



OpenCard Framework

Dirk Husemann, Reto Hermann, IBM Corp.

Abstract

This overview of the OpenCard Framework outlines the architecture, highlights the prominent features, and describes the rationale behind the design.

The OpenCard Framework is a set of guidelines, based on open standards, that has been announced by Apple, International Business Machines Corporation, Inc., Netscape, NCI, and Sun Microsystems, Inc., for integrating smart cards with network computers. Smart card-aware applications and services are important to e-commerce because they provide mobility and security.

This document will not go into great detail about the OpenCard Framework-for that purpose, another set of documents will be published soon. This paper is organized as follows:

Background and Objectives

- OpenCard and the Smart Card Industry
- OpenCard and Smart Card Standards
- OpenCard and NC Platforms
- OpenCard and the API Specification
- OpenCard and Interoperability

Architecture

- CardTerminal
- CardAgent
- CardIO
- CardAgentExtensions
- At a Glance

Java Reference Implementation

- The opencard.terminal Package
- The opencard.agent Package
- The opencard.io Package
- Example

Summary

- Benefits
- Perspective
- Outlook

Glossary

- References

Questions, comments, and suggestions are welcome:
Send email to OpenCard@opencard.org.

OpenCard Framework: Background and Objectives

OpenCard and the Smart Card Industry

The OpenCard Framework brings together the smart card industry and application developers.

The OpenCard Framework is based on a model that divides the functions provided by the smart card industry into the following categories:

Application and Service Developers

Application and service developers build smart card applications and services comprising the card-resident application part and/or the external card-aware application.

Card and Terminal Providers

Card providers typically offer families of cards and card operating systems together with software development toolkits for a number of operating system platforms. Terminal providers typically offer families of card terminals with software driver support for a number of operating system platforms.

The OpenCard Framework specifies two interfaces:

- a high-level application programming interface that hides the characteristics of a particular provider's components from the application and service developers.
- a common provider interface that enables the seamless integration of smart card building blocks from different vendors.

Thus, compliance with the OpenCard Framework becomes mutually beneficial for both developers and providers. Application and service implementers that write to these interfaces can deploy their solutions in many different settings without having to change a single line of code. Providers adhering to these interfaces make their components directly usable for application and service developers, thus gaining easy access to a rapidly growing market.

OpenCard and Smart Card Standards

The OpenCard Framework builds upon open *de jure* and *de facto* standards.

The foundation of virtually all existing smart card standards is ISO 7816. ISO 7816 specifies physical and electrical characteristics of the smart card interface; it specifies formats and protocols for information exchange with smart cards and for functions provided by smart cards; and it specifies additional functionality, such as file organization and security behavior. Since it does not address smart card applications as such, initial application and service providers from the banking and finance, telecommunications, and health care industries have developed their own, open business-related specifications. Most prominent among them are the EMV Specifications, a *de facto* standard for electronic payment systems. Developed by Europay, Visa, and MasterCard, the aim of these specifications is to ensure that all smart cards operate across all card terminals and related devices, regardless of location, financial institution, or manufacturer. CEN 726 is a set of European standards for the telecommunications use of smart cards defining a security framework, application-independent requirements for card and card terminals, payment methods, and telecommunications features.

With the advent of network computing, new standardization efforts have emerged with the aim of facilitating the use of smart cards for such applications as secure network logon and secure electronic transactions (authentication, integrity, privacy, and non-repudiation). For this class of applications, which views smart cards primarily as a crypto token, RSA Data Security, Inc., developed the PKCS#11 standard, also known as Cryptoki API. It specifies an API that presents a common, logical view of crypto token devices to the application. The standard aims to be token-neutral and to support the

sharing of tokens by multiple applications.

OpenCard Framework supports both ISO 7816-compliant cards and others. It exploits specific *de facto* standards such as Cryptoki in the area of crypto token access. Where appropriate, OpenCard Framework takes into account other widely accepted standards, such as EMV or CEN 726. OpenCard Framework will also release new guidelines for NC applications such as network sign-on. Finally, OpenCard Framework will honor similar efforts, such as PC/SC, and will coexist with their respective implementations.

OpenCard and NC Platforms

The *OpenCard Framework* is platform-neutral.

NCs of many different flavors are emerging or have been announced. NCs are expected to be highly scalable and to span a product range from the palmtop to the desktop. NCs can be implemented on existing personal computers or built anew from the ground up.

NCs implemented on Java Platforms run only Java applications. The Java Application Environment is provided by the Java OS operating system or is built on top of a native operating system. For these platforms, smart card resource management will be provided by the OpenCard Framework implementation.

NCs implemented on Native Operating System Platforms can run native code applications and Java applications side-by-side. The Java Application Environment is provided by the Java run-time, which is built on top of native operating system services. The latter may include support for managing smart card resources. When this is the case, implementations of OpenCard Framework will make appropriate use of them.

OpenCard and the API Specification

The *OpenCard Framework* defines a set of simple, high-level, and flexible APIs.

The APIs insulate the application programmer from the particularities of the information exchange with smart cards, thus simplifying the developer's task. They are considered high-level because card-specific details are handled outside the applications, making them portable across card and card-operating system vendors. They are considered flexible because they leave sufficient room to accommodate non-standard functionality or future extensions brought about by technological advances. For instance, they allow the addition of cryptographic functions (be they proprietary or eventually standardized, as the ones currently being defined under the auspices of ISO).

The *OpenCard Framework* is programming language-neutral.

Even though the API specifications are object-oriented and geared to Java, they can be transcribed to other object-oriented languages, such as C++, and to procedural languages. Network computing unfolds in a world in which the Java Application Environment (consisting of the Java Virtual Machine and run time environment as well as the Java class libraries) prevails over the other application environments.

OpenCard and Interoperability

The *OpenCard Framework* aims at interoperability across devices, card operating

systems, and applications.

The issue of smart card interoperability can be divided into three distinct levels:

Physical Interoperability

Physical interoperability is the ability to establish communication with a smart card, using electronic and mechanical specifications. Interoperability is achieved by designing devices to meet a required specification on demand or to follow the complete set of specifications.

Syntactical Interoperability

Syntactical interoperability is the ability to create the same functionality using different smart cards with different protocols and data structures. A high-level abstraction of the data objects and access mechanisms is necessary. Interoperability is achieved by using a common interface and by handling card-specific details outside of the application. Physical interoperability is a prerequisite for syntactical interoperability.

Semantical Interoperability

Semantical interoperability is the ability of different applications to share information stored in different formats. A common interpretation of the data objects as well as common rules for their manipulation and use are required. Interoperability can be achieved by regulating the presentation and interchange of information. Syntactical interoperability is a prerequisite for semantical interoperability.

OpenCard Framework addresses physical interoperability implicitly by expressing compliance with ISO 7816-in particular, parts 1-3. It is explicitly concerned with syntactical interoperability by defining high-level abstractions of data objects and access mechanisms and defining a common interface that allows applications to be kept free of card-specific details. Finally, OpenCard Framework will release guidelines for semantical interoperability in the areas of certificate and public-key representations and handling, making use of existing open standards, such as X.509.

OpenCard Framework: Architecture

According to the objectives and requirements of the OpenCard Framework, the architecture of the OpenCard Framework consists of four main components, namely **CardTerminal**, **CardAgent**, **CardIO**, and the **CardAgentExtension** components-as shown in Figure 1.

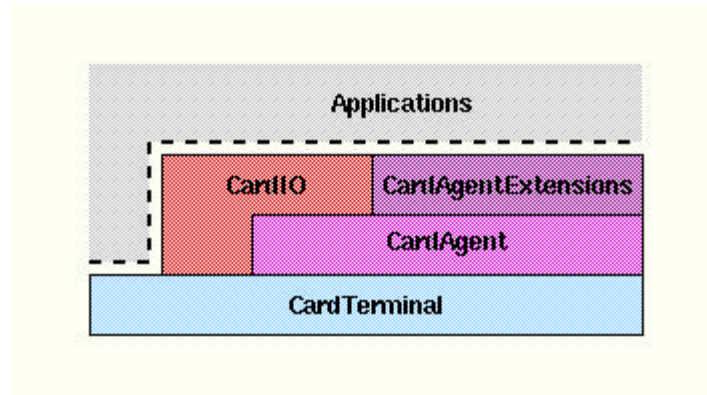


Figure 1. The CardTerminal, the CardAgent, the CardIO, and the CardAgentExtension components all make up the architecture of the OpenCard Framework. Applications interact with the CardIO component, the CardAgentExtension component, and the CardTerminal component.

The CardTerminal component encapsulates all card terminal-related classes; access to a card terminal can only take place through classes of the CardTerminal component.

The CardAgent component provides the necessary infrastructure to interact with a multitude of card operating systems (COS). The CardAgent is the only component not directly accessible to applications.

The CardIO component provides access to the file system functionality of a smart card; for example, an application can open, read, and update an existing elementary file on a smart card, or it can create new files. In addition, the CardIO component offers support for locating and selecting specific application-related file sets (application selection). Non-file-related smart card functionality is in the domain of the CardAgentExtension component: CardAgent extensions implement and offer interfaces to specific smart card functionalities—for example, to cryptographic functions (such as PKCS#11 functionality) or to other application-specific commands (ASCs).

In the following sections, we will describe the four components in more detail.

CardTerminal

The classes of the CardTerminal component serve a dual purpose: The most prominent function is to provide access to physical card terminals and their specific features (such as a card slot, a display, a PIN pad, etc.). This functionality is encapsulated in the CardTerminal class, the Slot class, and the CardID class.

The CardTerminal class is an abstract super class from which concrete implementations for particular card terminal types derive. Each CardTerminal owns one or more Slot objects that represent the physical card slots of that terminal. In addition, a CardTerminal offers access to input/output devices, such as displays, PIN pads, and so on (as Figure 2 shows).

In the context of the CardTerminal component, a physical smart card is represented by a CardID object. A CardID object is, in essence, just an encapsulation of the Answer-To-Reset (ATR) response of a card operating system. Note that a CardID object need not be fully specified; it is possible to instantiate a partially specified CardID object, a wild card CardID object, that represents a set of CardIDs.

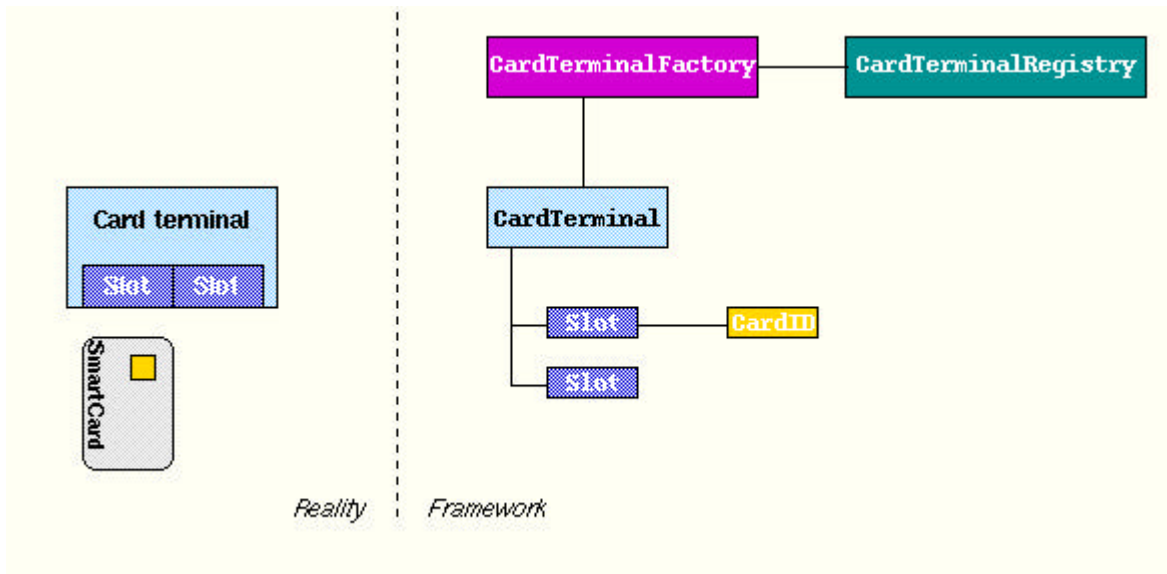


Figure 2. The representation of a physical card terminal in the OpenCard Framework is the `CardTerminal` class; likewise, the `Slot` class is the representation of the card slots of the physical card terminal. Within the `CardTerminal` component, the `CardID` class represents a physical smart card. Note that, for simplicity's sake, only one `CardTerminal` and one `CardTerminalFactory` are shown here; of course, there could be multiple instances of `CardTerminalFactory` classes as well as of `CardTerminal` classes.

The second function of the `CardTerminal` component is a mechanism for dynamically adding and removing card terminals. The `CardTerminalFactory` class and the `CardTerminalRegistry` implement this function. A card terminal manufacturer who wants to support the OpenCard Framework provides a `CardTerminalFactory` that "knows" about a family of card terminals; this manufacturer also provides the respective `CardTerminal` classes. The `CardTerminalRegistry` keeps track of the installed card terminals and offers methods to register and de-register `CardTerminal` objects and to enumerate all installed card terminals (again, see Figure 2).

`CardTerminal`, `Slot`, and `CardTerminalRegistry` are SmartCard object providers. All offer the same subset of methods for waiting for a card insertion:

- Insertion of just any smart card
- Insertion of a particular smart card specified by a `CardID` object
- Insertion of a particular smart card providing certain functionality specified by a `CardAgentExtension` class
- Insertion of a particular smart card carrying specific application data.

Furthermore, the `Slot` class provides methods for waiting for card removal.

CardAgent

The main part of the `CardAgent` component is the abstract `CardAgent` class. A particular derived `CardAgent` encapsulates the know-how about a specific card operating system; in

addition, it serves as a pivot for all classes that in any way deal with the card operating system (for example, the classes of the CardIO component). The CardAgent carries out the communication with the card operating system and keeps track of the card state. Additionally, it can handle authentication and message segmentation.

A CardAgent essentially provides just enough functionality to support the CardIO component. Other functionality, such as cryptographic functions and application specific commands, is taken care of by CardAgentExtensions. Also, note that the CardAgent itself might possibly draw on the services of other classes or even other frameworks (for example, a cryptographic framework).

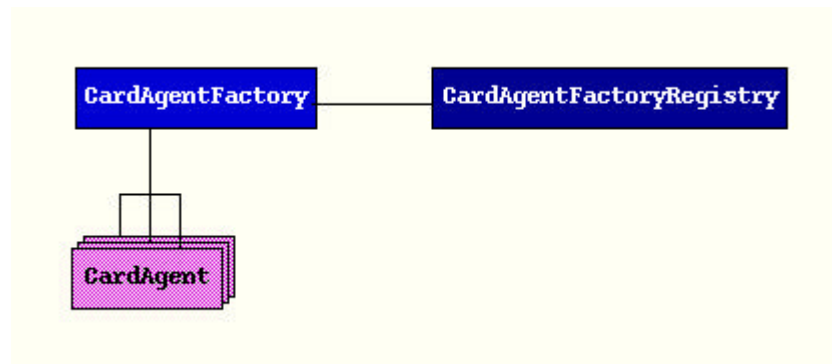


Figure 3. A CardAgent encapsulates the knowledge about a particular card operating system. The CardAgentFactoryRegistry knows about the CardAgentFactory classes that are available in a given system and also knows about which CardAgentFactory can produce a CardAgent for a given CardID.

As Figure 3 illustrates, the CardAgent component employs a CardAgentFactory/CardAgentFactoryRegistry scheme similar to the mechanism of the CardTerminal component. Any card operating system provider who wants to support the OpenCard Framework must provide a CardAgentFactory that knows the CardIDs of the card operating systems that it supports and that can instantiate an appropriate CardAgent. Each CardAgentFactory registers itself with the CardAgentRegistry and indicates the set of CardIDs that it supports. In combination with wild card CardIDs, that is, CardIDs that specify a set of related card operating systems, this scheme allows for operations such as dynamic retrieval of CardAgents from a remote CardAgent server and their subsequent installation and use on the local host.

For a particular CardID object, the CardAgentFactoryRegistry will return the set of CardAgentFactory objects that can supply a suitable CardAgent.

CardIO

The CardIO component represents the largest part of the application-visible interface; all application interaction with a smart card takes place through at least one of the CardIO component classes. Figure 4 shows the classes of the CardIO component.

The base class is the SmartCard class. All other CardIO component classes rely on it. The SmartCard class represents a physical smart card to the application programmer. Through a SmartCard object, the application can access the Slot object (and, thus, the associated CardTerminal) that represents the physical slot holding the smart card.

As Figure 4 shows, access to the file system contained on a smart card takes place by

mounting the root, the master file (MF), of the smart card file system; the result of the mount process is a CardFile representing the master file. Using the MF CardFile, an application can then access other files on the smart card by instantiating appropriate CardFile objects.

The CardRandomAccessFile class provides basic access to the contents of smart card files. Depending on the input/output paradigm of the surrounding run time environment, other input/output classes might be available (for example, the Java reference implementation provides for CardFileInputStream and CardFileOutputStream classes in addition to the CardRandomAccessFile class).

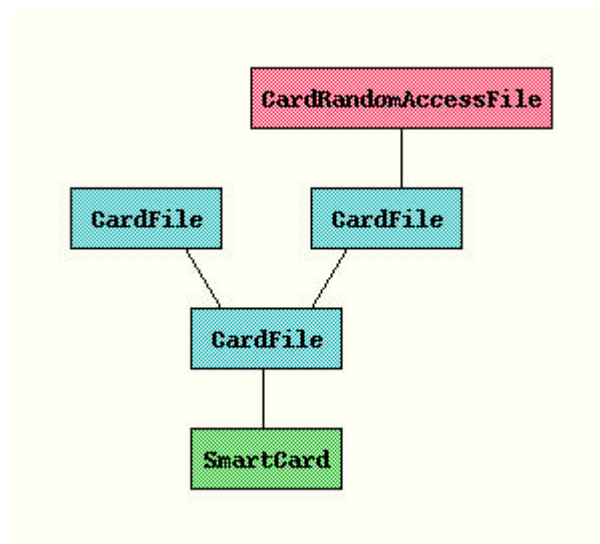


Figure 4. The basic CardIO class is the SmartCard class: Each input/output interaction with the physical smart card takes place through a SmartCard object. Files on the smart card are represented by CardFile objects. In order to access the content of a smart card file, an application must instantiate a CardRandomAccessFile.

Regardless of how an application uses the classes of the CardIO component, each access to the smart card is subject to security restrictions imposed by the smart card itself and the organization of its card file system, the card layout. Thus, in most cases, an application must provide appropriate CardAuthenticator objects containing references to authentication information (such as cryptographic keys) that authenticate it to the smart card (via the SmartCard and associated CardAgent objects).

Note that the CardIO component deals with accessing only files and objects contained on a smart card. Cryptographic functions cannot be accessed through the classes of the CardIO component; instead, access to such smart card functionality is in the domain of the CardAgentExtensions.

CardAgentExtensions

Modern smart cards usually offer considerably more functionality than only a simple file system. The additional functionality covers, for example, application-specific commands for supporting electronic purse functions and/or cryptographic functions for public key cryptography. Although for some domains (industry) standards are emerging (for example, PKCS#11 for cryptographic functions), more often than not, each card operating system supplier provides its own version of, for example, cryptographic functions, ASC for electronic purse implementation, or initialization and personalization functions. To

accommodate this wide range of functionality, the OpenCard Framework provides room for CardAgentExtensions.

As Figure 5 shows, a CardAgentExtension offers a specific API to the application programmer (for example, in Figure 5, the PKCS#11 API and an electronic purse API). Each CardAgentExtension communicates with a CardAgent to provide the offered functionality; thus, generally, a CardAgentExtension must be able to create and decode the APDUs that it exchanges with a CardAgent.

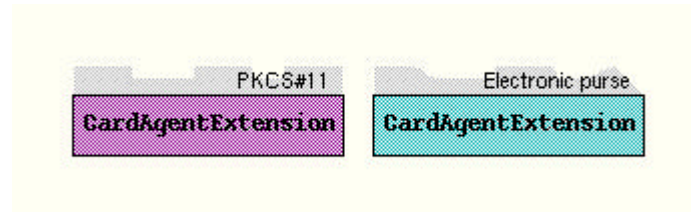


Figure 5. CardAgentExtensions make non-file system-related functionality, card-operating specific functionality, and application-specific command functionality available to the application. Shown here are two different CardAgentExtensions: One implements a PKCS#11 interface, the other an electronic purse API.

Usually, a card-operating system supplier will provide a bundle of CardAgent and CardAgentExtensions.

At a Glance

In summary, the OpenCard Framework is an open architecture that provides for

- a variety of card terminals through the CardTerminalFactory/CardTerminalRegistry mechanism
- different card-operating systems by incorporating the CardAgentFactory/CardAgentFactoryRegistry concept
- a wide range of smart card functionality through the CardAgentExtension mechanism.

Figure 6 shows an overview of the architecture as discussed above.

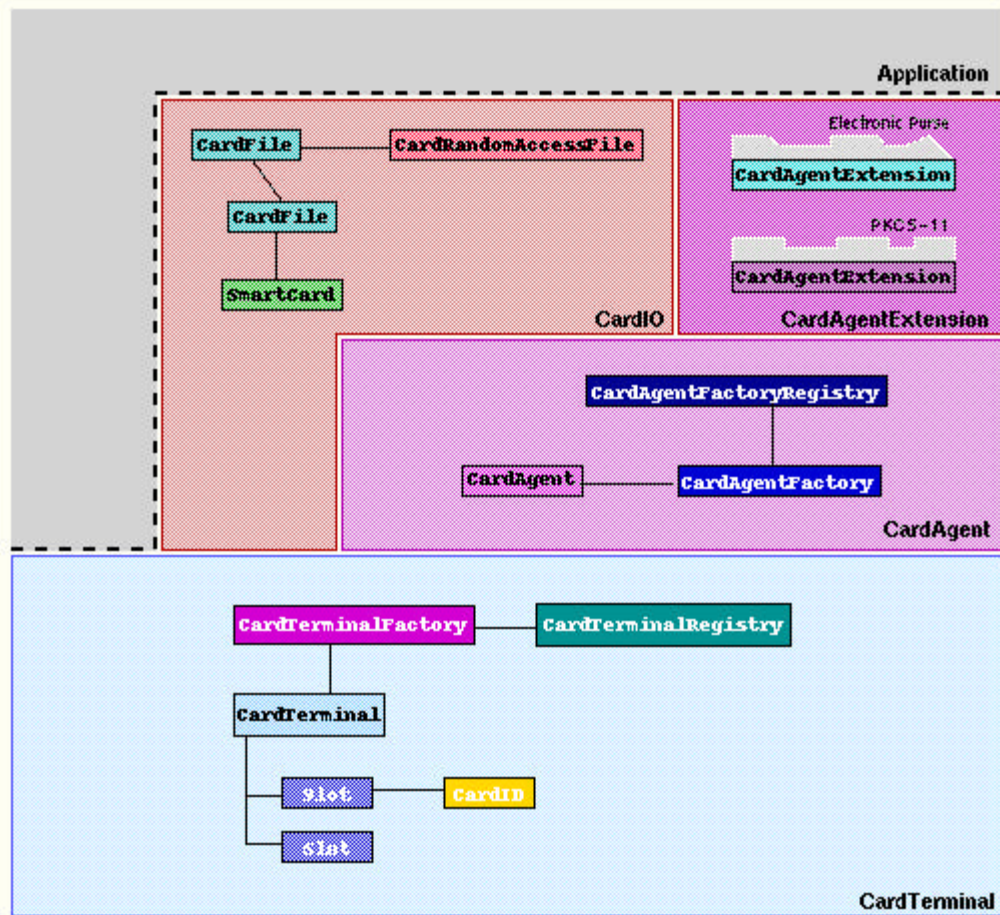


Figure 6. The OpenCard Framework is an open architecture accommodating different card-operating systems (through the CardAgentFactory and CardAgentFactoryRegistry mechanism), different card terminals (via the CardTerminalFactory and CardTerminalRegistry mechanism), and different smart card functionality (through the CardAgentExtension concept).

OpenCard Framework: Java Reference Implementation

Please note that the accompanying javadoc generated documentation of the Java classes and interfaces is still being finalized and is subject to change.

The reference implementation of the OpenCard Framework is written almost entirely in Java and closely follows the architecture description. The Java implementation consists of three Java packages:

opencard.terminal

The `opencard.terminal` package comprises the classes of the architecture's CardTerminal component.

opencard.agent

The opencard.agent package contains both the classes of the CardAgent component and the classes of the CardAgentExtension component - the CardAgentExtension component depends heavily enough on the CardAgent component to justify their collocation in one Java package.

opencard.io

The opencard.io package comprises the CardIO component classes.

Where possible and appropriate, the classes of the architecture description are augmented with additional Java classes that either facilitate programming with the OpenCard Framework or provide a seamless integration into the existing Java programming paradigm.

Since most of the classes discussed in the architecture section of the OpenCard Framework have their Java counterparts, the following sections will only discuss the Java-specific issues. The reader is referred to the respective javadoc generated documentation for further details.

The opencard.terminal Package

The classes of the CardTerminal component exist in the opencard.terminal package as normal Java classes with the exception of CardTerminal and CardTerminalFactory. CardTerminal is an abstract class that needs to be extended for a specific card terminal; CardTerminalFactory is an interface.

Additionally, the opencard.terminal package contains a CardProvider interface, implemented by Slot, CardTerminal, and CardTerminalRegistry, that provides methods for waiting for card insertion.

The opencard.agent Package

Of the CardAgent and CardAgentExtension component classes, only CardAgentFactory and CardAgentExtension are Java interfaces rather than classes.

Additional classes in the opencard.agent package are Channel, representing the ISO 7816-4 logical channel concept, and PDU (for communication between a CardAgentExtension and a CardAgent).

The opencard.io Package

In addition to the Java counterparts of the CardIO component classes (all realized as normal Java classes), the opencard.io package contains CardFileInputStream and CardFileOutputStream. The Java interface CardFilenameFilter is the counterpart of the java.io.FilenameFilter.

Example

The following example does not represent runnable code. Its purpose is to give an impression of the "look & feel" of the Java implementation of the OpenCard Framework.

Neither is the example exhaustive; only a subset of the functionality of the OpenCard Framework is demonstrated.

```
package opencard.demos;

import java.io.DataInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import opencard.agent.CardAgentException;
import opencard.io.CardFile;
import opencard.io.CardFileInputStream;
import opencard.io.SmartCard;
import opencardterminal.CardRequest;
import opencardterminal.CardTerminalException;
import opencardterminal.CardTerminalRegistry;

public class OpenCardExample {

    public static void main(String argv[]) {
        SmartCard card = null;
        CardFile file = null;
        DataInputStream dis = null;

        // Wait for insertion of smart card supporting the
        // "OpenCardExample" application; use an unlimited timeout.
        try {
            card = CardTerminalRegistry.cardTerminalRegistry().waitForCard(new CardRequest("OpenCardExample"));
        } catch (CardTerminalException e) {
            System.err.println("Oops, that card request didn't go through");
            e.printStackTrace();
        }

        try {
            // Access the root of the smart card file system, wait until card becomes
            // available.
            file = card.mount(CardFile.BLOCKING);

            // This example assumes that the "OpenCardExample" application data is
            // located directly at the master file.
            file = new CardFile(file, "OpenCardExample");

            // Now instantiate a CardFileInputStream
            dis = new DataInputStream(new CardFileInputStream(file));

            // Read in the owner's name
            String owner = dis.readUTF();

            // Explicitly close the InputStream to yield the smart card to other
            // applications
            dis.close();

            // Had we protected the "OpenCardExample" application data with a
            // password we would be sure that we indeed are talking to the card's
            // owner; now we cannot be sure...
            System.out.println("Hi there, you are " + owner + "?");
        } catch (CardAgentException e) {
            System.out.println("Hmm, some other application seems to be using the card " +
                "exclusively. Giving up, sorry.");
            System.exit(3);
        } catch (FileNotFoundException e) {
            System.out.println("Oops, where did my application data go?");
            System.exit(4);
        } catch (CardTerminalException e) {
            System.out.println("How can I retrieve your name from your smart card " +
                "if you all of a sudden remove it from the card terminal?");
            System.exit(5);
        } catch (IOException e) {
            System.out.println("Houston, we have a problem with communications ...");
            System.exit(6);
        }
    }
}
```

OpenCard Framework: Summary

Benefits

The OpenCard Framework offers significant benefits to application and service developers as well as to card and terminal providers.

Application and service developers benefit from the OpenCard Framework as follows:

- Vendor independence-Developers can choose cards and terminals from different suppliers.
- Asset protection-Extendability of the architecture enables developers to participate in future developments of smart card technology at low cost by migrating at the level of the API.
- Improved time-to-market-Developers profit from shorter development cycles by programming against a high-level API.
- Lower development cost-Developers save the extra cost for porting their applications to different platforms, and they benefit from lower skill requirements for accomplishing a given task.

Card and terminal providers benefit from OpenCard Framework in these ways:

- Increased clientèle-Providers gain access to new market segments reaching many more customers.
- Improved competition-Providers can compete in terms of functionality and are less vulnerable to the predominance of a single vendor.
- Less development effort-Providers inherit functionality provided by the framework, which reduces their development efforts.

Perspective

The OpenCard Framework architecture and reference implementation provide both the blueprint and the shell for what will eventually become the home of the mobile user in the future information economy, an environment in which network computers, smart cards, service providers, and application developers all play cooperatively together. The OpenCard Framework will enable mobile users to roam freely with their smart cards. After inserting an OpenCard-compliant smart card into a network computer, the card holder is authorized to access network-based information, databases, and services such as electronic mail, Internet applications, and electronic commerce.

The OpenCard Framework APIs become the standard interfaces for smart card-aware applications. For the world of Java this means that card-aware applications or applets will run on any network computer that is OpenCard-compliant.

The OpenCard Framework standard provider interfaces will foster the development of smart card building blocks by numerous vendors.

The OpenCard Framework reference implementation lowers the risks and investments of all parties involved by providing them a functionally rich structure to plug in.

Outlook

OpenCard will publish a comprehensive architecture document giving more details about the OpenCard Framework.

OpenCard will closely follow the development of new open standards relevant to smart cards and will supplement or adapt the guidelines of the OpenCard Framework. Similarly, as new technological advances emerge, OpenCard will incorporate them into its architecture.

OpenCard will establish a clearing process through which third parties may submit their standards for approval and inclusion in the OpenCard Framework.

A focal point of future OpenCard activities will be the establishment of new standards that aim at achieving interoperability at the semantic (or application) level. OpenCard will further such interoperability through its clearing process and by developing its own open standards for selected application domains.

OpenCard Framework: Glossary

Answer-To-Reset (ATR)

Data that is returned by the smart card when the smart card is physically reset.

Application-Specific Command (ASC)

An extension of the basic smart card operating system, often stored in the smart card EEPROM.

Application Protocol Data Unit (APDU)

Communication with a card operating system is message-oriented. Within the context of ISO 7816-4, a message is called an application protocol data unit.

API

Application Programming Interface.

ASC

See Application-Specific Command.

CardAgent

Abstraction that encapsulates the knowledge of how to access data on a smart card. This knowledge depends on the operating system of the smart card and on the capabilities of the card terminal device.

Card-Aware Application

The part of an application running on the computer to which is connected a card terminal hosting the card with the card-resident application.

Card Operating System (COS)

The micro-code contained in the smart card ROM that is used for communicating with the smart card, managing security, and managing data in the smart card files.

Card-Resident Application

The part of an application situated on the smart card that resides in the card terminal connected to the computer running the card-aware application.

COS

See Card Operating System.

Crypto Token

Device that holds cryptographic information and/or performs cryptographic operations.

Dedicated File (DF)

A dedicated file is part of a smart card's file system and can contain elementary files or other dedicated files. See also Elementary File, Master File.

Elementary File (EF)

An elementary file is part of the smart card file system that contains application data. See also Dedicated File, Master File.

ICC

See Integrated Circuit(s) Cards.

Initialization

The process of writing initialization data to the smart card EEPROM. The initialization data is the same for smart cards sharing a common smart card layout.

Integrated Circuit(s) Cards

Terminology used in the International Standards Organization (ISO) for smart cards.

Master File (MF)

The master file of a smart card's file system is a special dedicated file that represents the root of the file system. An MF can contain dedicated files and elementary files. See also Elementary File, Dedicated File.

NC

See Network Computer.

NC Reference Profile

A set of guidelines promoted by Apple, IBM, Netscape, Oracle, and Sun to provide a common denominator of popular and widely used features and functions across a broad range of scalable network computing devices, including personal computers.

Network Computer

Network computing device compliant with the NC Reference Profile.

Personalization

The process of writing card holder-specific data to the smart card.

Smart Card Layout

The smart card layout refers to the organization of dedicated and elementary files in the smart card's EEPROM.

OpenCard Framework: References

CEN 726

European Committee for Standardization. EN 726 Requirements for IC cards and terminals for telecommunications use. 1995.

EMV

Europay International S.A., MasterCard International, Inc., Visa International Service Association. Integrated Circuit Card Specification for Payment Systems: Card Specification, Terminal Specification, Application Specification. June 30, 1996.

IBM Open SmartCard Architecture

IBM Germany Smart Card Solutions. The IBM Open Smart Card Architecture. 3 February 1997.

ISO 7816

International Organization for Standardization. Identification Cards: Integrated Circuit(s) Cards with Contacts, Parts 1-7.

Network Computer Reference Profile

Apple, International Business Machines Corporation, Inc., Oracle Corp., Sun Microsystems, Inc., and Netscape. Network Computer. May 20, 1996.

PC/SC

CP8 Transac, Hewlett-Packard Company, Microsoft Corporation, Schlumberger SA,

Siemens Nixdorf Informationssysteme AG. Interoperability Specification for ICCs and Personal Computer Systems. Draft. 28 January 1997.

PKCS

RSA Laboratories. Public-Key Cryptography Standards. June 1991, cited 11 March 1997.

X.509

International Telecommunications Union. Recommendation X.509 (11/93) - Information technology - Open Systems Interconnection - The directory: Authentication framework. November 93.

Java is a trademark of Sun Microsystems, Inc.

Other companies, products, and service names may be trademarks or service marks of others.

[Copyright](#) [Trademark](#)

[Java Feature Page](#) [Java Home](#)

