly-read ATR encapsulated within a CardID object. Please don't forget to update the cached ATR.

The internalSendAPDU() method: Last but not least, you have to implement the internalSendAPDU() method. This method is responsible for the transfer of data between your terminal and the reader.

The caller of the internalSendAPDU() method wraps the card-specific commands in a CommandAPDU object. The CommandAPDU class incorporates a further method named getBytes(), which provides a means of extracting the whole command as a byte array.

Generally speaking, it is necessary to distinguish between two cases, depending upon whether the card protocol is based on the T0 protocol or on the T1 protocol.

The limitation of the T0 protocol is that it doesn't have the ability to send and receive APDUs with both additional data fields at the same time. In this case (known as the ″case four″ command — see ISO 7816-4), you have to initiate a receive command after the send command in order to get all of the information from the card.

If the card protocol is based on T1, you are provided with a reader command that can exchange data in both directions in one step.

The following sample code disregards the issue of card protocol:

```
protected ResponseAPDU internalSendAPDU(int slotID,
                                        CommandAPDU capdu, int timeout)
       throws CardTerminalException {

  ResponseAPDU responseAPDU = null;
  byte[] receiveBuf = null;

  try {
    responseBuf = functionLayer.exchange(slotID,
                                         capdu.getBytes(), timeout);
    if (receiveBuf != null)
      responseAPDU = new ResponseAPDU(responseBuf);

  } catch (RuntimeException re) {
    throw new CardTerminalException(re.toString());
  } catch OpenCard Exceptions {
    throw new CardTerminalException(e.toString());
  }

  return responseAPDU;
}
```

Polling

There are various different ways from which to choose in implementing a polling mechanism:

Polling using EventGenerator/CardTerminalRegistry: EventGenerator has a special poll thread. Via the CardTerminalRegistry, you can add your terminal to EventGenerator's polling list. This is the easiest method of doing this. You need merely to implement the Pollable interface in your CardTerminal. You must then implement your CardTerminal's poll() method and add your terminal instance to EventGenerator's polling list using the CardTerminalRegistry.addPollable() method. This should be done inside the open() method of your terminal. Please don't forget to remove your terminal from the polling list inside your implementation's close() method.

Note:
Because there's only one polling thread for all pollable terminals, you should endeavor to limit the time required for one poll. It is important that you prevent your terminal from being polled if it is not yet ready to work or is already closed.

Sample code for a single-slot reader:

```
private boolean cardIsInserted = false;

public void poll() throw CardTerminalException {
  if (!closed) {
    UpdateSlotStatus(0);// update the cache for slot 0
    if (!cardIsInserted) {        // what's the last status of the terminal?
      if (isCardPresent(0)) {
        cardIsInserted = true;
        cardInserted(0);          // distribute the news to all listeners
      }
    } else {
      if (!isCardPresent(0)) {
        cardIsInserted = false;
        cardRemoved(0);           // distribute the news to all listeners
    }
  }
}
```

The private variable named `cardIsInserted` helps your terminal to decide whether a card was found at the last poll phase. This variable is initially set to ″false″ so the terminal starts its work with the knowledge that no card was recognized the last time. In other words: a card has probably been inserted but has not yet been recognized by the polling mechanism.

Polling using your own thread: You might wish to set up your own polling thread within your terminal implementation. The advantage of this method is a better influence over the timing. This means a better control over the frequency of polling.

Event-driven recognition of reader states: This is the third way of implementing the recognition of slot changes in a `CardTerminal`. In this case, your **Function Layer** and your hardware must be capable of creating events on their own without being polled.

Event-notification

In the example above, two method calls of event-notification were introduced: the `cardInserted()` method and the `cardRemoved()` method. Well, what’s going on inside the framework? Each method calls its associated method in `CardTerminalRegistry`. `CardTerminalRegistry` (not `CardTerminal`) holds a list of `CardTerminalListeners` (`CTListener`) and distributes this event to all registered listeners.

Synchronization

What about the need to synchronize sections of your code? This is very important! As a rule, you need a synchronization of almost every method in the **function yayer** (see below). But if you synchronize entire methods belonging to the first layer, you soon begin encountering performance problems. The easiest way to avoid synchronization problems is to define and use a monitor object in the first layer. You synchronize on this object in every section of your code.

Sample code:

```
private Object readerMonitor = "reader monitor";
.
.
void exampleReset(int slotID) throws CardTerminalException {
  try {
    synchronized(readerMonitor) {
      functionLayer.reset(slotID);
      functionLayer.powerUp(slotID);
    }
  } catch OpenCard Exceptions {
    throw new CardTerminalException(e.toString());
  } }
```

Now you have a overview of the implemention requirements of the first layer. In the following section, the **function layer** — which provides the real knowledge about the terminal functions — is described in detail.

The function layer

This layer could be implemented inside the **interface layer** or as a separate protected class, depending upon the complexity of the implementation and your specific requirements.

The **Function Layer** is needed because the **interface layer** knows nothing about a concrete command sequence used for powering up a smart card. This command could be a byte sequence like `6Eh 00h 00h 00h` (this sequence sent to a GemPlus GCR410 powers a smart card within the first slot).

It’s possible to write to code of the **function layer** into the same `CardTerminal` class file, but it’s logically separated.

Generally speaking, you need at least the following functionalities:
- the `open()` and `close()` methods for initializing the **function layer**,
- a suitable `Constructor` matching your configuration requirements (with parameters for the device names, address, line speed, and whatever else you might need),
- a method for getting status information from a slot,
- a method for powering up and powering down the smart card,
- a method for ejecting a smart card (if applicable) — often in combination with the power-down mechanism, and
- a method to initiate a smart card reset.

Note:
Your methods should be at least protected and should take into consideration the problem of synchronization!

There are two ways of implementing this layer:
- PURE JAVA implementation and
- native code via JNI if you want to use an existing native code library (like a DLL) to connect to your reader.

Pure Java implementation of the function layer

PURE JAVA implementation is the preferred means of implementing the **function layer**.

PURE JAVA implementation has one big advantage: the reuseability of code for multiple platforms. One possible drawback: Writing the implementation may prove more complicated because it is also necessary to implement the **communication layer**. The only thing you have to do is to provide the set of functionalities listed above (see page 56) and to translate the requests into a format your reader understands.

Sample code for exchanging data between your terminal and the reader:

```
protected byte[] exchange(byte[] sendData, int slotID, int timeout)
                     throw CardTerminalException {

  byte[] responseBuf = null;
  byte[] tempBuf = null;

  try {
      // assemble exchange command for reader
      byte[] sendCommand = new byte[sendData.length + 1];

      // insert the exchange command identifier
      // and append the content of sendData
      // (only valid for T1 cards)
      sendCommand[0] = (byte)0x15;
      System.arraycopy(sendData, 0,
                      sendCommand, 1,
                      sendData.length);

      tempBuf = communicationLayer.transmit(sendCommand);

      // copy tempBuf to responseBuf
      // without copying the additional status information
      // appended by the reader
      responseBuf = new byte[tempBuf.length - 1];
      System.arraycopy(tmpBuf, 1,
                      responseBuf, 0,
                      responseBuf.length)

    // remap all runtime-exceptions to CardTerminalException
    } catch (RuntimeException re) {
      throw new CardTerminalException(e.toString());
    }
    return responseBuf;
  }
```

In this sample code, the method has no control over low-level data transfer between your system and the reader (a task performed by the **communication layer** as described on page 57.

Native code implementation of the function layer (via JNI)

Implementation of the **function layer** with native code support is necessary if you want to use existing libraries (like DLLs) to get a connection to your reader. In the most cases, you can avoid implementing a low-level transfer protocol (**communication layer**). You must create a wrapper class with a corresponding wrapper-library. For more information on writing native code, please refer to the Java *Native Interface Tutorial* from JAVASOFT. You'll find the documentation at the following address:

`http://www.javasoft.com/docs/books/tutorial/index.html`

The communication layer

The **communication layer** encapsulates the knowledge of data exchange between your hardware and the reader. In addition to the pure transmit methods, it is very important to implement methods for the handling of communication errors. The whole protocol inside this layer should be transparent for the layers above it.

In order to simplify implementation of the **Communication Layer**, we recommend that you use the `javax.comm` extension as a basis whenever possible.

Implementing the Lockable interface

If your CardTerminal implementation is used in security sensitive scenarios where you must ensure exclusive access to a specific application across card insertion and card removal events you should implement the optional `opencard.opt.terminal.Lockable` interface.

To implement this interface the CardTerminal must do some additional security checks when a SlotChannel is requested or other functions on the terminal are invoked. To reduce the task of implementing these security checks the framework comes with a convenience abstract class `opencard.opt.terminal.AbstractLockableTerminal`. This class does most of the checking for you. If you derive your Lockable CardTerminal from this class you only need to implement some terminal specific methods which actually do the locking/unlocking like

- internalLock()
- internalUnlock()
- internalLockSlot()
- internalUnlockSlot()
- lockabelOpenSlotChannel()

For more details on the contract between the CardTerminal user and the CardTerminal implementer see `RFC 17-1 OCF Terminal Locking Mechanism` on the OpenCard website.

Implementation of a CardTerminalFactory

To get a working implementation of a `CardTerminalFactory`, you need merely write a few lines of code as presented in the example below. The primary purpose of `CardTerminalFactory` is to analyze the given configuration data (`String[] terminalInfo` in `createCardTerminals()`), to search for a known terminal type, and to instantiate the terminal with the associated terminal name.

Note:
The terminal name must be unique within the OpenCard Framework.

If you have extended the requirements made of your factory — for example by requiring the connection to a resource manager with the simultaneous instantiation of multiple terminals — you'll probably need the `open()` and `close()` methods for initialization (see also "The open() and close() methods" on page 52).

Using `createCardTerminals()`, you're able to simultaneously create multiple terminals, thus enabling you to get a list from an alien resource manager about its registered terminals and instantiate them in one piece.

The following sample code pertains to a terminal with one additional parameter:

```
public class SiemensCardTerminalFactory
       implements CardTerminalFactory {

    public void createCardTerminals(CardTerminalRegistry ctr,
                                    String[] terminalInfo)
       throws CardTerminalException,
              TerminalInitException {

      // check for minimal parameter requirements
      if (terminalInfo.length < 2)
```

```
        throw new TerminalInitException(
                      "at least 2 parameters necessary"
                      + " to identify the terminal");

      // check the given type
      if (terminalInfo[1].equals("MYMODEL")) {

        // optional: check for an additional parameters...
        if (terminalInfo.length != 3)
          throw new TerminalInitException(
                        "createCardTerminals: "
                        + "Factory needs 3 parameters "
                        + "for snuggle terminal");

        // creates the terminal instance
        // and registers to the CardTerminalRegistry
        ctr.add(new MyCardTerminal(terminalInfo[TERMINAL_NAME_ENTRY],
                                   terminalInfo[TERMINAL_TYPE_ENTRY],
                                   terminalInfo[TERMINAL_ADDRESS_ENTRY]));
                                   // these three TERMINAL variables are
                                      defined in CardTerminalFactory
    } else
        throw new TerminalInitException("Type unknown: "
                                        + terminalInfo[TERMINAL_NAME_ENTRY]);
  }

  public void open() throws CardTerminalException {}
  public void close() throws CardTerminalException {}
}
```

The code above pertains to an example factory which provides a skeleton for you to start. The most interesting method is createCardTerminals(). The factory needs at least two parameters: the reference to the CardTerminalRegistry and a string array (terminalInfo) with a list of configuration parameters of one terminal entry. Thus, the terminal name could be terminalInfo[0], and so on.

Note:
It is important that you catch all possible exceptions and remap them into CardTerminalExceptions. This recommendation pertains to all non-private methods. Otherwise, the application described above has to catch exceptions which are not derived from CardTerminalException.

Chapter 8. Writing a card service

The OpenCard Framework `CardService` level implements a standard interface for use by the application programmer, hiding smart card specifics. `CardService` generates application program data units (APDU's) to implement high-level API functions. In this section, we'll show you how to develop a particular card service and a corresponding card service factory.

CardService environment — overview

Looking up towards the application programmer, a card service implements a programming interface to provide functionality to applications. One goal of OpenCard is standardization of these *application interfaces*.

Also, an `OpenCard` `CardService` implementation must provide functions that are used by the framework itself for administration purposes. Default *framework interfaces* are provided through inheritance from standard classes.

OCF provides methods and classes for use by the `CardService` implementation to access the card. The major classes in this category are `CardChannel` and `CardService`.

A concept for *secure messaging* is provided by OpenCard. Although the actual implementation of secure messaging will not be covered here, the OpenCard provisions for secure messaging will be explained in enough detail to allow you to implement secure messaging using your cryptographic code.

Associated with each `CardService` implementation is a `CardServiceFactory` that is capable of constructing it. `CardServiceFactory` must be able to identify the card or cards for which it can construct instances of `CardService`. When a smart card is inserted into the reader, the OpenCard Framework goes through its list of registered card service factories until it finds one that can handle the card.

Application interfaces

The application interface is the set of methods that the `CardService` implementation makes available to the application program. These are typically high-level functions that fit well into the Java programming paradigm.

The `CardService` implementation has the task of mapping the method call to smart card APDU's that are communicated to the card. One method may cause one or many command - response sequences to be carried out with the card.

When a new `CardService` implementation is defined, any desired functionality may be implemented. Standard OpenCard interfaces may be used, or an application specific interface may be developed. The use of standard interfaces facilitates smart card interoperability, where application specific interfaces can provide more customized functionality.

Using standard CardService interfaces

In order to understand how the use of standard interfaces aids interoperability, it is helpful to review how `CardService` interfaces are used by the application.

After OCF has been started, the waitForCard() method can be used to wait for a card to be inserted that implements the desired interface. In the code fragment below, the application instantiates a FileAccessCardService (this service provides access to ISO file system cards) for an inserted card.

```
...
// Initialize the framework
SmartCard.start ();

...
// Assume card has been inserted, 'card' is SmartCard object.
// Instantiate file access card service.
FileAccessCardService fs = (FileAccessCardService)
                            card.getCardService(FileAccessCardService.class);
...
```

When a JAVA class is passed to the SmartCard object getCardService() method in this manner, OCF looks for a CardServiceFactory that can create an object implementing the specified class for the inserted card. If the JAVA class parameter represents an interface object rather than a class object, the returned CardService object will implement the specified application interface.

The standard application interfaces can be implemented for any smart card. In order to do this, a class implementing the interface must be written. The declaration statement for this class could appear as in the following code fragment. The accompanying CardServiceFactory (for example com.ibm.opencard.factory.MFCCardServiceFactory), will create an instance of MFCFileAccess when called upon to instantiate a FileAccessCardService object.

```
public class MFCFileAccess extends MFCCardService implements FileAccessCardService {
... (implementation)
}
```

Naturally, this mechanism can be used to implement a FileAccessCardService for any smart card offering the required functionality, and any number such CardServices could be registered in a single OPENCARD installation. The SmartCard getCardService() method will search through the registered factories to obtain the correct FileAccessCardService implementation for the inserted card.

Standard application interfaces

In the process of creating the OPENCARD FRAMEWORK reference implementation, a number of standard CardService interfaces have been defined. For detailed documentation on these interfaces, please see the API documentation.

- opencard.opt.signature.SignatureCardService provides an interface for digitally signing data and hashes using a public key algorithm and private key stored on a smart card.
- opencard.opt.signature.KeyImportCardService allows new PKA private and public keys to be stored on a smart card.
- opencard.opt.signature.KeyGenerationCardService supports generation of new PKA key pairs.
- opencard.opt.applet.mgmt.AppletAccessCardService allows applets on the card to be listed and to getthe information required to select an applet.
- opencard.opt.iso.fs.FileAccessCardService provides for ISO 7816-4 file access functionality.
- opencard.opt.iso.fs.FileSystemCardService supports file creation, deletion, invalidation, and rehabilitation for file system cards.

Defining your own interface

It is possible to define card services that implement any required interface - not just the standard interfaces described above. This can be done simp ly by extending the `CardService` class (described in following sections).

```
public class FooBarCardService  extends CardService {
   public FooBar getFooBar() { ... implementation ... }
   ..... (rest of implementation)
}
```

Interface standardization

The `CardService` work group of the OpenCard Consortium is the official board for the approval of new standard services. If you define a new interface that you believe to be of general interest to the OpenCard community, you are encouraged to submit it for standardization. Please contact the OpenCard Consortium through the `http://www.opencard.org` web site for further information.

Framework interfaces

A Java class becomes an OCF card service by inheriting from the `CardService` class. The `CardService` class provides methods for use by the application as well as functions used by the card service implementation.

There are also several methods meant for administrative use by the OpenCard Framework itself. These methods will not be covered here.

CardService methods for implementation use

Communication with the card is carried out through the framework `CardChannel` class. `CardChannel` must be allocated before use and de-allocated after use. To do this, three functions are necessary.

- `allocateCardChannel()` makes sure that `CardChannel` is available for communication with the card. If another thread is already communicating with the card, this function will block until the active thread releases `CardChannel`. After successful completion of this call, the current thread will have exclusive access to `CardChannel`.
- `getCardChannel()` returns a reference to a `CardChannel` object that can be used for communication with the card.
- `releaseCardChannel()` - signals that `CardService` is done using `CardChannel`. It should be noted that this call does not destroy the `CardChannel` object, but merely signals to `CardServiceScheduler` that `CardService` is finished communicating with the card.

Another method that can be useful is `getCHVDialog()`. This function returns a cardholder verification dialog object that can be passed into the `CardChannel` object when communicating with the card.

CardService methods for application use

- The `getCard()` method returns a reference to the smart card object associated with this `CardService`.
- The `setCHVDialog()` method allows the application to pass a `CHVDialog` object that will be used to obtain the password from the user.

CardServiceInterface

The methods for application use are also described by the CardServiceInterface Java class. This convenience interface allows card service programmers to include access to the setCHVDialog() and getCard() functions in their interfaces, which obviates the need to downcast. The interface definition extends CardServiceInterface, and the implementation extends CardService.

Subclassing CardService

When designing OpenCard, care was taken to keep dependencies on external packages to a minimum. In particular, dependencies on the Java reflection API were avoided, since this package will likely not be available on embedded devices.

This resulted in a somewhat non-intuitive mechanism for instantiating card services. Instantiation is performed by CardServiceFactory in two steps. First, the default constructor runs, and then the initialize(CardServiceScheduler, SmartCard, boolean) method is called.

If a default constructor is provided, it is important that super() be called. Similarly, the initialize() method can be overridden to perform initialization of the implementation, but if this is done, it is required that super.initialize() be called.

Example

These code fragments illustrate the concepts presented in this section.

```
// declare interface, allowing access to application relevant CardService functions -
public interface FooBarCardService  extends CardServiceInterface {
   public FooBar getFooBar();
}

// class implementing the card service interface -
public class BobsFooBarCardService extends CardService implements
            FooBarCardService {

      // default constructor - 1st step of two-step construction -
      public void BobsFooBarCardService() {super();}

      // 2nd step of two-step construction -
      protected void initialize(CardServiceScheduler  sched,
                                SmartCard card,
                                booloean blocking) {
         super.initialize(sched, card, blocking);
         .... (initialize implementation) ....
   }

   public FooBar getFooBar() {
         .... perform setup stuff ...

         // allocate CardChannel, communicate with card, release channel -
         try {
            allocateCardChannel();
            CardChannel chan = getCardChannel();
            .... do some work with card ... make a FooBar ...
         }
         catch ( ... various exceptions ... ) { ... do appropriate thing ... }
         finally {releaseCardChannel();}
         return foobar;
      }
}
```

CardChannel

This class is the primary means through which the card service implementation communicates with the smart card. Just as the SmartCard object represents the inserted smart card, CardChannel abstracts the ISO 7816-4 logical channel concept. As such, there will be one CardChannel object for each logical channel supported by the card.

Before use of CardChannel is described, several helper classes and concepts must be covered.

The APDU classes

The CommandAPDU and ResponseAPDU helper classes are used by CardChannel for communication with the card. As can be imagined, CommandAPDU is sent to the card, and ResponseAPDU contains the data returned by the card.

These classes provide methods for accessing and setting the data to be sent to the card on a byte and byte array basis. In addition, ResponseAPDU contains methods for accessing the status word bytes returned by the card. Both APDU classes are defined so that they can be instantiated once and reused for subsequent calls. This feature can improve performance since it reduces the number of objects created and destroyed. To do this, the APDU is created with the maximum expected buffer size during the initialization phase. Each time the APDU is used, the length is first reset to 0, and then the new data is appended.

```
... (initialization) ...
CommandAPDU cmd = new CommandAPDU(MAX_SIZE);
...

... (before each use) ...
cmd.setLength(0);
cmd.append(NEW_APDU_BYTES);
```

The CardChannel state

The CardChannel state allows one arbitrary object to be associated with the channel. Since CardServiceScheduler maintains the CardChannel objects even after they have been released by the card services, the object stored as channel state is also maintained across card service calls.

This mechanism is also useful for passing data between related card services. For instance, data about the last selected application or the last security state reached could be stored. Naturally, card services communicating in this manner must agree on a common data representation.

The state is stored using the setState() and read using the getState() methods.

Cardholder verification

The OpenCard Framework was designed to provide a wide range of flexibility in obtaining CHV (cardholder verification) from the user. Depending on the card terminal device capabilities and the application, the cardholder verification may be obtained from different sources.

Some card terminal devices are equipped with a PIN pad and are capable of accepting CHV input from the user and passing it directly to the card. The OCF card terminal package provides a special interface (opencard.core.terminal.VerifiedAPDUInterface) that allows such a device to be recognized and used in a standard manner.

Since the APDU used for cardholder verification is card specific, the device must be provided with a partial APDU along with information that allows the device to complete the APDU by inserting the CHV, encoded properly, at the correct location. The completed APDU is then sent to the card and the resulting response APDU is returned to the card service implementation.

When the CHV is to be obtained from a GUI running on a workstation, the concept is similar. Again the card service implementation must generate a partial APDU and provide information on how to encode the CHV information and insert it into the partial APDU for transmission to the card.

In either case, the descriptive information for inserting the CHV information into the partial cardholder verification APDU must be placed in the `CHVControl` class, which is a simple container for such information. Required data is CHV offset within the APDU and the name of the encoding mechanism to be used. Coding mechanism names are described by the `CHVEncoder` interface. The other fields may be set to null when not needed.

The OpenCard Framework provides a default GUI dialog box for obtaining the cardholder verification when running on a workstation with full AWT (Advanced Windowing Toolkit) capability. It is recognized, however, that some applications will require a password entry dialog that is more closely integrated (from the look-and-feel point of view) into the application.

By implementing the `CHVDialog` interface and passing the implementation class to the card service using the `setCHVDialog()` method, the application can override `OpenCardDefaultCHVDialog`.

The `CHVDialog` contains only one method - `getCHV(int number)`. This method returns a string representing the CHV indicated by the CHV number. Although this interface will often be implemented by some type of GUI dialog or window class, `CHVDialog` itself has no dependencies on the AWT package. This means that the class implementing `CHVDialog` is free to obtain the CHV string from any source - from a command line interface, from a constant string in the source code (Yikes!!), or even from a PIN pad if OpenCard happens to be running on an actual high-function terminal platform.

Communicating with the card

After the card service implementation has successfully allocated and obtained a card channel, the implementation has exclusive access to the card until the channel is released.

The `sendCommandAPDU()` method provides the most straightforward possibility for a card service to communicate with the card. The command APDU provided as a parameter is sent to the card, and the card response is returned in a `ResponseAPDU`.

The `sendVerifiedAPDU()` method is used when cardholder verification information is to be sent to the card. This method requires a number of parameters:

- a partial command APDU,
- `CHVControl` parameters,
- a `CHVDialog` and, last but not least,
- a time-out value.

To use this method, a partial command APDU for cardholder verification along with a descriptive `CHVControl` object must be created as described in the preceding section. The CHVDialog set by the application is retrieved from the base CardService class using its getCHVDialog() method.

When executing the sendVerifiedAPDU() method, the CHVControl class will access the card terminal to determine its capabilities. If the terminal device is capable of performing CHV, the CHVControl information and the partial APDU will be sent to the terminal. Otherwise, if a CHVDialog object was set, it will be used. Finally, if all else fails, OpenCardDefaultCHVDialog will be used. The card service implementation is not involved in this decision.

Example

The previous example is enhanced with the main concepts from this section.

```
// class implementing the card service interface -
public class BobsFooBarCardService extends CardService implements FooBarCardService {

   private CommandAPDU cmd = new CommandAPDU(MAX_SIZE);
   private ResponseAPDU rsp;

   ... do initialization ....

   public FooBar getFooBar() {
      // allocate CardChannel, communicate with card, release channel -
      try {
         allocateCardChannel();
         CardChannel chan = getCardChannel();

         // carry out cardholder verification
         cmd.setLength(0);
         cmd.append(CHV_APDU_BYTES);
         CHVControl chvctrl = new CHVControl( "Enter your password", 1,
                  CHVEncoder.STRING_ENCODING, 0,
                  null);
         CHVDialog chvdlg = getCHVDialog();
         rsp = chan.sendVerifiedAPDU(cmd, chvctrl, chvdlg, -1);

         .... check sw1 & sw2, do some work with card ... make a FooBar ...
      }
      catch ( ... various exceptions ... ) { ... do appropriate thing ... }
      finally {releaseCardChannel();}
   }
}
```

Implementing secure messaging

Sometimes secure messaging is required to access smart card files. In secure messaging, cryptographic functions are used to protect and authenticate the data passed to and from the card. Since the card service implementation maps function calls from the application to one or more APDU command / response pairs, the implementation must be aware of secure messaging if supported by the card. However, the card service implementation will not in general implement its own cryptographic functions. The cryptographic functions can be installation and / or application specific. Also, the cryptographic keys needed to access files on the card are application dependent. The algorithms and keys may be available in software and data format in the client machine, or they may be contained in a security access module (SAM). There have been a number of attempts to create a cryptographic framework that encapsulates key and algorithm storage, notably PKCS#11, CDSA, and the JAVA CRYPTOGRAPHIC ARCHITECTURE. Rather than defining a new cryptographic framework specifically for use in secure messaging, OPENCARD defines a minimal infrastructure that allows cryptographic algorithms and data to be handled as such without constraining the interface.

Credentials

The Credential interface defines no members, but is used to tag classes containing cryptographic keys and/or algorithms. Implementers of card services requiring secure messaging will have to implement or obtain appropriate credentials from CredentialStore objects.

Credential interfaces will be defined (by the card service implementer or perhaps by a third-party provider) to provide appropriate functionality for the card service at hand. Depending on availability, credentials can implement the cryptographic functionality directly or can access a security module.

The CredentialStore class can be used as a container for related credentials. This abstract class must be extended to support a specific card operating system, since the method through which keys are identified and selected vary from card to card. Customization of the CredentialStore class requires work in two areas:

1. The boolean supports(CardID) method of the CredentialStore class must be implemented. This is required for identification of the store since it is card operating system dependent.
2. An identifier object for the keys contained in the credential store must be defined. This identifier is used with the storeCredential() and fetchCredential() methods to specify the desired credential. Typically, this identifier class will be defined by the card service implementer and will abstract the manner in which keys are specified on the card. The only restriction on the identifier definition is that hashCode() and equals() methods be defined appropriately for use in a hash table.

Card interoperability

Credential as well as CredentialStore being card service implementation specific, it would seem difficult to achieve card interoperability.

To help with this, another concept - CredentialBag - was introduced. CredentialBag is simply a container class for CredentialStore objects. The addCredentialStore() and getCredentialStore() methods allow objects to be stored and retrieved.

The CredentialStore objects are retrieved from CredentialBag by CardID. This allows the card service implementation to query the key bag for an appropriate key store object. To achieve card interoperability, the application fills CredentialBag with key stores for all supported cards. After a card has been inserted and the application has obtained a card service that implements secure messaging, the application will pass CredentialBag to the card service. The card service implementation will use CredentialBag's retrieval functions to obtain an appropriate CredentialStore for the inserted card.

Interfaces for CardService

On a smart card, a security domain is defined by a set of cryptographic credentials. An object on the card - a file for example - can belong to one or more security domains. In fact, on some smart cards, notably the IBM MFC cards, an object can belong to a different security domain for each operation allowed on the object. An object can belong to one security domain for a *read binary* operation and to another for an *update* operation.

On a smart card, the security domain is specified by some characteristic. The characteristic specifying the security domain will differ depending on card type. For example, it will be different for JAVACARD-based smart cards than for ISO file system based smart cards.

This concept of security domain is abstracted in the OpenCard Framework as the SecurityDomain interface. To allow support for varying card types, it is defined as a tag interface only.

The SecureService interface allows applications to pass credential bags to the card service. All card services that provide secure messaging functionality must implement the SecureService interface. This interface provides one new function - provideCredential() - that accepts a CredentialBag and a SecurityDomain object as parameters. The security domain object specifies the object characteristic for which the credential bag is valid.

File system based cards typically use a partial subdirectory in which an object resides to specify the security domain. For this reason, the opencard.opt.iso.fs.CardFilePath class implements the SecurityDomain interface. For such cards, the CardFilePath object would represent the object characteristic for which a credential bag is valid. When performing secure messaging for file access, the card service implementation would select the credential bag to use based on the active card file path.

CardServiceFactory

Registry mechanism overview

The OpenCard Framework requires a CardServiceFactory to instantiate a card service. The factory is in a sense the middleman between CardServiceRegistry and card services. A CardServiceFactory knows how to create card services for a smart card or for a family of cards.

When the application requests a card service, CardServiceRegistry goes through its list of registered instances of CardServiceFactory, querying each in turn whether it supports the inserted card. When aCardServiceFactory supporting the card is found, the registry attempts to use that factory to instantiate the requested card service. If successful, the card service is returned to the caller; otherwise the search continues.

Implementing instances of CardServiceFactory

All instances of CardServiceFactory must inherit from the opencard.core.service.CardServiceFactory common base class. The interface of this base class has changed between OCF 1.1 and OCF 1.1.1. Existing card service factories still implementing the outdated interface should now inherit from the opencard.opt.service.OCF11CardServiceFactory common base class.

Card service factories implemented with OCF 1.1.1 or later must overwrite two abstract methods that are used by the OpenCard Framework to instantiate services - getCardType() and getClasses(). The constructor of the CardServiceFactory can be empty.

The getCardType() method accepts a CardID object which basically represents the ATR obtained from the card and returns a CardType.

The card type object returned should either be the reserved instance CardType.UNSUPPORTED or another instance of card type that allows the factory’s getClasses method to decide which card services can be instantiated. If the card service factory supports a whole family of cards from one card manufacturer the card type could for example contain the information about the card operating system version of the inserted card. If the CardID is not sufficient to determine the card type the CardServiceScheduler can be used to allocate a CardChannel to communicate with the card.

The getClasses() accepts the CardType object determined in the previous step as a parameter and returns an enumeration of card services supported by the factory for the specified card. This method is used by the CardServiceFactory superclass method called by CardServiceRegistry when instantiating a CardService class for CardServiceRegistry.

Example

The following example code shows the implementation of the factory that supports theFooBarCardService:

```
public class FooBarCardServiceFactory extends CardServiceFactory {
public FooBarFactory () {}

// implement abstract method -
protected CardType getCardType(CardID cid,
                               CardServiceScheduler scheduler)
  throws CardTerminalException
{

   byte[] historicalBytes = cid.getHistoricals();
   // analyze historicals
   if ((historicalBytes[0] != ...) || (historicalBytes[1] != ...))
     return CardType.UNSUPPORTED;

   // find out information about card
   ...
   int cardOSVersion = ...;

   return new CardType(cardOSVersion);
}

// defines supported services
static {
   foo_classes = new Vector();
   foo_classes.addElement(FooBarCardSvc.class);
}


protected Enumeration getClasses(CardType type)
{
   switch (type.getType()) {
   case ...:
     return foo_classes.elements();
   case...:
     ...
}
```

PrimaryCardServiceFactory

Of all the registered factories that support a specific smart card, one and only one may implement the PrimaryCardServiceFactory interface. When a card is first inserted, CardServiceRegistry will call the setupSmartCard(SlotChannel) method of the primary card service factory for that card.

Using the SlotChannel object provided by the card service registry, PrimaryCardServiceFactory can communicate with the card in order to carry out any required initialization.

One possible use for this feature would be communication speed selection.

Support for JavaCards and other Multi Application Cards

In this Chapter, we present a set of classes that can be used to implement card services for JavaCards and other Multi Application Cards, e.g. EMV-compliant smart cards. These cards have in common that they allow for application selection by application identifier (AID). In OCF 1.2, we added the package `opencard.opt.applet` to support implementation of card services in the form of proxy classes for applets on JavaCards, applications on EMV/ISO cards etc. The `applet` package contains base card service classes that allow for multiple threads to concurrently use derived smart card application proxies. Multiple proxy instances associated with the same applet on the card can exist at the same time and are properly synchronized. Selection and deselection of applications on the card is detected and handled.

State Objects

As applications and the card itself have a state that is affected by application selection, a representation for these states is needed. OCF 1.2 provides base classes from which programmers may derive state classes for their own applications or cards. We differentiate between application state (`AppletState`) and card state (`CardState`).

AppletState

Application states are to be used for representing the state of an application to applet proxies associated with that application. The state of an application may reflect whether external authentiction has been performed since selection or wether a password has been provided, for example.

CardState

A card state object encapsulates the state of a multi application card. It must at least contain information about the currently selected application. OCF provides a base class for card states, named `CardState`. All instances of the class `BasicAppletCardService` associated with the same physical card share a common `CardState` object to ensure a consistent view.

Selection

Applet selection is a basic mechanism that is required by card services for multi application cards. OCF provides an interface for applet selection and an ISO-conformant implementation for this interface.

AppletSelector

This interface defines the minimum selection features that are necessary for working with card-resident applets. It specifies a low-level selectApplet method that will be called by `CardServices`. The caller must provide an already allocated `CardChannel` for communicating with the card.

ISOAppletSelector

The class `ISOAppletSelector` is a very simple helper class that implements the interface `AppletSelector`. It only provides the "selectApplet" functionality as specified in `AppletSelector`. It is a basic implementation based on ISO7816-4 and ISO7816-5 that does not check warning/errors code but provide them to caller.

Proxy Base Classes

The following classes provide a mechanism to transparently perform the selection of applets when needed.

BasicAppletCardService

The class `BasicApppletCardService` offers methods to applications or proxies derived from it, which sends a given command APDU to the card applet identified by the given application identifier and returns the result APDU. See also the description of `CardState` and `AppletState`.

AppletProxy

`AppletProxy` is the base class for all applet proxies. This class is derived from the class `BasicAppletCardService`. It is aware of the applet to which the applet proxy is associated. It provides methods similar to those of the base class, except that the application identifier parameter is not necessary because every `AppletProxy` knows the AID of the applet associated with it.

Support for SCQL Database Smart Cards

In this Chapter, we present a set of classes that can be used to implement card services for ″Database Smart Cards″.

What we call here ″Database smart cards″ are smart cards (or applets) that integrates an engine of SCQL (Smart Card Query Language) relational data base. The database concept is based on SQL (ISO 9075).

Such cards allow to store personal data in database structures in a secure way. They include security mechanisms such as PIN code authentication, login / password authentication or internal / external authentication via a DES algorithm enabling the user to secure his / her personal data optimally.

As specified by the [ISO 7816-7] international standard, the dialog with the card is based on exchanges of APDU commands which are structured as byte arrays. The set of commands managed by the card (at database level) is a subset of the SQL-92 command set.

As of today, this subset does not authorize neither join nor multiple predicates in ’where’ clauses. In spite of the numerous physical constraints due to the memory size of the database engine and databases themselves, the provided functionalities are nearly similar to those offered by a classical database. Indeed, this type of card can manage users, users profiles, privileges about the different objects of the base(s), transactions (commit and rollback notions) and tables / views / dictionaries.

Please refer to the ISO norm [ISO 7816-7] for more details on database smart cards.

Package features

In OCF 1.2, we added the `opencard.opt.database` package to support implementation of card services dedicated to Database Smart Cards.

Layers of abstractions

The `opencard.opt.database` package is composed of two layers of abstractions for helping to develop CardServices for Database Smart Cards, i.e.:

1. an interface layer, `DatabaseCardService`, to allow interoperability among competitive implementations for smart cards or applets that all respect compliance with [ISO 7816-7], and
2. a general purpose concrete implementation, `BasicDatabase`, that can be used out-of-the-box by programmers of applications that wish to use a database card, and that can also be considered as a base class for the implementation of specific CardServices. In both cases, the programming effort will be reduced by

the usage of the framework, and also if everybody uses the same basic implementation and propose enhancements for it, the quality of this component will increase.

Miscellaneous

In addition, the package also provides:

- a set of specialized exceptions, all inheriting from the `SCQLException` class (the base class for all exceptions related to the use of a SCQL Database Smart Card),
- `DataObject`, a utility class used by `BasicDatabase` for parsing parameters according to the standard (e.g., privileges, user profiles, etc.), and
- `SecurityAttribute`, the base class for security attributes DO (i.e., Data Objects) as specified by the standard. It is currently mostly a wrapper for a byte array, as the ISO7816-7 standard does not specify what information should be provided in a security attribute DO, and in what form. In the case where the chosen semantics is to use the ASCII codes of a ″String″ password, a constructor is provided in addition to the default one (using a byte array).

The next sections describe in details the interface of the package and an example of use of the basic implementation.

Interface Details

The most important part of the provided API is the `DatabaseCardService` interface. This Java interface defines:

- Constants as specified by the 7816-7 norm (i.e., class byte, coding for SCQL, transactions, or user operations, coding for comparison operators, coding for privileges, and error codes),
- Constants as specified by the 7816-6 norm (i.e., TLV tag values for card holder certificate and name) that are used for encoding a `PRESENT_USER` operation with security attributes, and
- Methods to handle database smart cards, as specified by the 7816-7 norm: `createTable`, `createView`, `createDictionary`, `dropTable`, `dropView`, `grant`, `revoke`, `declareCursor`, `open`, `next`, `fetch`, `fetchNext`, `insert`, `update`, `delete`, `begin`, `commit`, `rollback`, `presentUser`, `createUser`, `deleteUser`. The method names are the ones that are specified by the norm, except that they also follow standard Java naming guidelines. Please refer to the [ISO 7816-7] ISO norm for more details about these commands semantics.

Example of use

This part gives a detailed description of the way the `BasicDatabase` or one of its sub-class can be used. This allows to understand their operating mode and to observe how easy it is to use them. In order to do this, we will take the example of a simple Java program which is connected to a database card and which performs a few SCQL operations.

Program

The program source code is listed below:

```
import opencard.opt.database.*;

public class TestDatabase {

    public static void main (String [] args) {
        int i;
        String[] result;

        try {
            // Initialize the framework
            SmartCard.start ();
            System.out.println ("Waiting for a 7816-7 SCQL smart card...");
            SmartCard sm
                = SmartCard.waitForCard (new CardRequest(BasicDatabase.class));

            // Get a card service and perform SCQL commands
            BasicDatabase cs = (BasicDatabase)
                sm.getCardService(BasicDatabase.class, true);

            // Try to send a PRESENT USER to the card
            cs.presentUser("GUEST", new SecurityAttribute("guest"));

            // Try to send SCQL commands to the card to read the DIARY table
            cs.declareCursor("DIARY","*","");
            cs.open();
            try {
                while (true) {
                    result = cs.fetch();
                    for (i=0 ; i < result.length ; i++) {
                        System.out.println (" - DIARY("+i+") = "+result[i]);
                    }
                   ; System.out.println ("---");
                    cs.next();
                }
            } catch (EndOfTableReachedException scqle) {
                System.out.println ("");
                System.out.println ("Warning: End of table reached.");
            }

            // Try to create a new table 'FOOTABLE'
            try {
                cs.createTable("FOOTABLE", "FOO,BAR,GEE", new SecurityAttribute(""));
            } catch (ObjectAlreadyExistsException scqle) {
                System.out.println ("Warning: Table already exists...");
            }
            cs.insert("FOOTABLE", "foo1,bar1,gee1");

            // Shutdown the framework
            SmartCard.shutdown ();
        }
        catch (SCQLException scqle) {
            System.out.println ("SCQL Exception: ");
            scqle.printStackTrace();
            if (scqle.getMessage () != null) {
                System.out.println ("details:");
                System.out.println (scqle.getMessage () );
            }
        }
        catch (CardTerminalException cte) {
            System.out.println ("CardTerminalException: ");
            System.out.println (cte.getMessage () );
        }
    }
}
```

Description

First of all, it imports the required classes, i.e., the classes and interface of the opencard.opt.database package.

```
import opencard.opt.database.*;
```

The next line allows Framework initialization.

```
            SmartCard.start ();
```

Then, a database smart card is waited to be inserted in the card reader. This is a blocking method call.

```
SmartCard sm
    = SmartCard.waitForCard (new CardRequest(BasicDatabase.class));
```

The next command allows to instantiate a CardService for the database smart card. Instantiation is processed by the relevant factory.

```
BasicDatabase cs = (BasicDatabase)
    sm.getCardService(BasicDatabase.class, true);
```

The `presentUser(login, securityAttribute)` command allows to be connected with the database.

```
cs.presentUser("GUEST", new SecurityAttribute("guest"));
```

A cursor is declared on an existing table. This cursor actually selects all the columns of the view (i.e., ″*″). There is no limit on selecting (i.e., the third parameter remains empty).

```
cs.declareCursor("DIARY","*","");
```

The information which meets the conditions defined by the cursor is retrieved via a loop in which each line pointed out by the cursor is read (`fetch()` command). Then, the cursor moves on the next line and meets the conditions (`next()` command). The result returned by the `fetch()` command is simply displayed. This command returns an array of character string, each column of the table corresponds to a value located in a column of the table in the database.

```
try {
    while (true) {
        result = cs.fetch();
        for (i=0 ; i < result.length ; i++) {
            System.out.println (" - DIARY("+i+") = "+result[i]);
        }
        System.out.println ("---");
        cs.next();
    }
```

The result display loop is stopped when the CardService returns an exception indicating that the end of the table has been reached (i.e., `EndOfTableReachedException`).

```
} catch (EndOfTableReachedException scqle) {
```

A ″FOOTABLE″ table containing 3 columns entitled ″FOO″, ″BAR″, and ″GEE″ is created.

```
try {
    cs.createTable("FOOTABLE", "FOO,BAR,GEE", new SecurityAttribute(""));
```

Via the following instructions, one line is inserted in the ″FOOTABLE″ table.

```
cs.insert("FOOTABLE", "foo1,bar1,gee1");
```

At the end, the Framework is shutdown.

```
SmartCard.shutdown ();
```

As we can observe, the use of such CardServices remains simple and accessible by all developers having a minimum knowledge about database smart cards.

In addition, an application can request a CardService implementing the `opencard.opt.database.DatabaseCardService`, thus ensuring interoperability with all database smart cards or applets that are compliant with the ISO 7816-7 standard.

Acknowledgements

We are indebted to numerous colleagues from BULL, GEMPLUS, IBM, SCHLUMBERGER, and SUN for their feedback and constructive criticism in the design and implementation phases — without their contributions, support, and enthusiasm, OCF would probably not have become an industry standard.

Glossary

The following is a glossary of OpenCard-related terms.

APDU. An acronym for **a**pplication **p**rotocol **d**ata unit. Interactions with *smart cards* occur by exchanging pairs of APDUs and are initiated by the *external application*.

API. An acronym for **a**pplication **p**rogramming **i**nterface.

applet. A Java-based ″mini-application″ which runs in a browser on the client side.

application. As defined by ISO 7816, the card-resident component (consisting of data and functions) of software interacting with the card-external component.

application developer. A software programmer specializing in the development of applications. One of the chief beneficiaries of the OpenCard Framework.

AppletAccessCardService. A special OCF card service for dealing with the various different applets residing on a *smart card*.

asymmetric algorithm. A type of cryptographic operation using one key for encrypting data and another key for decrypting the resultant encrypted data. Together, the two keys are referred to as a *key pair* (also known as *public/private keys*).

ATR. An acronym for **a**nswer-**t**o-**r**eset. The ATR is the initial information emitted by a *smart card* upon being reset and powered up.

authentication. The process of verifying the identity of the participants in an exchange of electronic data. When the card-external application authenticates itself to the card-resident application, this is called External Authentication. Conversely, when the card-resident application authenticates itself to the card-external application, this is referred to as Internal Authentication. *Cardholder Verification (CHV)* is when the cardholder authenticates him or herself to the card.

authorization. The security process to decide whether a service can be given or not.

Card Acceptance Device (CAD). Synonym for *card reader* or *card terminal*.

card authentication. The security process for verifying the genuineness of an inserted *smart card*.

cardholder. The legitimate holder of a *smart card*.

Cardholder Verification (CHV). The process of checking whether the person presenting a card to the card system is indeed the legitimate holder. Usually, a secret number or password known only to the cardholder is used.

CardID. An OCF class for the identification of the card type based on the card's *ATR*.

card initialization. The process of writing initialization data to the *smart card*'s *EEPROM*. The EEPROM image generated during *layout definition* is then transferred to the *smart card*.

card issuer. An institution which issues cards to *cardholders*.

cardlet. A word formed from two parts: *(smart) card* and *applet*.

CardManagementService. A special OCF card service defining a high-level *API* for installing, removing, blocking, and unblocking the various different applications residing on a card in an issuer-independent fashion.

Card Operating System (COS). The microcode contained in a *smart card*'s ROM, used for communicating with the smart card, managing security, and managing data in the smart card.

card personalization. The process of writing *cardholder*-specific data (e.g. serial number, cardholder's name) to the *smart card*. After *card initialization*, the smart card's *EEPROM* reflects the basic data structure as defined in the *layout definition* and may contain some initial data that is constant across all cards. During personalization, the information peculiar to an individual cardholder is written to the card prior to issuing it.

card reader. An I/O device attached to the computing platform into which the smart card is inserted. Roughly synonymous with*Card Acceptance Device (CAD)*, *Interface Device (IFD)*, or *card terminal*.

card recognition. The process of checking whether an inserted *smart card* has the correct physical and electrical characteristics.

CardService. Any one of a variety of OCF components which make *smart card* functions available to the application programmer.

CardServiceFactory. Associated with each instance of CardService is an instance of CardServiceFactory capable of constructing it.

CardService layer. A layer of OCF which provides the basic infrastructure for accessing card services.

`CardServiceRegistry`. A system-wide OCF class for keeping track of the various different instances of `CardServiceFactory` which are available. Application developers can configure `CardServiceRegistry` using the `OpenCard.services` property.

`CardServiceScheduler`. Responsible for managing `CardChannel` objects.

card system. A body establishing a set of rules governing the issuance and usage of *smart cards* carrying its mark.

`CardTerminal` class. A software construct. Instances of the `CardTerminal` class represent actual physical card terminal devices.

card terminal. (1) A frequently-used synonym for *card reader*, but in fact more sophisticated. A physical device which performs some interactive function (e.g. reading, writing) with an inserted *smart card*. The simplest card readers merely provide basic card input-output (I/O) functionality via a single slot to insert the card. Card terminals offer multiple slots or include a PIN pad and a display. (2) When appearing in monospace (`CardTerminal`), the OCF abstraction (device driver code) for specific card terminals.

`CardTerminal` layer. A layer of OCF containing abstractions for card terminals.

`CardTerminalFactory`. An OCF class of services which are capable of creating individual instances of `CardTerminal`.

`CardTerminalRegistry`. A system-wide OCF class of services for keeping track of all available instances of `CardTerminal`. Application developers can configure `CardTerminalRegistry` using the `OpenCard.terminals` property.

CHVDialog. An interface registered via the `CardService` object's `setCHVDialog()` method and implemented by a default GUI component to enter the *cardholder verification* data.

CommandAPDU. An *APDU* sent to a *smart card* via the *card reader*'s device driver.

component. The smallest selectable set of elements includable in a package.

communication layer. One of the three layers of `CardTerminal` implementation. This layer provides methods for handling the transfer of the data between the host and the *card reader*.

confidentiality. The prevention of the unauthorized disclosure of information.

constructor. A special function used in object-oriented programming to initialize the state of a program object. Generally speaking, a constructor must be called whenever a new object is created.

credential. (1) Typically, cryptographic keys or certificates made available in (card-external) applications which fit the access conditions of the files which the given application is supposed to interact with. (2) When appearing in monospace (`Credential`), OCF parlance for an interface typically representing a *cryptographic key* and the associated algorithm for performing *cryptographic functions*. OCF's concrete implementations of `Credential` include `RSASignCredential` and `DSASignCredential`.

`CredentialBag`. A container class for `CredentialStore` objects. It offers retrieval functions which the card service implementation can use to obtain an appropriate `CredentialStore` for inserted *smart cards*.

Credential Store. In OCF, an abstract class containing credentials for a smart card.

cryptographic algorithm. An algorithm for transforming confidential data so as to encrypt or decrypt its information content. Cryptographic algorithms can be either symmetric (the same key is used to both encrypt and decrypt the data) or asymmetric (different keys are employed for encrypting and decrypting).

cryptographic functions. Any of various tools (typically based on *cryptographic algorithms*) for encrypting and decrypting information, thus making it unintelligible to any but authorized persons (who must undergo *authentication*). When employed in *smart cards* for *secure messaging*, cryptographic functions can be installation and/or application specific.

digital signature. An asymmetric cryptographic operation on data proving the data's origin and integrity to the recipient, and thus protecting against *forgery*.

EEPROM. An acronym for **e**lectrically **e**rasable **p**rogrammable **r**ead-**o**nly **m**emory. A *non-volatile* memory technology in which data can be electrically erased and rewritten.

external application. The program code running on computing platform (personal computers, network computers, automatic-teller machines, etc.) interacting with *smart cards*.

`FileAccessCardService`. A particular card service incorporated by OCF for dealing with file-oriented smart cards as per ISO 7816–4.

forgery. The illicit copying and alteration of intercepted information. A form of tampering.

function layer. One of the three layers of `CardTerminal` implementation. This layer encapsulates the detailed knowledge about your *card reader*.

hashing. The one-way transformation of data having an arbitrary length into a fixed-length digest.

initialization. A step in the manufacture of *smart cards* in which the basic data common to a batch of cards is loaded onto their chips.

Interface Device (IFD). A synonym for *card reader*.

interface layer. One of the three layers of `CardTerminal` implementation. This layer is derived from the abstract `CardTerminal` class and provides methods for interfacing with the upper part of the framework.

key generation. The task of creating a public / private key pair to be used with public key cryptography. In OCF, `KeyGenerationCardService`'s `generateKeyPair` method can be used to obtain this functionality from an inserted *smart card*.

`KeyGenerationCardService`. An extension of OCF's `SignatureCardService` interface offering methods for generating a key pair for a card-resident public key algorithm and for reading the *public key* part of the pair for use outside of the card.

key importation. A task performed in OCF by the `KeyImportCardService`'s `importPrivateKey` and `importPublicKey` methods (non-validating) or its `importAndValidatePrivateKey` and `importAndValidatePublicKey` methods (validating). It allows the *public/private keys* to be stored in a smart card.

`KeyImportCardService`. An extension of OCF's `SignatureCardService` interface offering methods for the importation and subsequent in-card verification of keys for *asymmetric key algorithms*.

layout definition. The process of generating an *EEPROM* image from a high-level definition of the EEPROM layout. Most smart card applications maintain information typically kept in EEPROM-based files on the card's file system. Layout definition is about identifying the information items that should go in the card and defining an appropriate file structure in the card. The latter includes specifying the type of file (transparent, record-oriented, cyclic), file names and / or identifiers, access conditions, initial data, etc.

Master File (MF). A specially dedicated file in a smart card's file system representing the file system's root.

non-volatile. Said of memory storage technologies which are not dependent upon a power supply for storing data.

OpenCard Consortium. A consortium including 3GI, Bull, Dallas Semiconductors, First Access, Gemplus, intellect, International Business Machines Corp., Network Computer Inc., Newcom Technologies, Schlumberger, SCM Microsystems, Siemens, Sun Microsystems, UbiQ, and Visa International.

OpenCard Framework (OCF). The name of an object-oriented software framework for `smart card` access. It is implemented in the Java programming language and is located between a smart card-aware application or *applet* written in Java and the *card reader*. The terms OpenCard Framework and OCF are protected trademarks.

padding. Appending extra bits to either side of a data string up to a pre-defined length.

Personal Identification Number (PIN). The secret code used to authenticate a *cardholder*.

polling. The periodic querying of device status in order to detect status changes. In the case of OpenCard, polling can be used to detect smart card insertion and removal.

polling mechanism. A program construct; typically, a loop that runs periodically and that calls the appropriate status query methods for individual devices.

`PowerManagementInterface`. An OCF interface comprising the `powerUpCard()` and `powerDownCard()` methods, both of which take the slot number as a parameter and allow the supply of power to the smart card in the designated slot to be controlled. This optional `CardTerminal` function is useful in the case of long-lasting applications that only sporadically interact with *smart cards* in environments where low power consumption matters.

private key. The privately-held component of an integrated asymmetric key pair.

protocol. The procedures used by two or more computer systems for communicating with each other.

public key. The public component of an integrated asymmetric key pair.

ResponseAPDU. An *APDU* sent back by a *smart card* in response to a *CommandAPDU* it has received.

RSA. An acronym for an asymmetric cryptographic algorithm named after its inventors, Ron **R**ivest, Adi **S**hamir, and Len **A**dleman. It is used in public-key cryptography and is based on the fact that it is easy to multiply two large prime numbers together, but hard to factor them out of their product. RSA is a protected trademark of RSA Security Data, Inc.

SCQL. An acronym for **S**tructured **C**ard **Q**uery **L**anguage.

secure messaging. The use of *cryptographic functions* to protect and authenticate data passed to and from a *smart card* on a per-message basis. The card service implementation, while it generally does not implement its own cryptographic functions, must be aware of secure messaging if supported by the card.

Security Access Module (SAM). A unit in which *cryptographic algorithms* and keys may be stored.

SignatureCardService. A card service incorporated by OCF for generation and verification of digital signatures. SignatureCardService's signData and verifySignedData methods perform this operation.

signature generation. The act of creating a special data block known as a *digital signature* which allows changes in the signed data to be detected and data to be authenticated. In OCF, this is performed by SignatureCardService's signData method, which computes a hash value on the message and then encrypts this hash value using the *private key* of a key pair. The (time-consuming) computation of the hash value can also be performed outside of the OpenCard Framework.

signature verification. A task performed in OCF by SignatureCardService's verifySignedData method, which computes the decryption on the given signature using the *public key*, computes a hash value on the plain message, and then compares the hash value with the decrypted signature. The (time-consuming) computation of the hash value can also be performed outside of the OpenCard Framework.

slot. (1) A physical opening in a *card terminal* into / from which a *smart card* can be inserted / removed. (2) When appearing in monospace font (Slot), OCF's representation of such a physical opening and an instance of the Slot class.

smart card. Integrated circuit cards corresponding to the ISO / IEC standards 7816, JavaCards as defined in Sun's *JavaCard 2.0* specifications, or any other smart tokens (including smart accessories). Important functions include secure data storage and (optionally) cryptographic functionality.

SmartCard. An OCF class providing an entry point to OCF.

SmartCard object. An object with a pivotal role in interacting with a physical *smart card*. Can be obtained e.g. by calling the waitForCard method on the SmartCard class, which returns a SmartCard object.

TerminalCommand. An OCF interface comprising the sendTerminalCommand() method, which takes a byte array and sends it to the *card terminal*. Assuming an application knows the set of commands that a given card reader supports, it can use this optional CardTerminal function to control the card reader.

UserInteraction. An optional CardTerminal function. This OCF interface comprises the display() method to present a message to the *cardholder*, the clearDisplay() method to erase the message from the display, the keyboardInput() method to collect a string entered by the cardholder, and the promptUser() convenience method, which combines displaying the message to and collecting input from the cardholder into a single call.

Bibliography

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns — Elements of Reusable Object-Oriented Software*, ISBN: 0–201–63361–2, Reading, Massachusetts: Addison-Wesley Publishing Company, 1994.

Scott B. Guthery & Timothy M. Jurgensen, *SmartCard Developer's Kit*, ISBN: 1–57870–027–2, Indianapolis, Indiana: Macmillan Technical Publishing, 1998.

Henry Dreifus & J. Thomas Monk, *smartcards — A guide to building and managing smart card applications*, ISBN: 0-471-15748-1, New York: John Wiley & Sons, 1998.

Jack M. Kaplan, *SmartCards — The Global Information Passport*, ISBN: 1–850–32212–0, Boston, Massachusetts: Thomson Computer Press, 1996.

W. Rankl / W. Effing, *Handbuch der Chipkarten*, ISBN: 3-446-18893-2, Carl Hanser Verlag Muenchen Wien, 1996.

U. Hansmann, M. S. Nicklous, T. Schõck, F. Seliger *Smart Card Application Development Using Java*, ISBN: 3-540-65829-7, Springer Verlag Berlin, 1999.

Mike Hendry, *Smart Card Security and Applications*, ISBN: 0-89006-953-0, Norwood, Massachusetts: ARTECH House, Inc., 1997.

W. Rankl & W. Effing, *Smart Card Handbook*, ISBN: 0–47196–720–3, New York: John Wiley & Sons, 1997.

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

B0EB-0CFP-00