PROYECTO FIN DE CARRERA

IMPLEMENTACIÓN DE TARJETAS INTELIGENTES CON TECNOLOGÍA JAVA CARD

Tutor del Proyecto Fin de Carrera
ANTONIO JESÚS SIERRA COLLADO

Alumno
MANUEL VALENZUELA ROMERO

Índice

Índice de ilustraciones

Índice de tablas

1 OBJETIVO

La tecnología Java Card está orientada a la realización de aplicaciones en dispositivos con altas restricciones en cuanto a cantidad de memoria, capacidad de proceso e incluso con carencia de teclado y de display. Por ello, esta tecnología es adecuada para el desarrollo de aplicaciones en tarjetas inteligentes. Sin embargo, estas aplicaciones están limitadas por la capacidad de la tarjeta inteligente y por las características propias de la tecnología Java Card.

Este proyecto fin de carrera tiene como objetivo, la implementación de aplicaciones con la tecnología Java Card en las tarjetas inteligentes usadas en dispositivos móviles (SIM y USIM).

Con este proyecto fin de carrera, también se pretende describir las características (hardware y software) de las tarjetas inteligentes (como la SIM ó la USIM), las tendencias en cuanto a las aplicaciones en las que se usan las tarjetas inteligentes y la utilización la tecnología Java Card para realizar aplicaciones, en targetas inteligentes, que interaccionen con un ordenador o con el teléfono móvil.

Como resultado del objetivo fijado, se presenta una aplicación desarrollada específicamente para tarjetas inteligentes (tarjetas SIM) de dispositivos móviles, usados en redes de telefonía móvil GSM. Esta aplicación, que se encuentra en la tarjeta SIM, es capaz de presentar una serie de menús (en la pantalla del terminal móvil) que permiten al usuario, acceder a las distintas funcionalidades que puede realizar. La aplicación realizada presenta opciones en pantalla, que se pueden encontrar en las aplicaciones que las operadoras de telefonía móvil instalan en sus tarjetas SIM. En concreto, esta aplicación incluye varios menús, una eurocalculadora totalmente operativa y una funcionalidad que permite mandar un correo electrónico mediante un SMS, sin que el usuario se percate de ello.

2 INTRODUCCIÓN

En la actualidad, se hace un uso intensivo de las tarjetas magnéticas e inteligentes (también llamadas smart cards), propiciado por la bajada del precio de las mismas. Los campos de aplicación en los que intervienen las tarjetas son diversos: telefonía, operaciones bancarias (en cajeros), pago de pequeñas cantidades de dinero (en peajes de autopista, pago de billetes de autobús, ...) e identificación (para el próximo año 2004 se espera la implantación de los nuevos DNI's electrónicos, que también nos identificarán en el mundo digital). Por estos ejemplos expuestos se demuestra la importancia que están adquiriendo las tarjetas.

Este proyecto fin de carrera tiene el objetivo de llevar a cabo la implementación de aplicaciones en tarjetas empleadas en las comunicaciones móviles (SIM y USIM), usando la tecnología Java Card.

Para empezar, se explicarán las características de las tarjetas inteligentes (entre las que se encuentran las SIM y USIM) y porqué son preferibles a sus hermanas que poseen la banda magnética. Luego se detallarán las ventajas de la tecnología Java Card.

Las tarjetas inteligentes se componen de un sustrato de plástico, que suele tener el tamaño de una tarjeta de crédito y en él se aloja un circuito de silicio. Mediante unos contactos metálicos situados en una de las caras de la tarjeta, el circuito de silicio puede comunicarse con el exterior, obtener una señal de reloj y la alimentación necesaria.

A diferencia de las tarjetas magnéticas, el circuito de silicio dota a la tarjeta de cierta inteligencia o capacidad de proceso. En este circuito hay memoria ROM, memoria EEPROM, memoria RAM y una pequeña CPU (que quizás incorpore un coprocesador matemático, para agilizar las operaciones criptográficas que requiera la aplicación). En definitiva, las smart cards son pequeños ordenadores de bolsillo. En la memoria ROM se almacena el sistema operativo y algunas aplicaciones que vengan de fábrica. En la memoria EEPROM (con una cantidad del orden de kbytes) se suelen guardar aplicaciones que se añaden después del proceso de fabricación y datos que interesen preservar (como el saldo de un monedero electrónico). Gracias a la memoria RAM y la CPU, se pueden ejecutar las aplicaciones que se almacenan en memoria ROM y EEPROM. Los tamaños de las tarjetas, los contactos, las características eléctricas y los protocolos de comunicación con la tarjeta, están normalizados por la ISO.

Gracias a estas características hardware y software de las smart cards, las ventajas de estas tarjetas frente a las tarjetas magnéticas son claras. La diferencia más importante es la seguridad que se puede obtener con las smart cards, ya que las tarjetas magnéticas se pueden leer y copiar fácilmente. En cambio, el circuito de silicio no se puede copiar (se evita el problema de las tarjetas magnéticas). Para obtener un dato guardado en la smart card, quizás se tenga que dialogar con una aplicación que se encuentre en la tarjeta y que quizás pida la introducción de un número secreto ó PIN, que solo conoce el propietario de la tarjeta.

Cada fabricante suele utilizar tecnologías propietarias en los procesos de fabricación, en el diseño de los chips de las tarjetas inteligentes y en los sistemas operativos de dichas tarjetas. Para programar aplicaciones en las tarjetas, el

programador usaba el lenguaje ensamblador y debía tener en cuenta las peculiaridades del hardware y el sistema operativo de la tarjeta (programación a muy bajo nivel), que como se ha comentado anteriormente, dependían del fabricante de tarjetas. Además, también debían tener en cuenta restricciones como la poca cantidad de memoria de la que disponen las tarjetas inteligentes y la baja capacidad de computación de las mismas. Entonces, el programador se especializaba en la programación de las tarjetas de un determinado fabricante.

Además, las aplicaciones generadas para un determinado tipo de tarjetas, podían no funcionar para otras, es decir, el código fuente de la aplicación no era portable. Por lo tanto, la adaptación de la aplicación a otro tipo de tarjeta conllevaba unos costes temporales y en recursos humanos.

Descrito lo que son las tarjetas inteligentes y la problemática planteada a la hora de desarrollar las aplicaciones, resultará más fácil comprender que es lo que aporta la tecnología Java Card. La tecnología Java Card ofrece varias ventajas:

- La tecnología Java Card está diseñada para ejecutar aplicaciones en dispositivos con altas restricciones de memoria (decenas de kbytes) y de capacidad de proceso. Esto hace que la tecnología Java Card sea la adecuada para las tarjetas inteligentes.
- El desarrollador de aplicaciones programa usando el lenguaje de programación Java con un esquema orientado a objetos.
- Las aplicaciones contenidas en la tarjeta, se ejecutan en una máquina virtual de Java reducida (debido a que la cantidad de memoria y recursos de la tarjeta inteligente, son muy reducidos).
- La existencia de la máquina virtual de Java, esconde al programador la complejidad del hardware subyacente (esta tecnología ofrece API's bien definidas, para acceder por ejemplo, a los recursos hardware o al sistema de ficheros) y del sistema operativo sobre el que se monta la máquina virtual.
- Portabilidad del código entre tarjetas Java Card de distinto fabricante.
- Ofrece la posibilidad de instalar aplicaciones en la tarjeta después de la fabricación de la misma.

En estas ventajas radica el interés de esta tecnología, sobre la que versa este proyecto fin de carrera. Gracias a estas ventajas, el desarrollo de aplicaciones es más fácil y el coste asociado a dicho desarrollo es menor porque el código de la aplicación solo se escribe una vez y porque la tecnología Java Card esconde al programador la complejidad del hardware subyacente.

En la siguiente sección se explicará el alcance de este proyecto fin de carrera, que no solo se limita al estudio de la tecnología Java Card.

2.1 ALCANCE

Este proyecto fin de carrera comienza en primer lugar por una descripción de las aplicaciones más comunes en las que se usan las tarjetas inteligentes, las dificultades de la programación de las smart cards y los beneficios que aporta la tecnología Java Card.

Después se explica de forma detallada qué es una smart card (o tarjeta inteligente). Con la explicación se pretende mostrar la forma física de las tarjetas, la posición y forma de los contactos, los protocolos que se usan para la comunicación con la tarjeta, la configuración de la tarjeta en cuanto al tipo de memoria y CPU que se suelen incluir en las tarjetas, y los estándares que las definen.

En segundo lugar se detalla la arquitectura usada en la tecnología Java Card, sus API's y la forma de programar una aplicación. En un capítulo posterior se verá como utilizar la criptografía para conseguir que las aplicaciones sean más seguras.

En tercer lugar, se presenta a la tarjeta SIM usada en las redes de telecomunicaciones móviles GSM y a la tarjeta USIM (que se usará en las redes de telecomunicaciones móviles UMTS). En esta sección se verán las peculiaridades de las tarjetas SIM y USIM: características físicas y eléctricas de la tarjeta, el sistema de ficheros (solo de dará una explicación de algunos ficheros o directorios destacados), los comandos que usa el terminal móvil para comunicarse con la tarjeta SIM y USIM y la arquitectura de Java Card en este tipo de tarjetas. Esta sección no incluye una descripción a bajo nivel de los comandos, ni una descripción exhaustiva de las estructuras de datos intercambiadas entre el móvil y la SIM, ni del algoritmo GSM (procedimientos de autenticación en la red GSM, gestión de claves, ...).

Vista la tarjeta SIM y USIM, se hace un estudio del SIM Application Toolkit. El SIM Application Toolkit permite que una tarjeta SIM ó USIM pueda enviar órdenes al terminal móvil (como mostrar menús por la pantalla, emitir un tono, enviar SMS's, iniciar una llamada, ...). El SIM Application Toolkit se encuentra presente en las tarjetas SIM phase 2+ (estandarizado finalmente en 1999). En este estudio, se detallan los comandos que puede emplear la SIM, pero no a bajo nivel. Para más información acerca de estos comandos, se puede consultar la especificación del ETSI relacionada con estos comandos.

Enlazando con la sección anterior, se presenta una aplicación realizada que utiliza el SIM Application Toolkit. Esta aplicación realizada podría ser la que incluyese una operadora de telefonía móvil en sus tarjetas SIM. La aplicación muestra un menú, en el que se podrá elegir entre una eurocalculadora y una utilidad para enviar correo electrónico. La aplicación de correo se basa en el envío de un SMS de forma transparente al usuario. En cuanto a esta utilidad de envío de correo, no se detalla la implementación del centro de conmutación de mensajes para tratar esos mensajes y enviar los correos electrónicos.

Por último, se incluye una guía de cómo instalar el simulador necesario para simular el terminal móvil y probar que el código de la aplicación realizada funciona correctamente, y los planos del código fuente de dicha aplicación. También se incluye otra guía de instalación de herramientas estándar necesarias para desarrollar una aplicación Java Card, con ejemplos para familiarizarse con dichas herramientas y con la estructura que suelen tener las aplicaciones Java Card. Hay que señalar que el coste de estas herramientas es nulo, ya que se ha optado por la elección de software libre o de evaluación.

Para finalizar, en los apéndices del proyecto, se encuentra la descripción de todas las clases (y sus métodos) de la tecnología Java Card y la SIM API.

3 LAS SMART CARDS

En este capítulo se explican las ventajas que supone la utilización de las tarjetas inteligentes y los campos de aplicación en los que se usan. La dificultad a la que se enfrenta el desarrollador de aplicaciones consiste en que la programación de dichas tarjetas se realiza a muy bajo nivel (lenguaje ensamblador). Con el uso de la tecnología Java Card, se facilita el desarrollo de aplicaciones. Esta tecnología oculta al desarrollador la complejidad del hardware y del sistema operativo de la tarjeta inteligente, y le permite utilizar el lenguaje de programación Java. En este capítulo también se realiza una descripción de las características de las smart cards en cuanto a hardware, software, y protocolo de comunicación (entre la tarjeta y un lector de tarjetas).

3.1 EL PORQUÉ DE LAS SMART CARD Y JAVA CARD

Las smart cards (ó tarjetas inteligentes, ver Figura 1) guardan y procesan información gracias a los circuitos electrónicos hechos de silicio y que se encuentran incluidos en un soporte de plástico. Una smart card es una especie de ordenador portátil. A diferencia de las tarjetas que poseen bandas magnéticas, las smart cards poseen poder de computación y capacidad para almacenar información.

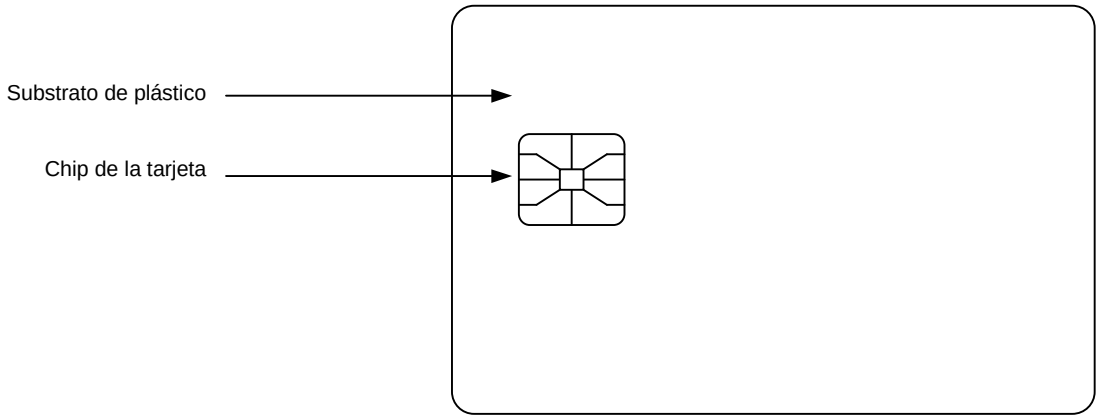


Figura 1: Smart card (tarjeta inteligente)

3.1.1 HISTORIA DE LAS SMART CARDS

La idea de incorporar un circuito integrado en una tarjeta de plástico, fue de dos inventores alemanes, Jürgen Dethloff y Helmut Grötrupp, en 1968. Más tarde registraron la patente de su invención. Independientemente, Kunitaka Arimura del Arimura Technology Institute de Japón, registró una patente de smart card en 1970. Aunque los verdaderos progresos vinieron con las 47 patentes relacionadas con smart cards de Roland Moreno, registradas en 11 países entre 1974 y 1979. Al final de la década de los 70, CII-Honeywell-Bull (ahora grupo Bull) fue el primero en comercializar la tecnología smart card y de introducir las smart cards con microprocesador.

Los ensayos iniciales con smart cards tuvieron lugar en Francia y Alemania al principio de la década de los 80, como tarjetas telefónicas de prepago y como tarjetas

bancarias. Estos ensayos satisfactorios demostraron el potencial de las smart cards contra la falsificación y su flexibilidad.

Recientemente, con los avances en las tecnologías de fabricación de chips y en la criptografía, las smart cards son más potentes. Ahora se usan para guardar dinero electrónico (reemplazando el dinero de papel), para guardar de forma segura los historiales médicos, para prevenir los accesos no autorizados a transmisiones de programas de TV por cable o por satélite, y para mejorar la seguridad en la telefonía sin cables.

Ya son muy habituales en Europa y Asia debido al uso extendido de aplicaciones como GSM y las tarjetas bancarias. La entrada de las smart cards llegó a ser significativa en los EEUU (a finales de la década de los 90) gracias al crecimiento de la demanda de la seguridad en el campo del comercio electrónico.

3.1.2 VENTAJAS

El interés en las smart cards se debe a las ventajas que aportan. Una ventaja es su potencia computacional. La seguridad, la portabilidad, y la facilidad de uso, son otras de las ventajas clave que aportan.

El procesador, la memoria y el soporte de entrada y salida, están empaquetados en un único circuito incrustado en una tarjeta de plástico. Una smart card es resistente a los ataques porque no necesita depender de recursos externos vulnerables a ataques. Para conseguir la información que contiene una smart card hace falta la posesión física de la tarjeta, conocimientos del hardware y del software de la tarjeta y equipamiento adicional.

La seguridad se hace mayor con el uso de funciones de criptografía. Los datos guardados en la tarjeta se pueden codificar para salvaguardar la privacidad en la memoria física, y los datos intercambiados entre la tarjeta y el mundo exterior se pueden firmar y codificar. Además, el acceso a una smart card suele requerir la introducción de un PIN (número de identificación personal), que evita su uso por parte de personas no autorizadas.

Otra ventaja de las smart cards es su portabilidad. Se puede llevar una smart card en la cartera de la misma forma en que se llevan las tarjetas bancarias.

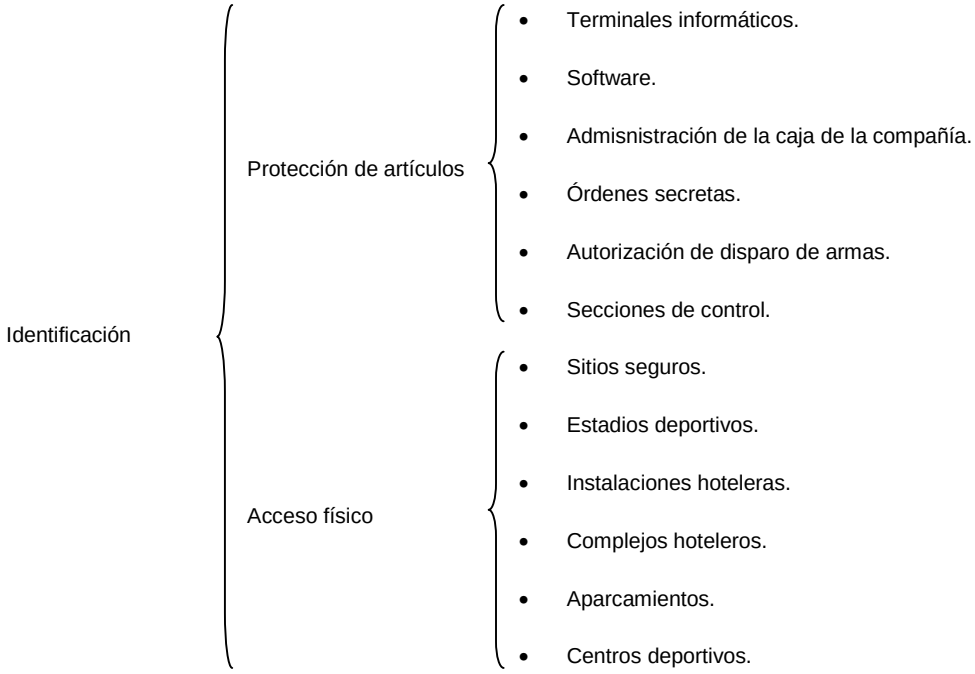
La smart cards son también muy prácticas. Para comenzar una transacción, se inserta la tarjeta en un dispositivo, y se retira del dispositivo cuando el trabajo ya se ha realizado.

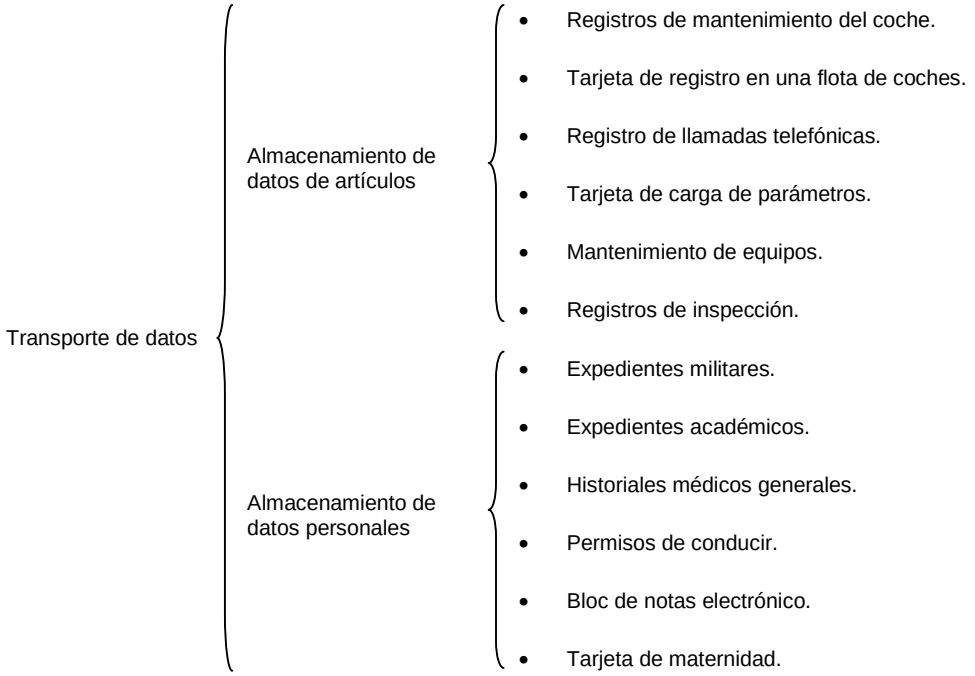
3.1.3 APLICACIONES DE LA TECNOLOGÍA JAVA CARD

Las smart cards se están introduciendo en un número cada vez mayor de aplicaciones de servicios. Están consiguiendo ocupar el lugar del dinero, los tickets, los ficheros e incluso ciertos documentos. Algunos ejemplos de estas aplicaciones son las transacciones bancarias (usando la tarjeta que proporciona el banco), la tarjetas telefónicas de prepago, el pago del peaje de las autopistas, el control de acceso, el archivado de los historiales médicos y la televisión de pago. Todas estas aplicaciones necesitan cada vez más inteligencia en las smart cards (que no ofrece las tarjetas de

bandas magnéticas) para asegurar el creciente nivel de seguridad que exigen estas aplicaciones.

Las aplicaciones prácticas de una smart card se pueden clasificar de forma general en tres categorías principales:





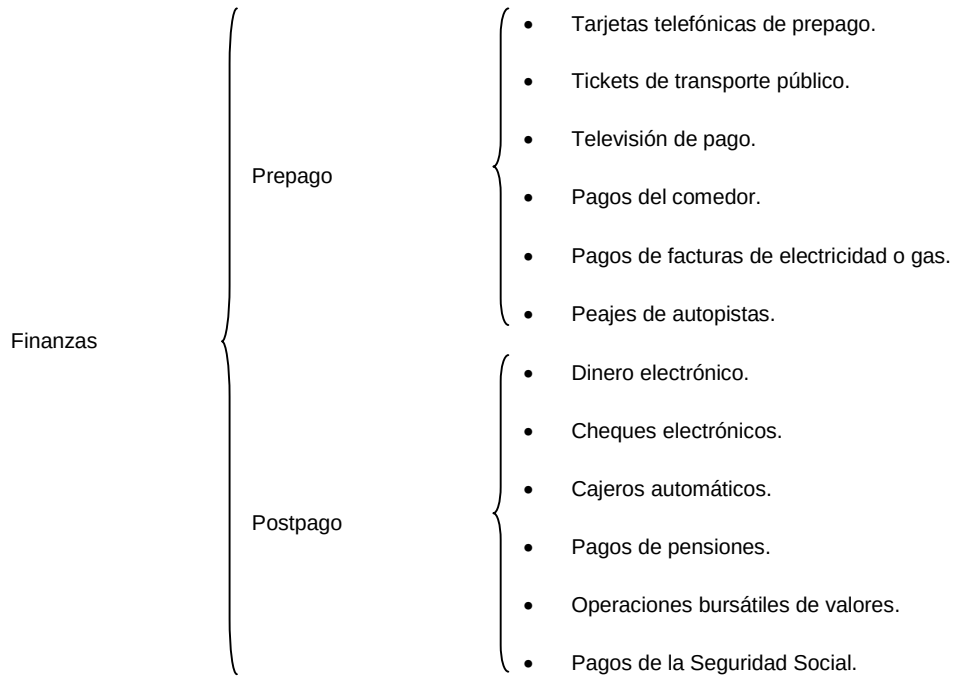


Figura 2: Clasificación de las aplicaciones de una smart card

- Identificación: Las smart cards proveen unas medidas de seguridad para identificar al titular de la smart card con el fin de permitirle o no el acceso (por ejemplo, la autorización para utilizar un PC).
- Transporte de datos: Las smart cards se usan como un dispositivo de almacenamiento de información práctico, portátil y seguro.
- Finanzas: Las tarjetas se pueden usar en transacciones, reemplazando por ejemplo a los cheques.

Algunas de estas aplicaciones se detallan a continuación.

3.1.3.1 Transportes

Con millones de transacciones que ocurren cada día relacionadas con el transporte, las tarjetas han encontrado fácilmente un lugar en este mercado que crece rápidamente. A continuación se presentan algunos ejemplos de uso de las smart cards en el mundo del transporte.

3.1.3.1.1 Transporte público

El uso de tarjetas sin contactos, permite a los pasajeros ir en varios autobuses y trenes durante sus viajes diarios al trabajo. Todo esto sin tener que preocuparse de los precios de los billetes o de llevar cambio. En Londres, los autobuses están usando smart cards sin contactos para recaudar el precio del billete. Cada vez que los pasajeros entran en un autobús, pasan sus tarjetas enfrente de un lector que descuenta el precio del billete del crédito que se guarda en la tarjeta. En Londres se planea extender esta forma de pago al metro y a los parkings.

3.1.3.1.2 Cobro electrónico del peaje

En Singapur, el ERP (Electronic Road Pricing) fue diseñado para automatizar el sistema de pago de peajes basado en funcionarios que realizan el cobro en una caseta.

El sistema ERP consiste en tres sistemas principales: una unidad que se encuentra en el vehículo IU (In-vehicle unit), la caseta y el sistema central de computación. El IU es un dispositivo electrónico que se encuentra instalado en el vehículo que acepta un valor guardado en una CashCard (smart card). El IU descuenta de la CashCard un ERP apropiado cada vez que el vehículo pasa a través de una caseta ERP. Las matrículas de los vehículos que realicen una entrada ilegal se fotografían gracias a las cámaras instaladas en las casetas, con lo que al propietario se le puede aplicar las medidas que se estimen. Un vehículo realiza una entrada ilegal cuando no dispone de una IU instalada, cuando no tenga una CashCard o cuando no disponga de suficiente crédito en la CashCard.

3.1.3.2 Comunicaciones

3.1.3.2.1 Tarjetas telefónicas de prepago

Los teléfonos públicos que utilizan monedas son: caros de construir porque deben ser muy robustos para que el dinero no se pueda robar, caros de mantener porque hay una necesidad de recoger el dinero y de poca fiabilidad ya que el mecanismo del teléfono se puede llenar de dinero, atascar y estropear debido al vandalismo. Aunque ya se han usado algunas formas de tarjetas magnéticas y ópticas para solucionar este problema, la mayoría de las operadoras telefónicas se están decantando por las smart cards como la solución más efectiva, por las siguientes razones:

- Con una smart card, la sobrecarga en la línea se puede minimizar ya que el teléfono de smart card puede operar sin línea durante una llamada e intercambiar información de control con un host cada cierto periodo de tiempo. Además, la carga que implica el mantenimiento de la llamada también se puede mantener al mínimo.
- Debido a su potencia computacional, las smart cards abren la posibilidad del uso de la tarjeta más allá de las fronteras. Cada operadora puede hacer disponible su clave pública a las otras, de este modo permite que las operadoras puedan autentificar todas las tarjetas.

Actualmente, cerca de 80 países de todo el mundo utilizan las smart cards en los teléfonos públicos. Se pueden encontrar los mismos beneficios en muchas otras aplicaciones donde las máquinas que funcionan con monedas están sufriendo un incremento de costes y pérdidas del servicio. Todo esto debido al vandalismo, la recogida de las monedas y la poca fiabilidad de la maquinaria. Las smart cards también se están usando en otras áreas donde tradicionalmente se usaban las monedas:

- Peajes de autopista.
- Parkings.
- Gasolineras.

- Máquinas de vending.
- Juegos de arcade.

3.1.3.2.2 Seguridad en la telefonía móvil

Hoy en día, la industria de las telecomunicaciones inalámbricas representa el mayor mercado que usa las smart cards para conseguir seguridad. El ejemplo más notable es el estándar GSM (Global System for Mobile Communication). Este estándar desarrollado por la ETSI permite que cada operador nacional mantenga el control de la seguridad y los aspectos de facturación, y al mismo tiempo facilita el uso de los teléfonos móviles en otros países donde funcione el sistema GSM. El estándar GSM y sus subconjuntos (PCN, PCS) se usan en más de 90 países de todo el mundo, lo que da idea del gran éxito que ha adquirido este estándar.

El GSM usa una smart card que guarda toda la información personal del abonado. La smart card, también llamada SIM (módulo de identificación de abonado), se puede insertar en un slot de cualquier teléfono móvil GSM para realizar las operaciones oportunas (como llamadas o envío de mensajes cortos). Las llamadas al número móvil del abonado se dirigen al teléfono donde se encuentra insertada la smart card y las facturas se cargan en la cuenta personal del abonado. Debido a que la identidad del usuario está programada en la tarjeta SIM, el usuario puede utilizar cualquier teléfono que acepte este tipo de tarjetas.

La tarjeta SIM identifica al usuario en la red GSM y provee claves de codificación para las transmisiones digitales de voz. La clave generada por la tarjeta SIM para codificar es temporal y se cambia con cada uso. Por ello, si se consiguiera descodificar una transmisión, esto sería inútil para la próxima transmisión. La tarjeta SIM incorpora un código secreto, conocido como PIN, para proteger al abonado de la malversación y la estafa.

Gracias a que las comunicaciones inalámbricas están adquiriendo cada vez mayor aceptación, el papel de los teléfonos móviles va más lejos de las transmisiones de voz. Las operadoras compiten por ofrecer servicios de valor añadido tales como el comercio electrónico.

3.1.3.2.3 Ejemplos de aplicaciones realizadas con el SIM Application Toolkit

El SIM Application Toolkit es un mecanismo, definido en el estándar GSM, que permite a las aplicaciones, que se encuentran en la tarjeta SIM, hacer uso del terminal móvil para desplegar menús en la pantalla y capturar las teclas pulsadas por el usuario. Mediante los menús y la pulsación de las teclas del terminal móvil, el usuario puede acceder a las distintas funcionalidades que la aplicación posee. Además, la aplicación podrá enviar y recibir SMS's (mensajes cortos) para comunicarse con un servidor, si la funcionalidad, elegida por el usuario, lo requiere.

A continuación se exponen dos ejemplos que hacen uso del mecanismo SIM Application Toolkit:

3.1.3.2.3.1 EcoMobile

El Ecomobile es un proyecto desarrollado por la Universidad de Murcia. Este proyecto ofrece un servicio para la comunidad universitaria que permite compartir vehículos con otras personas que no poseen medio de transporte. El servicio consiste en poner en contacto a las personas que poseen vehículos con las que no tienen, de una forma sencilla, rápida y segura a través de sus teléfonos móviles, teniendo en cuenta la localización espacial y temporal de los desplazamientos.

EcoMobile es una aplicación cliente-servidor, en la que el cliente se ejecuta en el ME y el servidor se comunicará con el ME gracias a una pasarela GSM-SMS. La comunicación entre ambos se realiza mediante SMS. SMS sobre GSM proporciona el canal de comunicación que la aplicación requiere, reproduciendo el esquema de comunicación de un protocolo no orientado a conexión.

El funcionamiento es el siguiente: cuando se desea utilizar el servicio ya sea para ofertar una plaza o para solicitar una plaza en el vehículo, el usuario selecciona la aplicación EcoMobile en el menú correspondiente de su móvil. Este applet le solicita al usuario que elija la localidad de origen y destino, dándole al usuario listas preconfiguradas basadas en su localización actual y preferencias. Además el usuario selecciona la fecha y hora de salida y de llegada. Tras la introducción de estos datos se registra en el servidor la solicitud. Todos estos pasos se pueden ver en la siguiente figura:

Figura 3: Aplicación Ecomobile en la pantalla de un móvil

Cuando el servidor recibe las demandas de servicio, intenta encontrar aquellas ofertas que mejor se adecuan a los requisitos (localización espacial y temporal) indicados en dichas demandas. En el momento en el que el servidor determina que una o varias ofertas pueden satisfacer una determinada demanda, procede a clasificarlas por orden de importancia. Dichas ofertas se envían y se muestran en el móvil del demandante del servicio a través del OTA. En cada una de ellas existirá una referencia (forma de contacto) al ofertante del servicio, lo cual permitirá que los dos interesados se puedan poner en contacto y determinar su interés real y el lugar y la hora concretos donde van quedar.

En el momento en el que el demandante del servicio (o el que lo ofrece) satisface su interés, puede anular su demanda (u oferta) haciendo uso del applet de gestión de demandas (o de ofertas) y el código interno que se almacenó cuando se solicitó el servicio (o se hizo la oferta).

3.1.3.2.3.2 Pago de los recibos de la electricidad

En Suecia, Telia (que es la operadora de telecomunicaciones número uno en ese país) y el Postgirot Bank (una filial del Sweden Post con 1,7 millones de abonados y que es el intermediario de pago más destacado en el mercado sueco de pagos) lanzó su aplicación MobilSmart en septiembre de 1997. Esta aplicación hace uso del de SIM Application Toolkit y ayuda al abonado a pagar las cuentas de la electricidad en cualquier parte con el terminal móvil.

El funcionamiento de la aplicación es el siguiente: se pide al usuario que teclee un código PIN específico en el teléfono GSM para empezar la aplicación. A continuación, el usuario introduce el número de cuenta del Postgirot Bank, la fecha y la cantidad a pagar. Entonces, la tarjeta SIM empaqueta esta información en un SMS.

El usuario podrá añadir un comentario para poder identificar el pago cuando se vaya a revisar el estado de la cuenta en un futuro. Además La tarjeta calcula automáticamente una firma digital mediante un algoritmo especializado para ello.

Este SMS se envía al SMSC (Short Message Service Center) de Telia y se procesa en el sistema SmartSec, perteneciente al Postgirot Bank. Una vez que ha sido procesado, SmartSec le devuelve al usuario un acuse de recibo, para confirmar el pago.

3.1.3.3 Empresas públicas

Las compañías públicas de electricidad en Reino Unido, Francia y otros países, están usando las smart cards para realizar el prepago y reemplazar las lecturas de los contadores. Los clientes compran electricidad en centros autorizados de pago y se les suministra una tarjeta que sirve para recargar el contador.

Los clientes también pueden utilizar la tarjeta para acceder a información acerca de su cuenta, como la cantidad que queda, la cantidad consumida ayer o el último mes, y la cantidad del último crédito. Se define un umbral de emergencia para permitir a los clientes usar la electricidad y pagar más tarde. Una vez que el umbral de emergencia se consume, se corta la electricidad.

3.1.3.4 Seguridad en Internet

En Internet, la autenticación de usuarios y el control de accesos es un motivo importante para elegir a las smart cards. Hay un creciente uso de las smart cards en la infraestructura de las claves públicas. Una smart card lleva la clave privada del titular y un certificado digital (dos componentes que verifica la identidad del titular en el mundo electrónico). Las aplicaciones están usando las smart cards para controlar el acceso a los sitios Web, para firmar digitalmente los correos electrónicos (E-mails), y en transacciones seguras on-line.

3.1.3.5 Seguridad en ordenadores

3.1.3.5.1 Boot Integrity Token System (BITS)

El BITS fue desarrollado para proteger los sistemas de ordenadores de un largo número de virus que afectan al sistema de arranque, y aplicar el control de acceso. BITS está diseñado para que el ordenador arranque desde un sector de arranque guardado en la smart card, prescindiendo del sector de arranque del ordenador que puede ser fácilmente infectado por un virus. La tarjeta también se puede configurar para permitir el acceso al ordenador solo a usuarios autorizados.

3.1.3.5.2 Autentificación en Kerberos

En un entorno de computación distribuido abierto, no se puede confiar en una estación de trabajo para identificar a sus usuarios, debido a que la estación de trabajo puede que no se encuentre en un entorno bien controlado y quizás bastante lejos del servidor central. Un usuario puede ser un intruso que quizás intente atacar al sistema o pretenda ser alguien para extraer información del sistema a la que su usuario no tiene acceso. Para proteger el sistema de ataques de hosts remotos conectados a la red, se debe tener en cuenta cierta clase de autenticación.

Kerberos es una de los sistemas que provee servicios de autenticación de confianza para autenticar los usuarios de un entorno de red distribuido. Cuando un usuario o cliente solicita un acceso a un servicio particular del servidor, obtiene un ticket o una credencial del servidor AS (Authentication Server) de Kerberos. Entonces el usuario presenta esa credencial al servidor TGS (Ticket Granting Server) y obtiene un ticket de servicio. De aquí en adelante, el usuario puede solicitar el servicio presentando el ticket de servicio al servidor deseado.

Con este protocolo se asegura que el servidor ofrece servicios al cliente correcto que está autorizado para tener acceso. Esto es porque Kerberos asume que solo el usuario correcto puede usar la credencial debido a que los otros no tienen la password para decodificarla. Y también debido a esto, un usuario puede solicitar realmente la credencial de otros. Es decir, el usuario no se autentica en la etapa inicial. Un atacante podría obtener la credencial de otro usuario, y llevar a cabo un ataque adivinando la contraseña ya que el ticket solo está protegido por una contraseña.

Para evitar esta falta de seguridad, se propuso integrar la smart card en el sistema Kerberos. La idea principal es mejorar la seguridad de la autenticación de Kerberos autenticado directamente al usuario en el comienzo y antes de conceder el ticket inicial, para que un usuario no pueda tener el ticket de otro.

3.1.3.6 Cuidados médicos

Las smart cards también pueden llevar información médica. Esta información puede consistir en detalles de la cobertura del seguro médico, reacciones adversas a ciertos medicamentos, historiales médicos, nombre y números de teléfono de los doctores y otra información vital en una emergencia.

Alemania ha empezado a suministrar tarjetas a todos los ciudadanos para que lleven su información del seguro médico básico. En Francia y Japón, los pacientes con enfermedades renales tienen acceso a las tarjetas que contienen sus historiales de diálisis y el tratamiento. La tarjeta fue diseñada con características de seguridad para controlar el acceso a la información a personal y doctores autorizados.

En los Estados Unidos, la ciudad de Oklahoma tiene un sistema de tarjetas inteligentes llamado MediCard. Diseñada por profesionales de la salud, esta smart card es capaz de controlar el acceso de forma selectiva a los historiales médicos del paciente, que se guarda en su MediCard. Si embargo, la información esencial, incluyendo al médico de cabecera, está disponible al personal de emergencia en casos extremos. Los lectores de smart cards están instalados en hospitales, farmacias, ambulancias, consultas de los médicos y hasta en el departamento de bomberos, permitiendo que la MediCard se pueda usar en circunstancias rutinarias y urgentes.

3.1.3.7 Identificación personal

Varios países como España y Corea del Sur han empezado a probar smart cards que identifican a los ciudadanos de estos países. En España, la tarjeta de identificación de la Seguridad Social es una smart card. La identificación se verifica mediante las huellas dactilares guardadas en las tarjetas. En Corea se está probando un proyecto piloto con tarjetas inteligentes que incluyen identificación, el carné de conducir, el seguro médico y prestaciones por jubilación.

3.1.3.8 Finanzas

Otro ejemplo de uso de las smart cards se encuentra en la industria bancaria. Sus funciones son parecidas a las realizadas por las tarjetas de banda magnética. A diferencia de estas últimas, con las smart cards se evita que la información se pueda copiar fácilmente y luego malversarla. Las smart cards previenen el fraude y por lo tanto los costes bancarios debidos a dichos fraudes (que se estiman en billones de dólares en todo el mundo).

Gracias las nuevas tendencias en el área de los micropagos y de los bancos, se están introduciendo los monederos electrónicos. Las tarjetas guardan dinero electrónico, y el balance se puede incrementar o decrementar. Los monederos electrónicos basados en smart cards pueden reducir el coste de manejar dinero en formato de papel; en particular, proveen un mecanismo de pago ideal para las transacciones pequeñas, donde la carga por el uso de tarjetas de crédito es demasiado alta para las transacciones de poco valor.

3.1.3.9 Promociones comerciales

Además, las tarjetas se pueden usar como acumuladoras de puntos en promociones comerciales. La tarjeta guarda puntos de confianza, para patrocinar a los comerciantes, que se van acumulando cuando el titular de la tarjeta compra artículos. El titular de la tarjeta puede usar los puntos a cambio de descuentos u otros regalos. Los datos capturados cuando se usa la tarjeta, se pueden usar también para ayudar a los comerciantes a entender las preferencias de compra y el comportamiento de los clientes.

3.1.4 RETOS EN EL DESARROLLO DE APLICACIONES PARA LAS SMART CARDS

El desarrollo de aplicaciones para smart cards ha sido tradicionalmente un proceso largo y difícil. Aunque las tarjetas están estandarizadas en tamaño, forma, y el protocolo de comunicaciones, los trabajos para desarrollar una aplicación dependen generalmente de un fabricante a otro. La mayoría de los entornos de desarrollo, que están construidos por los fabricantes de smart cards, contienen herramientas (que usan el lenguaje ensamblador) y emuladores de hardware obtenidos a partir de los fabricantes de chips de silicio. Ha sido virtualmente imposible para terceras partes, el desarrollar aplicaciones independientemente del fabricante de smart card y vendérselas a los proveedores de smart cards. Por lo tanto, el desarrollo de smart cards ha estado limitado a un grupo de programadores altamente cualificado y especializado, que tiene un conocimiento profundo del hardware y software de una tarjeta específica.

Debido a que no hay aplicaciones de alto nivel estandarizadas disponibles para smart cards, los desarrolladores de aplicaciones necesitan tratar a bajo nivel con protocolos de comunicaciones, con el gestor de memoria, y con otros pequeños detalles dictados por el hardware específico de la smart card. La mayoría de las aplicaciones de smart card en uso hoy en día, han sido desarrolladas desde abajo, lo que supone un gran consumo en tiempo, y normalmente el llevar un producto al mercado suele tomar un año o dos. La actualización del software o mover las aplicaciones a plataformas diferentes, suele ser difícil o imposible.

Ya que las aplicaciones de smart card están desarrolladas para ejecutarse en plataformas propietarias, las aplicaciones de proveedores diferentes no pueden coexistir y ejecutarse en una sola tarjeta. La falta de interoperatividad impide un fuerte despliegue de aplicaciones de smart card.

3.1.5 JAVA APLICADO A LAS SMART CARDS

La tecnología Java Card ofrece un camino para superar los obstáculos que entorpece la aceptación de las smart cards. Permite que las tarjetas inteligentes y otros dispositivos con restricciones de memoria, ejecuten aplicaciones (llamadas applets) escritas en el lenguaje de programación Java. Esencialmente, la tecnología Java Card define una plataforma segura, portable y multiaplicación, que incorpora muchas de las ventajas del lenguaje Java.

3.1.5.1 Beneficios de la tecnología Java Card

Los desarrolladores de aplicaciones para smart cards pueden beneficiarse de la tecnología Java Card por los siguientes motivos:

- Facilidad de desarrollo de aplicaciones: el lenguaje Java conduce la programación de smart cards en la línea central de evolución del desarrollo de software, evitando que los desarrolladores tengan que programar en lenguaje ensamblador (como 6805 ó 8051). Los desarrolladores también se pueden beneficiar de muchos de los entornos de desarrollo de Java. Estos entornos están diseñados por compañías como Borland, IBM, Microsoft, Sun y Symantec. Además, la tecnología Java Card ofrece una plataforma

abierta que define las interfaces de programación estándares y el entorno de ejecución. La plataforma esconde la complejidad subyacente y los detalles del sistema de la smart card. Los desarrolladores de applets trabajan con interfaces de lenguaje de alto nivel. Así pueden concentrar sus esfuerzos en los detalles de la aplicación y beneficiarse de las extensiones y librerías que otros han creado.

- Seguridad: la seguridad es uno de los asuntos más importantes cuando se trabaja con las tarjetas inteligentes. El sistema de seguridad de Java, se ajusta bien al entorno de las smart cards. Por ejemplo, el nivel de acceso a todos los métodos y variables está estrictamente controlado y no hay manera de que los punteros habiliten programas maliciosos que puedan fisgonear dentro de la memoria. Además los applets en la plataforma Java Card están separados por el applet firewall. Así el sistema puede protegerse contra aplicaciones hostiles que puedan dañar otras partes del sistema.

- Independencia del hardware: la tecnología Java Card es independiente del tipo de hardware usado. Puede funcionar en cualquier procesador de smart card (8 bits, 16 bits ó 32 bits). Los applets de Java Card están escritos por encima de la plataforma Java Card y por ello son independientes del hardware de las smart cards.

- Capacidad para guardar y mantener múltiples aplicaciones: una smart card puede contener a múltiples applets, tales como programas de monederos electrónicos, de autenticación y cuidados médicos, todos ellos de diferentes proveedores. Esto es debido a que con el mecanismo de firewall de Java Card, los applets no pueden acceder a otro a menos que se le permita hacerlo.

  Una vez que se adquiere la tarjeta, se pueden descargar más applets a la tarjeta. La funcionalidad de las tarjetas se puede mejorar continuamente, con applets nuevos o actualizados, sin necesidad de adquirir una tarjeta diferente.

- Compatibilidad con los estándares de smart card existentes: la tecnología Java Card está basada en el estándar internacional de smart card ISO 7816. Por ello puede soportar fácilmente sistemas de smart card y aplicaciones que son generalmente compatibles con ISO 7816. Los applets pueden interoperar no solo con todas las smart card de Java, sino también con los dispositivos lectores de tarjetas existentes.

3.2 CONCEPTOS BÁSICOS SOBRE LAS SMART CARDS

3.2.1 DESCRIPCIÓN DE LAS SMART CARDS

A menudo, las smart cards son llamadas chip cards, o integrated circuit (IC) cards. El circuito integrado se incorpora en un sustrato de plástico del tamaño de una tarjeta de crédito. Este circuito integrado contiene elementos que se usan para la transmisión de datos, almacenamiento y procesado. Quizás la smart card tenga una línea magnética en una de las caras y en la otra incorpore el chip. La siguiente figura ilustra la apariencia física de una smart card:

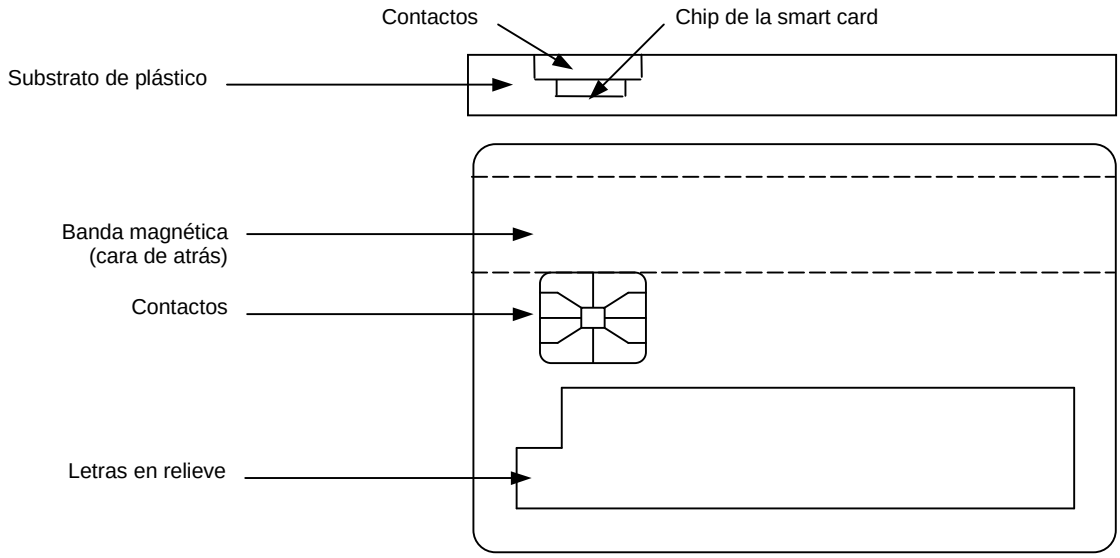


Figura 4: Apariencia física de una smart card

La apariencia física y las propiedades de una smart card se encuentran definidas en la ISO 7816, parte 1. La ISO 7816 es el documento que establece el estándar para la industria de smart cards.

Normalmente una smart card no contiene una fuente de alimentación, o un display ni un teclado. Para comunicarse con el mundo exterior, la tarjeta se debe colocar dentro o cerca de un dispositivo que acepte estas tarjetas (CAD = card acceptance device) y que está conectado a un ordenador.

3.2.2 TIPOS BÁSICOS DE TARJETAS

Las tarjetas se pueden clasificar en varios grupos. Se pueden dividir en tarjetas con memoria y tarjetas con microprocesador. Las tarjetas también se pueden caracterizar en tarjetas de contacto y tarjetas sin contacto, según el mecanismo de acceso a la tarjeta.

3.2.2.1 Tarjetas con memoria vs tarjetas con microprocesador

Las primeras smart cards producidas en grandes cantidades fueron las tarjetas con memoria. Las tarjetas con memoria no son inteligentes porque no contienen un microprocesador. Estas tarjetas incluyen un chip de memoria o un chip de memoria con lógica no programable.

Normalmente, las tarjetas con memoria pueden contener entre 1kbytes y 4kbytes de datos. Se suelen usar como tarjetas de prepago para teléfonos públicos u otros bienes o servicios.

Como las tarjetas con memoria no tienen una CPU para procesar datos, el procesado de datos se lleva a cabo mediante un circuito simple capaz de ejecutar unas pocas instrucciones preprogramadas. Tal circuito tiene funciones limitadas y no puede

ser reprogramado. Por lo tanto, las tarjetas con memoria no se pueden reutilizar. Cuando se consume el valor de la tarjeta, la tarjeta se tira.

Dependiendo de las necesidades de seguridad de los datos guardados, el acceso a los datos se puede proteger mediante medidas que consisten en memoria protegida o en lógica de seguridad. Por ejemplo, las tarjetas telefónicas de prepago pueden contener cierta lógica para evitar que el valor sea incrementado. Sin embargo las tarjetas con memoria se pueden falsificar fácilmente.

La ventaja de las tarjetas con memoria consiste en su tecnología, que es muy simple. Por lo tanto, son preferidas en aplicaciones donde su bajo coste es importante.

Por otro lado, están las tarjetas con microprocesador, que como el nombre indica, contienen un procesador. Ofrecen un grado de seguridad mayor y capacidad de soportar múltiples funciones. Con una tarjeta con microprocesador, los datos nunca están directamente accesibles a las aplicaciones externas. El microprocesador controla el tratamiento de los datos, el acceso a la memoria de acuerdo a un conjunto de condiciones (contraseñas, codificaciones, ...) y las instrucciones dadas por las aplicaciones externas. Muchos modelos actuales de microprocesador para tarjetas, presentan soporte criptográfico. Tales tarjetas son particularmente útiles para aplicaciones que necesitan conseguir seguridad en los datos.

Las tarjetas con microprocesador son muy flexibles en cuanto a funcionalidad. Se pueden optimizar para una aplicación o pueden integrar varias aplicaciones diferentes. Su funcionalidad está restringida solamente por la memoria disponible y por la potencia de cálculo.

Las tarjetas con microprocesador son ampliamente usadas para controlar el acceso a instalaciones, aplicaciones bancarias, telecomunicaciones inalámbricas y en todas aquellas aplicaciones en las que la seguridad y la privacidad de los datos son la mayor prioridad.

Como resultado de la producción masiva, el coste de las tarjetas con microprocesador ha caído drásticamente desde 1990. El precio de las tarjetas con microprocesador depende principalmente de los recursos de memoria y de las funcionalidades software que se incluyen en la tarjeta.

En general, el término smart card se refiere tanto a tarjetas con memoria como a tarjetas con microprocesador. Sin embargo, algunas publicaciones prefieren llamar smart card a las tarjetas con microprocesador, debido a la inteligencia provista por el procesador. El término chip cards se usa para designar a las tarjetas con memoria y a las tarjetas con microprocesador.

Debido a que se requiere un procesador programable de propósito general para soportar el entorno de ejecución de Java Card, en este documento el término smart card hace referencia a las tarjetas con microprocesador.

3.2.2.2 Tarjetas de contactos vs tarjetas sin contactos

Las tarjetas de contactos deben ser insertadas en un dispositivo de aceptación de tarjetas (CAD = card acceptance device). Estas tarjetas se comunican con el exterior

mediante el uso de una interfaz de comunicación serie consistente en ocho puntos de contacto, como se muestra en la Figura 4.

Debido a que las tarjetas de contactos necesitan ser insertadas en un dispositivo mecánico de aceptación de tarjetas y de forma correcta, las tarjetas sin contactos son populares en situaciones donde se requieren transacciones rápidas. Los sistemas de transporte público y los controles de acceso a los edificios, son aplicaciones excelentes para las tarjetas sin contactos.

Las tarjetas sin contactos no necesitan ser colocadas en un dispositivo de aceptación de tarjetas. Se comunican con el mundo exterior a través de una antena enroscada en la tarjeta. La energía se puede suministrar a través de una batería o la puede acumular la antena. Las tarjetas sin contactos transmiten los datos a un dispositivo de aceptación de tarjetas a través de campos electromagnéticos.

Debido a que los microcircuitos de las tarjetas sin contactos están completamente precintados, las tarjetas sin contactos superan las limitaciones de las tarjetas de contactos: no hay contactos que se lleguen a estropear por un uso excesivo, las tarjetas no necesitan ser insertadas cuidadosamente en un CAD, y las tarjetas no tienen que tener un espesor estándar para caber en el CAD.

Sin embargo, las tarjetas sin contactos tienen sus inconvenientes. Las tarjetas sin contactos deben estar a una cierta distancia para intercambiar datos con el dispositivo de aceptación de tarjetas. Como la tarjeta puede salirse rápidamente del rango de cobertura, solo se puede transmitir una cantidad limitada de datos. También es posible que las transacciones sean interceptadas sin que el titular de la tarjeta lo sepa. Además, las tarjetas sin contactos son actualmente más caras que las tarjetas de contactos.

3.2.3 HARDWARE DE LAS SMART CARDS

Una smart card tiene puntos de contacto en la superficie del sustrato de plástico, una unidad central de proceso empotrada en el sustrato de plástico y varios tipos de memorias. Algunas smart cards también vienen con coprocesadores para cálculos matemáticos.

3.2.3.1 Puntos de contacto de las smart cards

Una smart card tiene ocho puntos de contacto, la funcionalidad de cada uno se muestra en la Figura 5. Las dimensiones y localización de los contactos se encuentran especificados en la parte 2 de la ISO 7816.

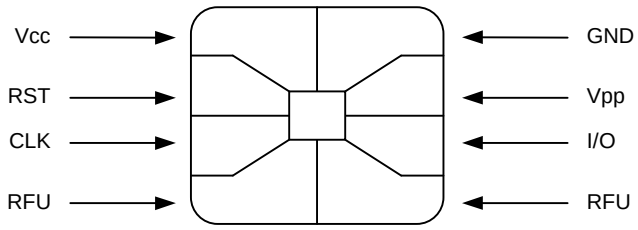


Figura 5: Puntos de contacto de una smart card

- El punto Vcc (3 ó 5 voltios) suministra energía al chip. Vcc suele tener un valor de 3 ó 5 voltios, con una desviación máxima del 10%. Las smart cards de los teléfonos móviles usan una Vcc de 3 voltios.
- El punto RST se usa para enviar una señal de reset al microprocesador. A esto se le llama reset en caliente. Un reset en frío se hace apagando y encendiendo la fuente de alimentación (por ejemplo, extrayendo e insertando la tarjeta del CAD).
- El punto CLK provee un reloj externo al chip. De este reloj externo se deriva el reloj interno (ya que las smart cards no poseen un generador de reloj interno).
- El punto GND se usa como punto de referencia de voltaje. Se considera que su valor es cero.
- El punto Vpp es opcional y se usa solamente en tarjetas antiguas. Cuando se usa, suministra el voltaje de programación, que tiene dos niveles (uno bajo y otro alto). El nivel alto de voltaje se utiliza para programar la EEPROM de algunos chips antiguos de smart cards.
- El punto I/O se usa para transferir datos y comandos entre la smart card y el mundo exterior en modo half-duplex. Esto significa que los comandos o los datos solo se pueden transmitir en una sola dirección en un determinado instante.
- Los puntos RFU están reservados para un uso futuro.

3.2.3.2 Unidad de proceso en una smart card

La unidad central de proceso en la mayoría de las smart cards actuales, es un microcontrolador de 8 bits, que usa el juego de instrucciones del Motorola 6805 o el Intel 8051, con una velocidad de reloj de 5 Mhz. Las tarjetas nuevas tienen microcontroladores de 16 ó 32 bits, con un juego de instrucciones reducido (RISC). En el futuro, las tarjetas de 16 y 32 bits serán las más comunes.

3.2.3.3 Coprocesador en una smart card

Las smart cards que están diseñadas para su uso en aplicaciones de seguridad suelen tener un coprocesador. Un coprocesador criptográfico es un circuito integrado especial que sirve para realizar cálculos, en particular, cálculos de enteros largos. Tales cálculos se requieren para llevar a cabo ciertas operaciones criptográficas, como en el algoritmo RSA. La inclusión de un coprocesador suele afectar al coste del chip.

3.2.3.4 Sistema de memoria de las smart cards

Las smart cards suelen contener tres tipos de memoria: memoria persistente no modificable; persistente y modificable; y no persistente y modificable. ROM, EEPROM y RAM son las memorias más usadas para estas tres clases de memorias.

- ROM (Read-Only Memory): se usa para guardar los programas de la tarjeta. No hace falta energía para mantener los datos en este tipo de memoria. Aunque como dice el nombre, no se puede escribir en ella después del proceso de fabricación. Suele contener rutinas del sistema operativo así como datos permanentes y aplicaciones de usuario. Es la menos cara de las tres clases de memoria.

- EEPROM (Electrical Erasable Programmable Read-Only Memory): al igual que la memoria ROM, puede preservar los datos cuando la fuente de alimentación de la memoria se apaga. La diferencia es que el contenido en este tipo de memoria se puede modificar durante un uso normal de la tarjeta. Se usa para guardar datos (es el equivalente al disco duro en un PC). Las aplicaciones de usuario también se pueden escribir en la EEPROM después de la fabricación de la tarjeta. Los parámetros más importantes de la EEPROM son el número de ciclos de escritura en el tiempo de vida de una tarjeta, el periodo de retención de los datos y el tiempo de acceso a los datos. En la mayoría de las tarjetas, las EEPROM's pueden soportar por lo menos 100.000 ciclos de escritura y pueden retener los datos durante 10 años. La lectura de una EEPROM es tan rápida como la de una memoria RAM, pero la escritura en la EEPROM es 1.000 veces más lenta que la escritura en una RAM. Una celda de memoria EEPROM suele ocupar cuatro veces más espacio que la celda de una memoria ROM.

- RAM (Random Access Memory): se usa como espacio temporal de trabajo para guardar y modificar datos. La RAM no es una memoria persistente, es decir, la información que contiene no se puede preservar cuando la fuente de alimentación se apaga. A la memoria RAM se puede acceder un número ilimitado de veces y ninguna de las restricciones de la EEPROM son aplicables a la RAM. Una celda de RAM suele ocupar cuatro veces el espacio de una celda de memoria de EEPROM.

3.2.4 COMUNICACIÓN CON LA SMART CARD

3.2.4.1 El dispositivo de aceptación de tarjetas y aplicaciones del host.

Una smart card se inserta en el interior de un dispositivo de aceptación (CAD), que puede estar conectado a un ordenador. Los CAD's se pueden clasificar en dos tipos: lectores y terminales.

Un lector se conecta al puerto serie, paralelo, o USB de un ordenador, a través del cual la smart card se comunica. Un lector tiene un slot en el que se inserta la tarjeta o puede recibir los datos a través de campos electromagnéticos procedentes de una tarjeta sin contactos. Además de suministrar potencia a la tarjeta, el lector establece un camino de comunicación de datos en el que la tarjeta puede hablar con el ordenador conectado al lector. Aunque normalmente los lectores no tienen la inteligencia para

procesar datos, muchos de ellos tienen funciones de detección y corrección de errores si los datos transmitidos no cumplen con el protocolo de transporte subyacente.

Por otra parte, los terminales son verdaderos ordenadores. Un terminal integra un lector de smart cards como uno de sus componentes. Los cajeros automáticos de los bancos, son otra forma de terminal. Además de tener la funcionalidad de un lector de tarjetas, un terminal tiene la capacidad de procesar los datos intercambiados con la tarjeta. Por ejemplo, un cajero automático que acepte smart cards, puede añadir o quitar dinero de una aplicación de monedero que se encuentre en la tarjeta.

En este documento, ambos dispositivos se denominan dispositivo de aceptación de tarjetas (CAD). Las aplicaciones que se comunican con la smart card, aunque residan en el ordenador conectado al lector o en el terminal, son llamadas aplicaciones del host. Estas aplicaciones dirigen el proceso de comunicación con la smart card, tal y como se explicará en la siguiente sección.

3.2.4.2 El modelo de comunicación con las smart cards

La comunicación entre la tarjeta y el host es half-duplex, es decir, sólo uno de ellos puede estar transmitiendo datos en un instante de tiempo.

Cuando dos ordenadores se comunican, intercambian paquetes de datos, que están construidos siguiendo un protocolo, como TCP/IP. De forma similar, las smart cards hablan con otros ordenadores usando sus propios paquetes de datos, llamados APDU's (application protocol data units). Una APDU puede contener un mensaje de orden (comando) o un mensaje de respuesta.

El modelo de comunicación (entre host y smart card) empleado es el de maestro y esclavo. La smart card siempre desempeña el papel de esclavo, esperando APDU's de comando procedentes del host (C-APDU). Entonces la tarjeta ejecuta la instrucción especificada en el comando y responde al host con una APDU de respuesta (R-APDU). La tarjeta y el host intercambian APDU's de comando y APDU's de respuesta, de forma alternada.

3.2.4.3 El protocolo APDU

El protocolo APDU, como se especifica en la ISO 7816-4, es un protocolo de nivel de aplicación entre una smart card y una aplicación del host. Los mensajes APDU de la ISO 7816-4 poseen dos estructuras: una usada por la aplicación del host en el lado del CAD para mandar órdenes a la tarjeta y otra estructura usada por la tarjeta para enviar las respuestas a la aplicación del host. Una APDU de comando (C-APDU) va siempre emparejada con una de respuesta (R-APDU). Sus estructuras se ven a continuación:

Tabla 1: Estructura de una APDU de comando

Cabecera obligatoria				Cuerpo opcional		
CLA	INS	P1	P2	Lc	Campo de datos	Le

Tabla 2: Estructura de una APDU de respuesta

Cuerpo opcional	Cuerpo obligatorio	
Campo de datos	SW1	SW2

El significado de los campos de las APDU's de comando es el siguiente:

- CLA: Clase de instrucción. Este byte identifica una categoría de APDU's de comando y respuesta. Las APDU's de tipo comando pueden tener los siguientes valores: 0x0X, 0x8X, 0x9X y 0xAX. Donde la X sirve para determinar canales lógicos y asegurar la codificación del mensaje.
- INS: Código de instrucción. Especifica la instrucción del comando.
- P1: Parámetro de la instrucción.
- P2: Parámetro de la instrucción.
- Lc: Especifica la longitud del campo de datos en bytes.
- Campo de datos: Contiene información adicional para llevar a cabo la instrucción. Es opcional.
- Le: Especifica el número de bytes esperados en la respuesta de la tarjeta.

A continuación se presenta el significado de los campos de las APDU's de respuesta:

- Campo de datos: Es opcional y su longitud coincide con la especificada en el campo Le de la APDU de comando anterior.
- SW1 (status word): Indica el estado de procesado en la tarjeta después de la ejecución del comando (el éxito se marca con el valor 0x9000).
- SW2 (status word): Indica el estado de procesado en la tarjeta después de la ejecución del comando (por ejemplo, un éxito se marca con el valor 0x9000).

Visto el significado de todos los campos, se distinguen 4 posibles casos. La distinción se basa en la existencia o no del campo de datos en la C-APDU o en la R-APDU (Figura 6).

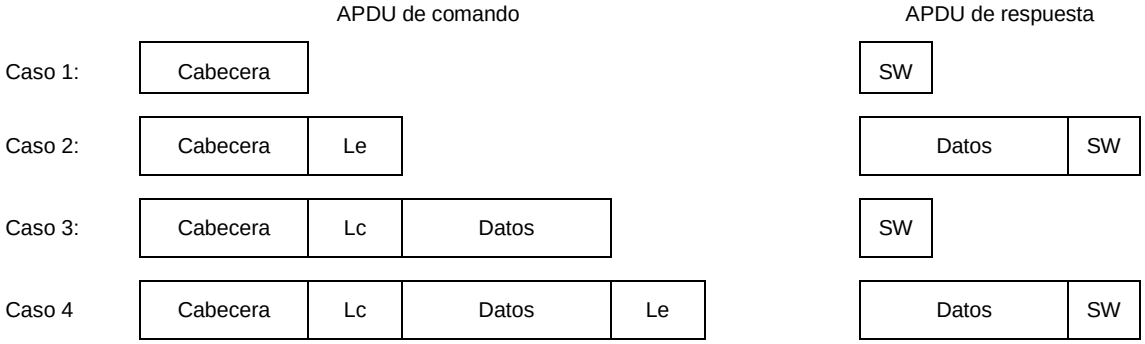


Figura 6: Casos de APDU's de comando y de respuesta

- En el caso 1, no hay datos transferidos desde o hacia la tarjeta, por ello la C-APDU solo contiene la cabecera y la R-APDU solo contiene las palabras de estado (SW).
- En el caso 2, no hay datos transferidos hacia la tarjeta, pero si que hay datos procedentes de la tarjeta. El cuerpo de la C-APDU contiene un solo byte (el campo Le, que especifica el número de bytes de datos de la correspondiente R-APDU).
- En el caso 3, los datos se transfieren a la tarjeta, pero no hay datos procedentes de la tarjeta como resultado del proceso del comando. El cuerpo de la C-APDU incluye el campo Lc y el de datos. El byte Lc especifica la longitud del campo de datos. La R-APDU solo contiene las palabras de estado.
- En el caso 4, los datos son transferidos a la tarjeta, y también hay datos que proceden de la tarjeta como resultado del proceso del comando. El cuerpo de la C-APDU incluye el campo Lc, el campo de datos, y el campo Le. La R-APDU contiene los datos y las palabras de estado.

3.2.4.4 Protocolo TPDU

Las APDU's se transmiten gracias al protocolo de transporte definido en la ISO 7816-3. Las estructuras de datos intercambiadas entre un host y una tarjeta usando el protocolo de transporte, se llaman unidades de datos del protocolo de transporte o TPDU's.

Actualmente, los dos protocolos de transporte que se usan predominantemente en los sistemas de smart cards son el protocolo T=0 y el protocolo T=1. El protocolo T=0 está orientado a bytes ó caracteres, esto significa que la mínima unidad procesada y transmitida por el protocolo es un byte. Este protocolo también es semidúplex, asíncrono y la conversación siempre la inicia el host siguiendo un esquema maestro-esclavo. Por el contrario, el protocolo T=1 está orientado a bloques de datos, es decir, los bloques, que consisten en una secuencia de bytes, son la mínima unidad de datos que se transmiten entre una tarjeta y el host. La comunicación entre la tarjeta y el host es entre iguales, alternándose el permiso para enviar datos.

Las estructuras usadas en los protocolos T=0 y T=1 son diferentes. Toda la información relativa al protocolo de transporte, se puede consultar en la norma ISO 7816-3.

3.2.4.5 ATR

Inmediatamente después del encendido, la smart card envía un mensaje "answer to reset" (ATR) al host. Esto le transfiere al host los parámetros requeridos por la tarjeta para establecer un camino de comunicación de datos. El ATR está compuesto por más de 33 bytes. Contiene parámetros de transmisión, como el protocolo de transporte soportado por la tarjeta (usualmente el T=0 ó el T=1); la tasa de transmisión; parámetros de hardware de la tarjeta (como el número de serie del chip y el número de la versión) y otra información que el host necesita conocer acerca de la tarjeta.

3.2.5 LOS SISTEMAS OPERATIVOS DE LAS SMART CARDS

Los sistemas operativos de las smart cards no tienen nada que ver con los sistemas operativos como UNIX, Windows o DOS. Los sistemas operativos de las smart cards soportan una colección de instrucciones sobre las que las aplicaciones pueden construirse. La ISO 7816-4 estandariza un amplio rango de instrucciones en formato de APDU's. Un sistema operativo de smart card podrá soportar algunas o todas de estas APDU's, además de las extensiones añadidas por los fabricantes.

La mayoría de los sistemas operativos de smart cards soportan un modesto sistema de ficheros basado en el ISO 7816-4. Las APDU's ISO 7816-4 son en su mayoría comandos orientados a sistema, tales como la selección de un fichero o los comandos para acceder al contenido de un fichero. En este caso, una aplicación de usuario es a menudo un fichero de datos que guarda información acerca de una aplicación específica. La semántica y las instrucciones para acceder a los archivos de datos de aplicación son implementadas por el sistema operativo. Sin embargo, la separación entre sistema operativo y las aplicaciones no está muy bien definida.

Aquellos sistemas operativos que poseen un sistema de ficheros centralizado están bien establecidos en las smart cards que están disponibles actualmente. Sin embargo, los nuevos sistemas operativos, que soportan una mejor distinción de los niveles del sistema y la posibilidad de descargar código de aplicaciones personalizadas, están siendo más y más populares. Java Card es una tecnología que va en esta línea.

3.2.5.1 Sistemas de ficheros de las smart card

Las smart cards definidas en ISO 7816-4 pueden tener un sistema de archivos con estructura jerárquica. El sistema de archivos ISO 7816-4 soporta tres tipos de archivos: el archivo maestro o raíz(MF), el archivo dedicado (DF), y el archivo elemental (EF). Cada archivo se especifica por un identificador de 2 bytes o un nombre simbólico de más de 16 bytes llamado FID (File Identifier).

Antes de que se pueda realizar cualquier operación sobre un archivo, primero se debe seleccionar el archivo (esto es el equivalente a abrir un archivo en un sistema operativo moderno). Algunas tarjetas seleccionan automáticamente el archivo maestro cuando se alimenta la tarjeta. El acceso a los archivos se controla mediante condiciones de acceso, que se pueden especificar para los accesos de lectura y de escritura de forma independiente.

3.2.5.2 Archivo maestro

El archivo maestro es la raíz del sistema de archivos. Puede contener archivos dedicados y archivos elementales. Hay un solo MF en una smart card.

3.2.5.3 Archivo dedicado

Un DF es un archivo de directorio que contiene otros DF's o archivos elementales. Un MF es un tipo especial de DF.

3.2.5.4 Archivo elemental

Un archivo elemental es un archivo de datos y no puede contener otros tipos de archivos. Hay cuatro tipos de ficheros elementales, como se muestra en la siguiente figura:

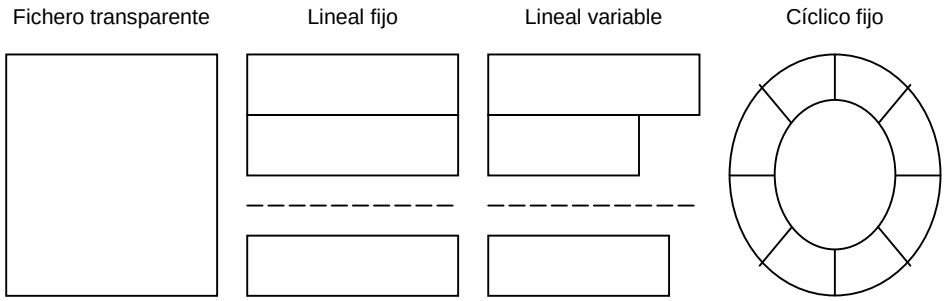


Figura 7: Estructuras de ficheros elementales

Un fichero transparente se estructura como una secuencia de bytes de datos, mientras que los otros tres tipos de EF's están estructurados como una secuencia de registros identificables individualmente. Un EF lineal fijo tiene registros de tamaño fijo; un fichero lineal variable tiene registros de tamaño variable; y un fichero cíclico tiene registros fijos organizados como un anillo.

En un fichero cíclico, los registros están dispuestos en orden inverso al orden en que fueron insertados en el fichero. El último registro insertado se identifica como registro 1. Una vez que el fichero está lleno, la próxima instrucción de escritura sobrescribe el registro más antiguo del fichero, y se convierte en el nuevo registro 1.

3.2.6 SISTEMAS DE SMART CARDS

Los sistemas de smart card, son sistemas distribuidos que constan de dos partes: el sistema host, residente en un ordenador conectado al lector o a un terminal y el sistema de la tarjeta, contenido en una smart card.

La mayoría del software para smart cards, incluido el software del sistema y las aplicaciones de usuario, corren en el lado del host. El software del sistema reconoce un tipo específico de smart card y arbitra la comunicación entre la aplicación de usuario y la tarjeta. El software del sistema también provee soporte para la infraestructura smart card, tales como la gestión de la tarjeta, seguridad y gestión de claves. Las aplicaciones de usuario implementan funciones que funcionan con una tarjeta específica o una aplicación específica de la tarjeta. Una simple aplicación de usuario es la única que soporta y maneja un conjunto de APDU's intercambiadas con la tarjeta, pero la mayoría de las aplicaciones de usuario tienen un rico juego de funciones. El software del host suele estar escrito en un lenguaje de alto nivel como Java, C ó C++.

El software de la tarjeta es el software que corre en la smart card. Esto incluye el software del sistema y el software de aplicación de usuario. El software del sistema suele incluir el sistema operativo y las utilidades que controlan la gestión de memoria, manejan las comunicaciones de Entrada/Salida con el host, aseguran la integridad de los datos y la seguridad, soportan el sistema de archivos ISO (si está implementado), y

proveen utilidades del sistema a las aplicaciones de las tarjetas. Las aplicaciones de la tarjeta contienen datos y soportan funciones que operan con los datos. El software de la tarjeta puede ser implementado en el lenguaje ensamblador del microprocesador de la tarjeta o en un lenguaje de alto nivel que pueda ser interpretado por el microprocesador.

Java Card ofrece una plataforma única en la que las aplicaciones del lado de la tarjeta se pueden escribir en Java y se pueden ejecutar en cualquier smart card que soporte el entorno de ejecución de Java Card.

3.2.7 ESTÁNDARES Y ESPECIFICACIONES DE SMART CARD

Durante los pasados 15 años, se han definido un gran número de estándares de smart cards y especificaciones, para asegurar que las smart cards, CAD's, y aplicaciones desarrolladas y fabricadas por diferentes vendedores puedan trabajar juntos. A continuación se enumeran algunos estándares y especificaciones (se explican los más importantes) para el desarrollo de los sistemas de smart cards.

3.2.7.1 Estándares de la ISO 7816

La ISO 7816, es el estándar más importante que define las características de los chips de las smart cards que poseen contactos eléctricos. La ISO 7816 comprende varios aspectos de las smart cards:

- Parte 1: características físicas.
- Pare 2: dimensiones y localizaciones de los contactos.
- Parte 3: señales eléctricas y protocolos de transmisión.
- Parte 4: comandos.
- Parte5: identificadores de las aplicaciones.
- Parte 6: elementos de dato.
- Parte 7: comandos para SCQL.

3.2.7.2 GSM

El instituto Europeo para la estandarización de las telecomunicaciones (ETSI) publica un conjunto de estándares que recoge el uso de las smart cards en sistemas de telefonía pública y celular. El Global System for Mobile Comunications (GSM) que está definido por la ETSI, es una especificación para un sistema de telecomunicación móvil internacional. En un principio estaba pensado para dar cobertura a algunos países de Europa central, pero ha ido creciendo, dando lugar a un estándar internacional para teléfonos móviles. Hay varios estándares de GSM. En particular:

- GSM 11.11: especificación del equipamiento de la interfaz SIM-teléfono.
- GSM 11.14: especificación del SIM Application Toolkit para la interfaz SIM-ME (mobile equipment).
- GSM 03.48: mecanismos de seguridad para el SIM Application Toolkit.

- GSM 03.19: SIM API (Application Programming Interface) para la plataforma Java Card. Este estándar, que se basa en GSM 11.11 y GSM 11.14, define la API de Java para desarrollar aplicaciones GSM que se ejecutan en la plataforma Java Card. La API es una extensión de la API de Java Card 2.1.

3.2.7.3 EMV

La especificación EMV, definida por Europa, MasterCard y Visa, está basada en las series de estándares de la ISO 7816 con características propietarias adicionales para cumplir con las necesidades específicas de la industria financiera. La última versión de estas especificaciones, la EMV 96 versión 3.1.1, fue publicada en Mayo de1998 y tiene tres partes:

- EMV ’96 Integrated Circuit Card Specification.
- EMV ’96 Integrated Circuit Card Terminal Specification.
- EMV ’96 Integrated Circuit Card Application Specification.

3.2.7.4 Open Platform

La Open Platform (OP) define un entorno integrado para el desarrollo y operación de sistemas smart card multiaplicación. La Open Platform consiste en una especificación para la tarjeta y una especificación para el terminal. La especificación para la tarjeta define los requisitos no específicos del producto para implementar una tarjeta Open Platform. Define las comunicaciones entre la tarjeta y el terminal, y la gestión de las aplicaciones de la tarjeta. Las especificaciones para el terminal definen la parte de la arquitectura de la aplicación en el terminal. Además define al terminal para que sea compatible con ISO y EMV.

Las especificaciones de la Open Platform fueron inicialmente desarrolladas por Visa y ahora ha sido transferidas a la GlobalPlatform, una organización que promueve una infraestructura global para la implementación de smart cards gracias al apoyo de múltiples industrias.

3.2.7.5 OpenCard Framework

El OpenCard Framework fue inicialmente producido por IBM y actualmente está siendo desarrollado por el consorcio OpenCard, que incluye a la mayor parte de las industrias de smart cards. OCF es el soporte de la aplicación del lado del host que ofrece una interfaz estándar para interaccionar con los lectores de tarjetas y las aplicaciones de la tarjeta. La arquitectura de OCF es un modelo estructurado. Cada una de las divisiones de las que se compone el modelo, se implementa por separado. Para realizar cada una de esas partes se tienen fabricantes de terminales de tarjetas, proveedores de sistemas operativos de tarjetas y proveedores de tarjetas. El objetivo es reducir la dependencia con cada una de esas partes y de la dependencia con los proveedores de plataformas.

En esencia, OCF está diseñado para el uso de una smart card en una red de ordenadores, y además está implementado en el lenguaje de programación Java.

3.2.7.6 PC/SC

Las especificaciones de PC/SC (Interoperability Specification for ICCs and Personal Computer Systems) pertenecen y están definidas por el grupo de trabajo PC/SC, un consorcio industrial al que pertenece la mayor parte de las empresas relacionadas con la industria de las smart cards. PC/SC define una arquitectura de propósito general para el uso de smart cards en sistemas de ordenadores.

En la arquitectura PC/SC, la parte de aplicación del host está construida por encima de uno o más proveedores de servicios y un gestor de recursos. Un proveedor de servicios encapsula la funcionalidad de una smart card específica y la hace accesible a través de interfaces de lenguajes de programación de alto nivel (como C ó C++). Un gestor de recursos gestiona los recursos más relevantes en el sistema de una smart card. Esto sirve para acceder a los dispositivos de aceptación de tarjetas y a través de ellos, a las smart cards individuales.

PC/SC y OCF tienen conceptos muy similares. Cuando corren bajo una plataforma Windows, OCF puede acceder a los CAD's a través del gestor de recursos PC/SC instalado.

4 LA TECNOLOGÍA JAVA CARD

Este capítulo está dedicado a presentar la tecnología Java Card. En él, se incluye una descripción de la arquitectura de tecnología Java Card, del modelo de objetos, de las excepciones, de la estructura de las aplicaciones (también llamadas applets) y de la forma en la que pueden compartir datos, y del protocolo de comunicación. Además, en este capítulo se muestran ejemplos de programación de las distintas partes de las que se compone una aplicación Java Card.

4.1 COMPONENTES DE LA TECNOLOGÍA JAVA CARD

La tecnología Java Card permite que los programas escritos en el lenguaje de programación Java se ejecuten en smart cards o en otros dispositivos con restricciones de recursos. Esta sección da una visión en conjunto de la tecnología Java Card (su arquitectura y sus componentes).

4.1.1 ARQUITECTURA

Las smart cards representan una de las plataformas de computación más pequeñas que se usa actualmente. La configuración de memoria de una smart card puede ser la siguiente: 1k de RAM, 16k de EEPROM y 24k de ROM. El gran reto del diseño de la tecnología Java Card es introducir el software del sistema Java en una smart card mientras se conserva suficiente espacio para las aplicaciones. La solución es soportar un subconjunto de características del lenguaje Java y aplicar un modelo dividido para implementar la máquina virtual de Java.

La máquina virtual de Java está dividida en dos partes: una que se ejecuta fuera de la tarjeta y otra que se ejecuta dentro de la tarjeta. Muchas tareas de procesado (carga de clases, verificación de los bytecodes, resolución, linkado y optimización) son realizadas por la máquina virtual que se ejecuta en el host, donde los recursos no son un problema.

Las smart cards difieren de los ordenadores de escritorio en varios aspectos. Además de proveer soporte para el lenguaje Java, la tecnología Java Card define un entorno de ejecución que soporta la memoria, comunicación, seguridad y el modelo de ejecución de las smart cards. El entorno de ejecución de Java Card cumple con el estándar internacional de smart cards ISO 7816.

La característica más importante del entorno de ejecución de Java Card es que provee una clara separación entre el sistema de la smart card y las aplicaciones. El entorno de ejecución oculta la complejidad subyacente y los detalles del sistema smart card. Las aplicaciones piden servicios y recursos del sistema a través de interfaces de lenguaje de alto nivel y bien definidas.

Por lo tanto, la tecnología Java Card define una plataforma sobre la que las aplicaciones escritas en el lenguaje de programación Java pueden ejecutarse en smart cards y otros dispositivos con restricciones de memoria (el término applet hace referencia a las aplicaciones escritas para la plataforma Java Card). Debido a la división de la arquitectura de la máquina virtual, esta plataforma se distribuye entre el entorno de

la smart card y el ordenador de escritorio. La plataforma se compone de tres partes, cada una definida en una especificación.

- El Java Card Virtual Machine (JCVM) Specification: define un subconjunto del lenguaje de programación Java y una definición de la máquina virtual deseable para las aplicaciones smart card.
- El Java Card Runtime Environment (JCRE) Specification: describe con precisión el comportamiento del entorno Java Card, incluyendo la gestión de memoria, la gestión de applets, y otras características de la ejecución.
- El Java Card Application Programming Interface (API) Specification: describe un conjunto del núcleo y extensión de los paquetes y clases de Java para programar las aplicaciones de las smart cards.

4.1.2 EL SUBCONJUNTO DEL LENGUAJE JAVA CARD

Debido a las restricciones de memoria, la plataforma Java Card solo soporta un subconjunto de características del lenguaje Java. Este subconjunto incluye características que son adecuadas para escribir programas para las smart cards y otros pequeños dispositivos. Incluso preserva las posibilidades de la orientación a objetos para el lenguaje de programación Java. La siguiente tabla muestra alguna de las características soportadas y no soportadas del lenguaje Java.

Tabla 3: Características de Java soportadas y no soportadas

Características soportadas de Java	Características no soportadas de Java
• Tipos de datos primitivos: `boolean`, `byte`, `short`. • Arrays de una dimensión. • Paquetes, clases, interfaces y excepciones de Java. • Características de orientación a objetos de Java. • La clave `int` y el tipo de dato entero de 32 bits son opcionales.	• Tipos de datos primitivos grandes: `long`, `double`, `float`. • Caracteres y cadenas. • Arrays multidimensionales. • Carga dinámica de clases. • Gestor de seguridad. • Recolector de basura. • Serialización de objetos. • Clonación de objetos.

4.1.3 LA MÁQUINA VIRTUAL DE JAVA

Una diferencia principal entre la máquina virtual de Java Card (JCVM) y la máquina virtual de Java (JVM) es que la JCVM está implementada como dos piezas separadas, como se muestra en la Figura 8. La parte de la máquina virtual de Java de la tarjeta incluye un intérprete Java Card de bytecodes. El convertidor Java Card se ejecuta en un PC o en una estación de trabajo. El convertidor es la parte de la máquina virtual de Java que se ejecuta fuera de la tarjeta.

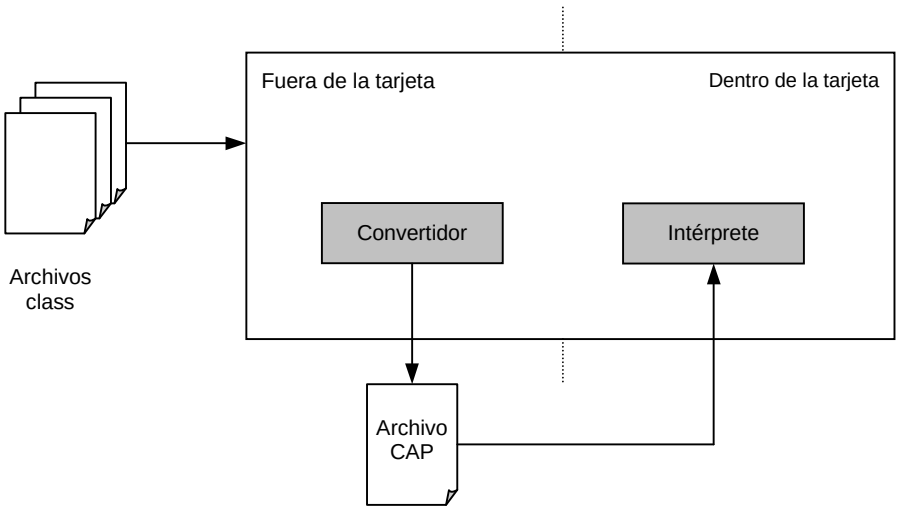


Figura 8: Máquina virtual de Java Card

Tomándolos juntos, implementan todas las funciones de la máquina virtual (carga de los archivos de clases de Java y ejecución de los mismos). El convertidor carga y preprocesa los archivos de clases, compone un paquete Java y saca un archivo de salida CAP (Converted Applet). Entonces el archivo CAP se carga en una smart card de Java y el intérprete lo ejecuta. Además de crear un fichero CAP, el convertidor genera un fichero de exportación que representa las API's públicas del paquete que ha sido convertido.

La tecnología Java Card soporta solamente un subconjunto del lenguaje Java. Igualmente, la máquina virtual de Java solo soporta las características que son requeridas por el subconjunto del lenguaje. Cualquier característica no soportada por el lenguaje y que sea usada en un applet, será detectada por el convertidor.

4.1.3.1 El fichero CAP y el fichero de exportación

La tecnología Java Card introduce dos nuevos formatos de ficheros binarios, que permiten el desarrollo independiente de la plataforma, y la distribución y la ejecución del software Java Card. Un fichero CAP contiene un representación binaria ejecutable de las clases de un paquete Java. Los archivos CAP tienen el mismo formato que el usado por los ficheros JAR. Un fichero CAP es un fichero JAR que contiene un conjunto de componentes, cada uno almacenado como un archivo individual en el fichero JAR. Cada componente describe un aspecto de los contenidos del fichero CAP, tales como información de la clase, bytecodes ejecutables, información de linkado, información de verificación, etc. El formato del fichero CAP está optimizado para ocupar poco espacio, usando estructuras de datos compactas y direccionamiento limitado. Define un conjunto de instrucciones bytecode que está basado y optimizado a partir del conjunto de instrucciones bytecode de Java.

La cualidad "write once, run anywhere" de los programas Java, es quizás la característica más importante de la plataforma Java. En la tecnología Java, el fichero de clase es la pieza central de la arquitectura Java. Define el estándar para la compatibilidad binaria de la plataforma Java. Debido a la característica distribuida de la arquitectura Java Card, el fichero CAP fija el formato de fichero estándar para la compatibilidad binaria de la plataforma Java Card. El formato del archivo CAP es la

forma en la que el software se carga en las smart cards Java. Por ejemplo, los ficheros CAP permiten la carga dinámica de clases de un applet, después de que la tarjeta haya sido fabricada.

Los ficheros de exportación no se cargan en el interior de las smart cards y por ello el intérprete no los usa directamente. Más bien, son producidos y consumidos por el convertidor para propósitos de verificación y linkado. Puede pensarse en los ficheros de exportación como los ficheros de cabecera (`.h`) en el lenguaje de programación C. Un fichero de exportación contiene información de las API's públicas para el paquete de clases entero. Define el alcance del acceso y el nombre de una clase, y el alcance del acceso y las firmas de los métodos y campos de la clase. Un fichero de exportación también contiene información de linkado, que se usa para resolver las referencias entre los paquetes que tiene la tarjeta.

El fichero de exportación no contiene ninguna implementación, es decir, no contiene bytecodes. Por ello un desarrollador de applets puede distribuir libremente un fichero de exportación, a los potenciales usuarios del applet, sin revelar detalles de implementación internos.

4.1.3.2 El convertidor de Java Card

A diferencia de la máquina virtual de Java, que procesa una clase a la vez, la unidad de conversión del convertidor es el paquete. El compilador de Java produce los ficheros de clases a partir del código fuente. Entonces, el convertidor procesa todos los ficheros de clases, construyendo un paquete Java que convierte en un fichero CAP.

Durante la conversión, el convertidor realiza tareas que una máquina virtual de Java en un entorno de escritorio podría realizar en el momento de la carga de clases:

- Verifica que las imágenes cargadas de las clases Java están bien formadas.
- Comprueba las violaciones del subconjunto del lenguaje Java Card.
- Realiza la inicialización de las variables estáticas.
- Resuelve las referencias simbólicas a las clases, métodos y campos en una forma más compacta para que puedan ser manejadas de forma más eficiente en la tarjeta.
- Optimiza los bytecodes, sacando ventaja de la información obtenida en el instante de la carga de las clases y del linkado.
- Distribuye el almacenaje y crea las estructuras de datos de la máquina virtual para representar a las clases.

El convertidor no solo toma como entrada los ficheros de clases para ser convertidos, sino también uno o más ficheros de exportación. Además de producir un fichero CAP, el convertidor genera un fichero de exportación para el paquete convertido. La Figura 9, demuestra como se convierte un paquete. El convertidor carga todas las clases de un paquete. Si el paquete importa clases de otros paquetes, el convertidor también carga los ficheros de exportación de aquellos paquetes. Las salidas del convertidor son un fichero CAP y un archivo de exportación por cada paquete que ha sido convertido.

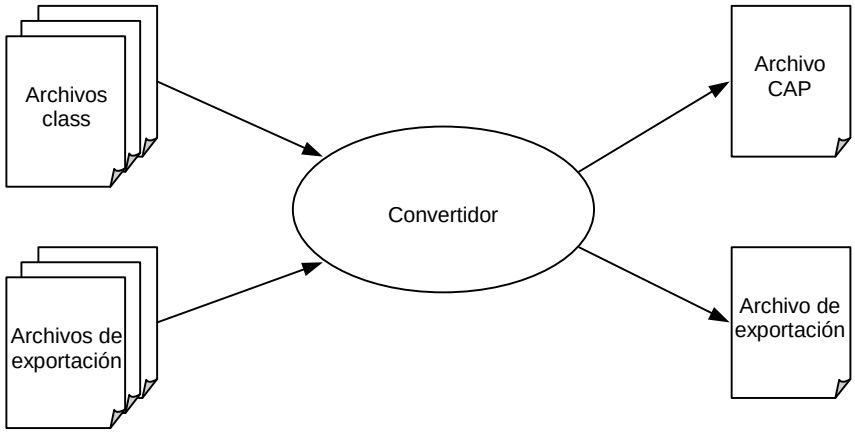


Figura 9: Conversión de un paquete

4.1.3.3 El Intérprete Java Card

El intérprete de Java provee soporte al modelo del lenguaje Java en tiempo de ejecución y por ello permite la independencia del hardware respecto del código del applet. El intérprete realiza las siguientes tareas:

- Ejecuta las instrucciones bytecode y ejecuta los applets.
- Controla el alojamiento en memoria y la creación de objetos.
- Juega un papel crucial en el control de la seguridad en tiempo de ejecución.

Hasta aquí, la máquina virtual de Java Card se ha descrito como formada por el convertidor y el intérprete. Sin embargo, la máquina virtual de Java Card se define informalmente como la pieza de la máquina virtual que se encuentra dentro de la tarjeta (el intérprete). Esta convención ha sido aplicada en muchas publicaciones de Java Card. Así pues, el término de intérprete de Java Card y la máquina virtual de Java Card son usados como sinónimos a no ser que se diga lo contrario. Pero debe tenerse en cuenta que las funciones de ejecución de los archivos de clases son llevados a cabo por el convertidor y el intérprete juntos.

4.1.4 EL INSTALADOR JAVA CARD Y EL PROGRAMA DE INSTALACIÓN FUERA DE LA TARJETA

El intérprete de Java Card no es en si mismo un cargador de ficheros CAP. Sólo ejecuta el código que se encuentra en el archivo CAP. En la tecnología Java Card, los mecanismos para descargar e instalar un fichero CAP se incorporan en una unidad llamada instalador.

El instalador de Java Card reside en la tarjeta. Coopera con un programa de instalación que se encuentra fuera de la tarjeta. Este programa de instalación que está fuera de la tarjeta, transmite el binario ejecutable al instalador (de la tarjeta) en un fichero CAP. Esta transmisión se realiza a través de un dispositivo de aceptación de tarjetas (CAD). El instalador escribe el binario en el interior de la memoria de la smart card, lo linka con otras clases que ya han sido introducidas en la tarjeta, y crea e inicializa cualquier estructura de datos que el entorno de ejecución de Java Card use

internamente. El instalador y el programa de instalación y como se relacionan con el resto de la plataforma Java se ilustran en la siguiente figura:

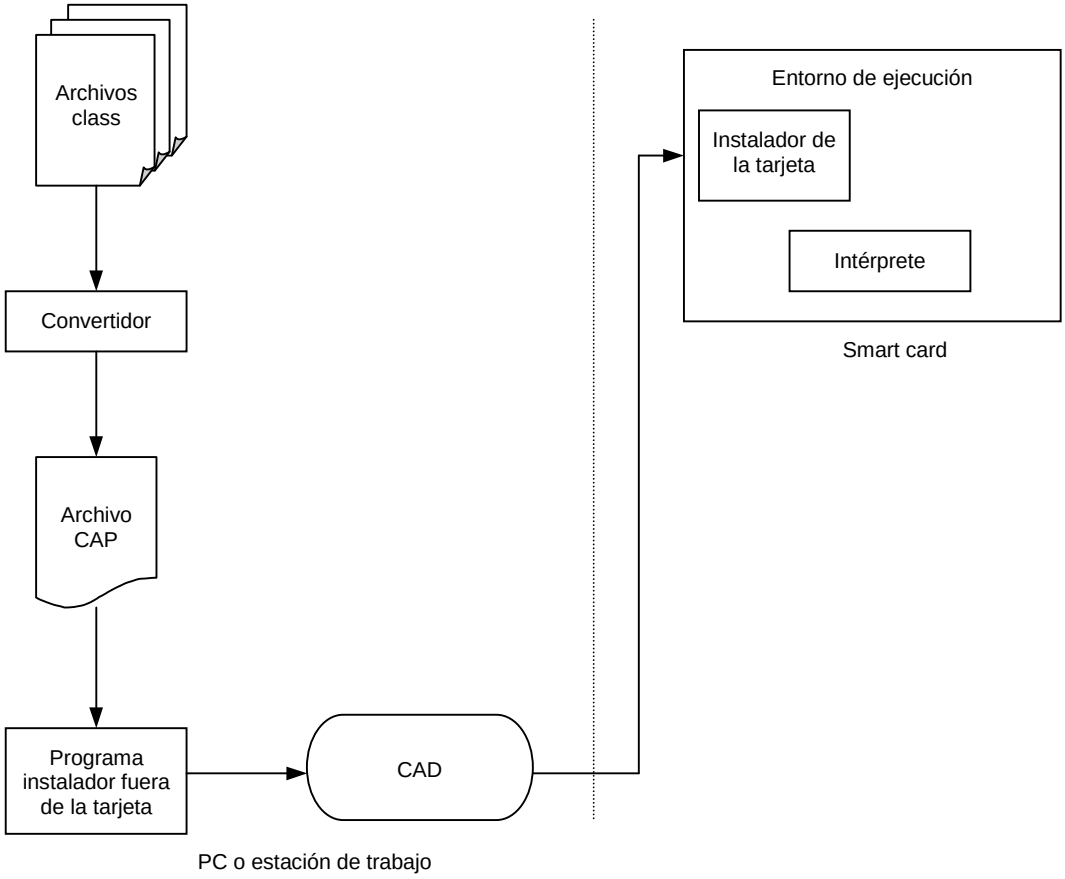


Figura 10: Instalador de Java Card y programa de instalación

La división de funcionalidad entre el intérprete y el instalador de archivos CAP mantienen el intérprete reducido y provee flexibilidad para implementaciones del instalador.

4.1.5 ENTORNO DE EJECUCIÓN DE JAVA CARD

El entorno de ejecución de Java Card (JCRE) consiste en una serie de componentes de un sistema Java Card y que se ejecutan en el interior de una smart card. El JCRE es responsable de la gestión de recursos, comunicaciones de red, ejecución de applets, y de la seguridad del sistema de la tarjeta y de los applets. De este modo, sirve esencialmente de sistema operativo de la smart card.

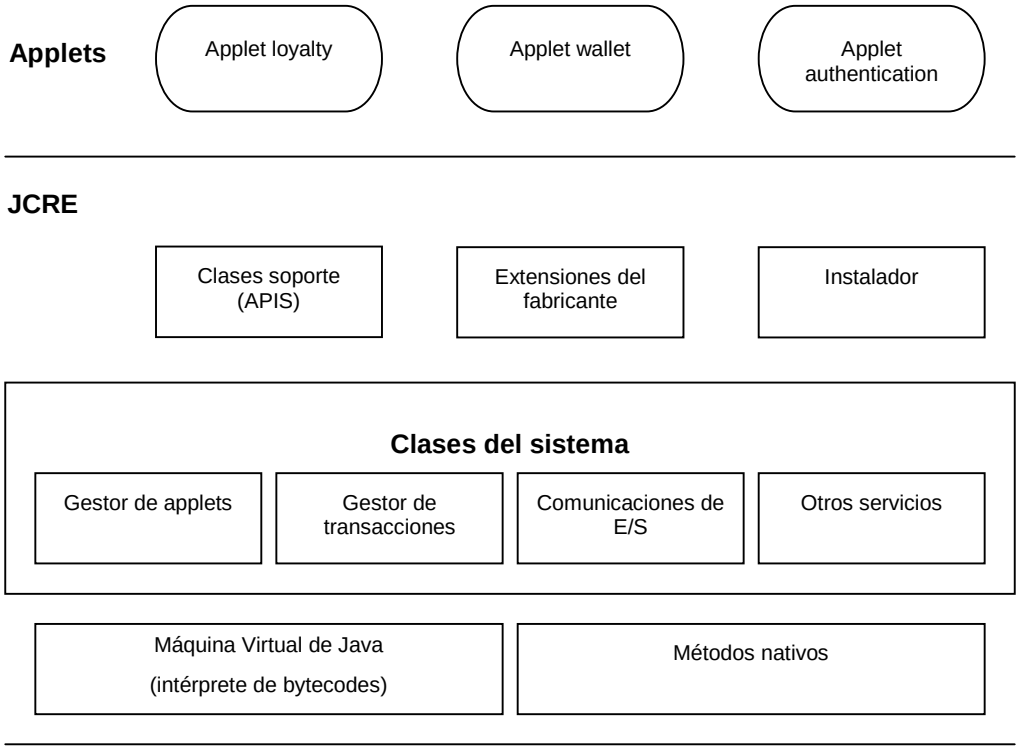


Figura 11: Arquitectura del sistema de la tarjeta

Como se ha ilustrado en la figura anterior, el JCRE está situado en lo alto del hardware y del sistema nativo de la smart card. El JCRE consta de una máquina virtual de Java (el intérprete de bytecodes), las clases de soporte de aplicaciones (API's), las extensiones específicas del fabricante y las clases del sistema JCRE. El JCRE separa bien los applets de las tecnologías propietarias de los fabricantes de tarjetas y provee un sistema estándar e interfaces API para los applets. Como resultado, los applets son más fáciles de escribir y son portables en varias arquitecturas de smart cards.

Los niveles bajos del JCRE contienen la máquina virtual de Java Card (JCVM) y los métodos nativos. El JCVM ejecuta bytecodes, controla el alojamiento en memoria, gestiona los objetos, y hace que se cumpla la seguridad en tiempo de ejecución. Los métodos nativos proveen soporte a la JCVM y a las clases de la siguiente capa. Ellos son los responsables del manejo de los protocolos de comunicación a bajo nivel, de la gestión de la memoria, del soporte criptográfico, etc.

Las clases del sistema actúan como los ayudantes del JCRE. Son análogos al núcleo de un sistema operativo. Las clases del sistema se encargan de gestionar las transacciones, gestionar la comunicación entre el host (las aplicaciones del host son las aplicaciones que corren en el lado del terminal, con las que los applets se comunican) y los applets de Java Card, y de controlar la creación, la selección y la deselección de los applets. Para completar las tareas, las clases del sistema suelen invocar a los métodos nativos.

Las clases soporte de aplicaciones de Java Card definen las interfaces de programación de las aplicaciones. El soporte consta de cuatro paquetes centrales y de

extensión. Las clases soporte (ó clases API) son compactas y están adaptadas para el desarrollo de applets de smart card. La mayor ventaja de este soporte es que hace relativamente fácil crear un applet. Los desarrolladores de applets pueden concentrar sus esfuerzos en los detalles de los applets antes que en los detalles de la infraestructura del sistema smart card. Los applets acceden a los servicios del JCRE a través de las clases API.

Un fabricante específico puede proporcionar librerías para proveer servicios adicionales o refinar la seguridad y el modelo del sistema. Por ejemplo el Open Platform extiende los servicios del JCRE para cubrir las necesidades específicas de seguridad que se requieren en las industrias financieras. Entre tantas características añadidas, se hace cumplir un control en las tarjetas que suministran los distribuidores y se especifica un conjunto de comandos estándar para la personalización de la tarjeta.

El instalador permite la descarga segura de software y applets en el interior de la tarjeta, una vez que ésta ya se ha fabricado. El instalador coopera con el programa de instalación que se encuentra fuera de la tarjeta. Juntos consiguen la tarea de cargar el contenido binario de los ficheros CAP. El instalador es un componente opcional del JCRE. Sin el instalador, todo el software de la tarjeta, incluyendo los applets, debe ser escrito en el interior de la memoria de la tarjeta durante el proceso de fabricación de la misma.

Los applets de Java Card son aplicaciones de usuario de la plataforma Java Card. Los applets son escritos en un subconjunto del lenguaje de programación Java y son controlados y gestionados por el JCRE. Los applets se pueden descargar al interior de la tarjeta. Los applets se pueden añadir a una tarjeta smart card, después de que ésta haya sido fabricada.

4.1.5.1 El tiempo de vida del JCRE

En un PC o en una estación de trabajo, la máquina virtual de Java se ejecuta como un proceso del sistema operativo. Los datos y los objetos se crean en RAM. Cuando el proceso del sistema operativo termina, las aplicaciones Java y sus objetos son destruidos automáticamente.

En una smart card de Java, la máquina virtual de Java Card se ejecuta en el entorno de ejecución de Java Card. El JCRE se inicializa en el momento de la inicialización de la tarjeta. La inicialización del JCRE se lleva a cabo una sola vez durante el tiempo de vida de la tarjeta. Durante este proceso, el JCRE inicializa la máquina virtual y crea objetos para proveer los servicios del JCRE y de los applets de gestión. Tan pronto como se instalan los applets, el JCRE crea las instancias de los applets, y los applets crean objetos para guardar datos.

La mayoría de la información de una tarjeta se debe preservar en los intervalos de tiempo, en los que no se suministra energía a la tarjeta. La tecnología de memoria persistente (tales como la EEPROM) se usa para conseguir preservar dicha información. Los datos y los objetos se crean en memoria persistente. El tiempo de vida del JCRE es completamente equivalente al tiempo de vida de la tarjeta. Cuando no se suministra energía a la tarjeta, la máquina virtual solo se encuentra suspendida. El estado del JCRE y los objetos creados en la tarjeta se preservan.

La próxima vez que se le dé energía a la tarjeta, el JCRE reanuda la ejecución de la máquina virtual leyendo datos de la memoria persistente (el JCRE también devuelve al host una respuesta al reset (ATR), indicando las características de la comunicación). Hay que señalar que el JCRE no continúa la operación de la máquina virtual en el punto exacto en el que se cortó la energía. La máquina virtual se resetea y se ejecuta a partir del comienzo del bucle principal. El reset del JCRE difiere de la inicialización, en que preserva los applets y los objetos creados en la tarjeta. Si no se completó una transacción previa al reset, el JCRE lleva a cabo cualquier tarea de limpieza para llevar al JCRE a un estado consistente.

4.1.5.2 ¿Como opera el JCRE durante una sesión con el CAD?

Al periodo, desde que la tarjeta se inserta en el dispositivo de aceptación de tarjetas (CAD) y se enciende, hasta el instante en que la tarjeta se retira del CAD, se le llama sesión CAD. Durante una sesión CAD, el JCRE opera como una smart card típica (soporta comunicación APDU I/O con la aplicación que se encuentra en un host). Las APDU's (unidades de datos del protocolo de aplicación) son paquetes de datos intercambiados entre los applets y la aplicación el host. Cada APDU contiene un comando proveniente del host dirigido a los applets o una respuesta del applet al host.

Después de un reset del JCRE, el JCRE entra en un bucle, esperando la llegada de APDU's de comando desde el host. El host envía APDU's de comando a la plataforma Java Card, usando la interfaz de comunicación serie a través del punto de contacto de E/S (entrada y salida) de la tarjeta.

Cuando llega un comando, el JCRE selecciona un applet para que ejecute la instrucción del comando o dirige el comando al applet actualmente seleccionado. Entonces el applet seleccionado toma el control y procesa la APDU de comando. Cuando termina, el applet envía la respuesta a la aplicación del host y abandona el control del JCRE. Este proceso se repite cuando llegue el próximo comando. La forma en que los applets procesan las APDU's se explica en los próximos apartados.

4.1.5.3 Características de Java Card en tiempo de ejecución

Además de soportar el modelo de tiempo de ejecución del lenguaje Java, el JCRE soporta tres características adicionales en tiempo de ejecución:

- Objetos persistentes y transitorios: Por defecto, los objetos de Java Card son persistentes y se crean en memoria persistente. El espacio y los datos para tales objetos se conservan entre sesiones CAD. Por seguridad y motivos de ejecución, los applets pueden crear objetos en RAM. Tales objetos se llaman objetos transitorios. Los objetos transitorios contienen datos temporales que no son persistentes a través de sesiones CAD.

- Operaciones y transacciones atómicas: La máquina virtual de Java asegura que cada operación de escritura en un solo campo de un objeto o en una clase, es atómica. El campo actualizado toma el nuevo valor o se devuelve a su valor anterior. Además, el JCRE provee API's de transacción. Un applet puede incluir varias operaciones de escritura en una transacción. O se completan todas las actualizaciones implicadas en una transacción o (si

algún error ocurre en mitad de la transacción) ninguna de ellas se lleva a cabo.

- El applet firewall y mecanismos de compartición. El applet firewall aísla a los applets. Cada applet se ejecuta en un espacio designado para ello. La existencia y las operaciones de un applet no tiene efecto en los otros applets de la tarjeta. El applet firewall se fuerza por la máquina virtual de Java Card que ejecuta bytecodes. En situaciones donde los applets necesitan compartir datos o acceder a los servicios del JCRE, la máquina virtual permite tales funciones a través de mecanismos seguros de compartición.

4.1.6 LAS API'S DE JAVA CARD

Las API's de Java Card consisten en un juego de clases personalizadas para programar aplicaciones de smart cards acordes al modelo ISO 7816. Las API's contienen tres paquetes centrales y un paquete de extensión. Los tres paquetes centrales son `java.lang`, `javacard.framework` y `javacard.security`. El paquete de extensión es `javacard.crypto`.

Los desarrolladores que estén familiarizados con la plataforma Java notarán que las API's de Java Card no soportan muchas de las clases de la plataforma Java. Por ejemplo, las clases para las interfaces GUI (interfaz gráfica de usurario) de la plataforma Java, la E/S de red, el sistema de E/S de ficheros de los sistemas de escritorio no se soportan. La razón de esto, es que las smart cards no tienen un display, usan un protocolo de red diferente y una estructura de sistema de ficheros diferente. También, no se soportan muchas de las clases de utilidades de la plataforma Java, para cumplir con las estrictas restricciones de memoria.

Las clases en las API's de Java Card son compactas y cortas. Incluyen clases adaptadas de la plataforma Java para proveer soporte al lenguaje Java y a los servicios de criptografía. También contienen clases creadas específicamente para soportar el estándar ISO 7816 de smart card.

4.1.6.1 Paquete `java.lang`

El paquete `java.lang` de Java Card es un riguroso subconjunto equivalente al paquete `java.lang` de la plataforma Java. Las clases soportadas son `Object`, `Throwable`, y algunas clases de excepciones relacionadas con la máquina virtual, que se muestran en la siguiente tabla:

Tabla 4: Paquete `java.lang` de Java Card

`Object`	`Throwable`	`Exception`
`RuntimeException`	`ArithmeticException`	`ArrayIndexOutOfBoundsException`
`ArrayStoreException`	`ClassCastException`	`IndexOutOfBoundsException`
`NullPointerException`	`SecurityException`	`NegativeArraySizeException`

Para las clases soportadas, muchos de los métodos de Java no están disponibles. Por ejemplo, la clase `Object` de Java Card sólo define un constructor por defecto y el método `equals`.

El paquete `java.lang` provee el soporte fundamental para el lenguaje Java. La clase `Object` define una raíz para la jerarquía de clases de Java Card, y la clase `Throwable` provee un ascendiente común para todas las excepciones. Las clases de excepciones soportadas aseguran semánticas consistentes cuando ocurre un error debido a una violación del lenguaje Java.

Por ejemplo, la máquina virtual de Java y la máquina virtual de Java Card, arrojan una `NullPointerException` cuando se accede a una referencia que es nula.

4.1.6.2 Paquete `javacard.framework`

El paquete `javacard.framework` es esencial. Provee clases soporte e interfaces para la funcionalidad principal de un applet Java Card. Lo más importante, es que define una clase base `Applet`, que provee un soporte para la ejecución del applet y su interacción con el JCRE durante el tiempo de vida del applet. Su papel respecto al JCRE es similar al de la clase `Applet` en el explorador de un ordenador. Un desarrollador que quiera implementar un applet deberá extender la clase base `Applet` y sobrescribir los métodos de la clase `Applet` para implementar la funcionalidad del applet.

Otra clase importante incluida en el paquete `javacard.framework`, es la clase `APDU`. Las APDU's son llevadas por el protocolo de transmisión. Los dos protocolos estandarizados de transmisión son el T=0 y el T=1. La clase `APDU` está diseñada para ser independiente del protocolo de transmisión. En otras palabras, está cuidadosamente diseñada para que la complejidad y las diferencias entre los protocolos T=0 y T=1 se oculten para los desarrolladores de applets. Los desarrolladores de applets pueden tratar las APDU's de tipo comando mucho más fácilmente, usando los métodos provistos en la clase `APDU`. Los applets trabajan correctamente a pesar del protocolo de transmisión subyacente que la plataforma soporte.

La clase `java.lang.System` de la plataforma Java, no está soportada. La plataforma Java Card suministra la clase `javacard.framework.JCSystem`, que provee una interfaz para examinar el comportamiento del sistema. La clase `JCSystem` incluye una colección de métodos que controlan la ejecución del applet, la gestión de recursos, la gestión de transacciones y la compartición de objetos entre applets, en la plataforma Java Card.

Otras clases soportadas en el paquete `javacard.framework` son las clases relacionadas con el PIN, las de utilidad y las de excepciones. El PIN es un número corto que se utiliza en la identificación personal. Es la forma más común de contraseña usada en las smart cards para autentificar a los titulares de las tarjetas.

4.1.6.3 Paquete `javacard.security`

El paquete `javacard.security` provee un soporte para las funciones de criptografía soportadas en la plataforma Java Card. Su diseño está basado en el paquete `java.security`.

El paquete `javacard.security` define una clase de clave llamada `KeyBuilder` (clase que viene de fábrica) y varios interfaces que representan claves criptográficas usadas en algoritmos simétricos (como DES) o asimétricos (como DSA o RSA). Además, soporta las clases base abstractas `RandomData`, `Signature`, y `MessageDigest`, que se usan para generar datos aleatorios y calcular las asimilaciones y las firmas de los mensajes.

4.1.6.4 Paquete `javacardx.crypto`

El paquete `javacardx.crypto` es un paquete de extensión. Contiene clases de criptografía e interfaces que están sujetas a las exigencias expresadas en las regulaciones de exportación de los Estados Unidos. El paquete `javacardx.crypto` define la clase base abstracta `Cipher` para soportar funciones de codificación y decodificación.

Los paquetes `javacard.security` y `javacardx.crypto` definen las interfaces a las que los applets llaman para pedir los servicios de criptografía. Sin embargo, no proveen ninguna implementación. Un proveedor de JCRE necesita suministrar clases que implementen las interfaces de clave y extenderlas a partir de las clases abstractas `RandomData`, `Signature`, `MessageDigest` y `Cipher`. Normalmente existe un coprocesador en las smart cards para llevar a cabo los cálculos criptográficos.

4.1.7 LOS APPLETS DE JAVA CARD

Los applets de Java Card no se deberían confundir con los applets de Java porque todos ellos se llamen applets. Un applet de Java Card es un programa Java que cumple un conjunto de convenciones que permiten ejecutarlo en el entorno de ejecución de Java Card. Un applet de Java Card no está pensado para ejecutarse en un explorador. La razón de poner el nombre de applet a las aplicaciones de Java Card se encuentra en que los applets se pueden cargar en el entorno de ejecución después de que la tarjeta haya sido fabricada. Es decir, a diferencia de las aplicaciones de muchos sistemas embebidos, los applets no necesitan ser "quemados" en la ROM durante el proceso de fabricación. Más bien, los applets pueden ser bajados dinámicamente a la tarjeta en un instante posterior a la fabricación.

La clase de un applet debe extenderse a partir de la clase `javacard.framework.Applet`. La clase base `Applet` es la superclase para todos los applets residentes en una tarjeta Java Card. La clase de un applet es un proyecto que define las variables y métodos de un applet. Un applet que se ejecuta en la tarjeta es una instancia de un applet (es decir, un objeto de la clase de ese applet). Como ocurre con cualquier objeto persistente, una vez creado, un applet vive en la tarjeta para siempre.

El entorno de ejecución de Java Card soporta un entorno multiaplicación. Pueden coexistir múltiples applets en una sola smart card, y un applet puede tener múltiples instancias. Por ejemplo, una instancia del applet wallet (que será visto en la sección 9.3.2.5.1, página 240) puede ser creado para soportar el dólar americano y otro puede ser creado para la libra inglesa.

4.1.8 CONVENCIÓN DE NOMBRADO DE PAQUETES Y APPLETS

Los paquetes y los programas de la plataforma Java se identifican usando cadenas Unicode y el esquema de nombrado está basado en el dominio de nombres de Internet. Sin embargo, en la plataforma Java Card cada instancia de applet está unívocamente identificada por un identificador de aplicación (AID). También, cada paquete Java tiene asignado un AID. Cuando se carga en la tarjeta, un paquete se linka con otros paquetes de la tarjeta a través de sus AID's.

La ISO 7816 especifica los AID's usados en la identificación de las aplicaciones de la tarjeta y de cierto tipo de ficheros del sistema de ficheros de la tarjeta. Un AID es un array de bytes que se puede interpretar como dos partes distintas, como se muestra a continuación:

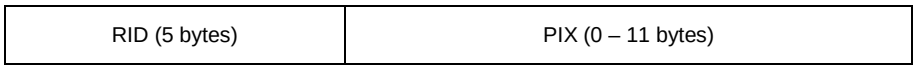


Figura 12: Identificador de aplicación (AID)

La primera parte es un valor de 5 bytes conocido como RID (Resource Identifier). La segunda parte es un valor de longitud variable conocido como PIX (Propietary Identifier Extension). Un PIX puede tener una longitud de 0 a 11 bytes. De este modo un AID puede tener una longitud de 5 a 16 bytes en total.

La ISO controla la asignación de RID's a las compañías. Cada compañía tiene un RID único. Las compañías gestionan la asignación de PIX's. Esta sección ofrece una corta descripción de los AID's. Para más información consultar la ISO 7816-5, AID Registration Category D format.

En la plataforma Java Card, el AID de un paquete se construye concatenando el nombre de la compañía y el PIX de ese paquete. El AID de un applet se construye de forma similar que para el AID de un paquete. Es la concatenación del RID del proveedor del applet y el PIX de ese applet. El AID de un applet no debe tener el mismo valor que el AID de cualquier paquete o el AID de cualquier otro applet. Sin embargo, ya que el RID en un AID identifica a un proveedor de applets, el AID del paquete y el (los) AID(s) del (de los) applet(s) definido(s) en el paquete deben compartir el mismo RID.

El AID del paquete y el AID del applet por defecto para cada applet definido en el paquete, se especifican en el fichero CAP. Estos se suministran al convertidor cuando se genera el fichero CAP.

4.1.9 EL PROCESO DE DESARROLLO DE APPLETS

El desarrollo de un applet de Java Card comienza como cualquier otro programa de Java: un desarrollador escribe una o más clases de Java y compila el código fuente con un compilador de Java, produciendo uno o más ficheros de clase. La siguiente figura demuestra el proceso de desarrollo de un applet:

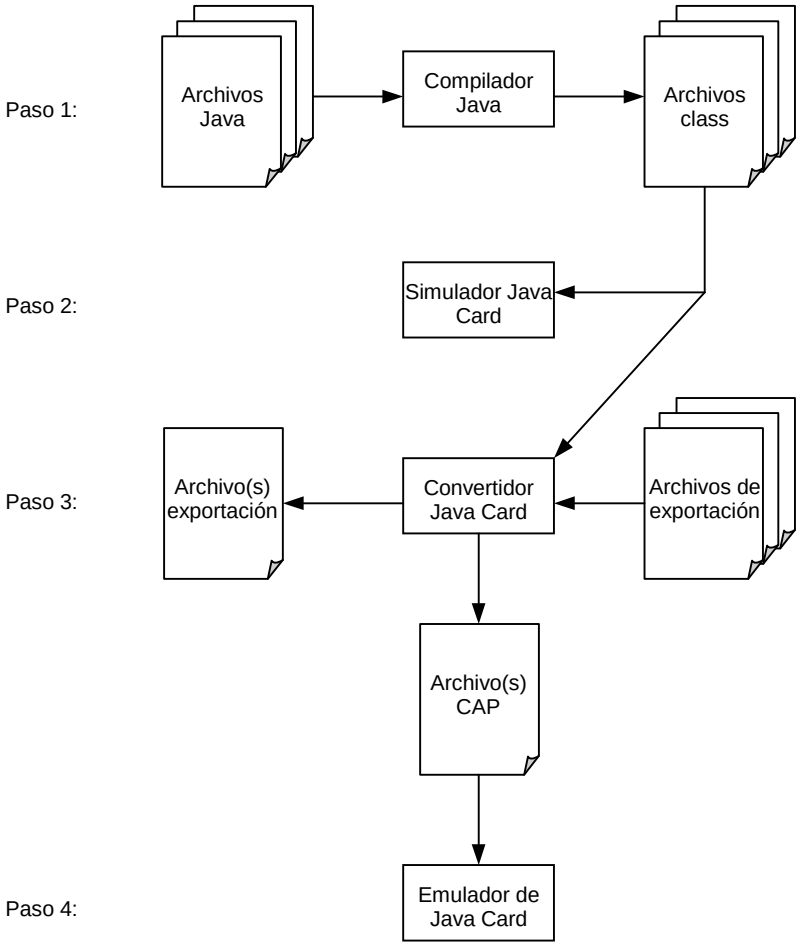


Figura 13: Proceso de desarrollo de un applet

A continuación, el applet se ejecuta, se testea y se depura en un entorno de simulación. El simulador emula el entorno de ejecución de Java Card en un PC o en una estación de trabajo. En el entorno de simulación, el applet se ejecuta en una máquina virtual de Java, y de este modo se ejecutan los archivos class de los applets. El simulador puede utilizar muchas herramientas de desarrollo de Java (la máquina virtual, el depurador y otras herramientas) y permite al desarrollador testear el comportamiento del applet y ver rápidamente los resultados sin tener que realizar los procesos de conversión. Durante este paso, se testean la totalidad de los aspectos de funcionalidad del applet. Sin embargo, algunas de las características de la máquina virtual de Java Card, como el applet firewall y el comportamiento de los objetos transitorios y persistentes, no se pueden examinar.

Entonces, los archivos de clases del applet que se integran en un paquete Java, se convierten en un archivo CAP usando el convertidor de Java Card. El convertidor de Java Card toma como entrada, no solo los archivos class a ser convertidos, sino también uno o más ficheros de exportación. Cuando el paquete del applet se convierte, el convertidor puede producir también un archivo de exportación para ese paquete. Un archivo CAP o un archivo de exportación representan a un solo paquete Java. Si un applet comprende varios paquetes, se crean un archivo CAP y un archivo de exportación por cada uno de los paquetes.

En el siguiente paso, el archivo(s) CAP que representa(n) al applet, se cargan y se testean en un entorno de simulación. El emulador también simula el entorno de

ejecución de Java Card en un PC o en una estación de trabajo. Sin embargo, el emulador es una herramienta de testeo muy sofisticada. Efectúa la implementación de una máquina virtual de Java Card. El comportamiento del applet ejecutándose en el emulador debería ser el mismo que su comportamiento corriendo en una tarjeta real. En esta fase de desarrollo, no sólo se testea el applet, sino también el comportamiento del applet en tiempo de ejecución.

La mayoría de los simuladores y emuladores vienen con un depurador. El depurador permite al desarrollador introducir puntos de parada o ejecutar paso a paso, viendo el estado de la ejecución del applet en el simulador o emulador (del entorno de ejecución de Java Card).

Finalmente, cuando el applet está testeado y listo para ser descargado en una tarjeta real, el applet, representado por un o varios archivos CAP, se carga e instala en la smart card de Java.

4.1.10 INSTALACIÓN DE APPLETS

Cuando una smart card de Java se fabrica, el sistema propietario de la smart card y el entorno de ejecución de Java Card, incluyendo los métodos nativos, la máquina virtual de Java, las clases API y las librerías, se "queman" en la ROM. Este proceso de escritura en componentes permanentes en un chip de una memoria ROM, se llama masking. La tecnología para llevar a cabo el masking, es una tecnología propietaria del vendedor de smart cards (apartado en el que no se entrará).

4.1.10.1 ROM Applets

Las clases de applets de Java Card se pueden introducir en memoria ROM junto con el JCRE y otros componentes del sistema durante el proceso de fabricación de la tarjeta. Las instancias de los applets que son "instanciadas" en EEPROM por el JCRE, durante la inicialización del JCRE o en una etapa posterior, se llaman ROM applets.

Los ROM applets son los applets por defecto que vienen con la tarjeta y son provistos por los suministradores de tarjetas. Debido a que los contenidos del ROM applet están controlados por los suministradores, la tecnología Java Card permite a los ROM applets declarar métodos nativos, cuyas implementaciones están escritas en otro lenguaje de programación, como C o ensamblador. Los métodos nativos no están sujetos a las comprobaciones de seguridad forzadas por la máquina virtual de Java.

4.1.10.2 Preissuance or Postissuance applets

Alternativamente, las clases de applet de Java Card y las librerías de clases asociadas se pueden descargar y escribir en la memoria (como la EEPROM) de una smart card después de que la tarjeta haya sido fabricada. Tales applets son llamados preissuance o postissuance applets. Los términos preissuance y proissuance derivan del hecho de que los applets se descargan después de que la tarjeta haya sido suministrada (issue). Los preissuance applets se tratan de la misma manera que los ROM applets, ya que ambos son controlados por el suministrador.

A diferencia de los ROM applets y los preissuance applets, a los postissuance applets no se les permite declarar métodos nativos. La razón es que el JCRE no tiene manera de controlar los contenidos de este tipo de applets. Permitir que los applets descargados contengan código nativo podría comprometer la seguridad de Java Card.

4.1.10.3 Instalación del postissuance applet

La instalación del applet se refiere al proceso de cargar las clases del applet en un archivo CAP, combinarlas con el estado de ejecución del entorno de ejecución de Java Card, y crear la instancia de un applet para llevar al applet a un estado de selección y ejecución.

En la plataforma Java, la unidad de carga e instalación es el archivo CAP. Un archivo CAP consiste en las clases que componen un paquete de Java. El applet mínimo es un paquete Java con una sola clase derivada de la clase `javacard.framework.Applet`. Un applet más complejo con un cierto número de clases, puede ser organizado en un paquete Java o en un conjunto de paquetes Java.

Para cargar un applet, el instalador que se encuentra fuera de la tarjeta toma el fichero CAP y lo transforma en una secuencia de APDU's de comando, que llevarán el contenido del fichero CAP. Para intercambiar comandos con el programa de instalación fuera de la tarjeta, el instalador de la tarjeta escribe el contenido del fichero CAP en memoria persistente y linka las clases del fichero CAP con otras clases que residen en la tarjeta. El instalador también crea e inicializa cualquier dato que sea usado internamente por el JCRE para dar soporte al applet. Si el applet requiere varios paquetes para ejecutarse, se carga cada fichero CAP en la tarjeta.

En el último paso de la instalación, el instalador crea una instancia del applet y registra la instancia en el JCRE (en una implementación del JCRE, la operación de crear una instancia de un applet puede llevarse a cabo en una etapa posterior a la instalación del applet). Hecho todo esto, el instalador invoca al método `install`:

```
public static void install(byte[] bArray, short offset, byte length)
```

El método `install` es un método de punto de entrada al applet, similar al método `main` en una aplicación Java. Un applet siempre debe implementar el método `install`. En el método `install`, se llama al constructor del applet para crear e inicializar una instancia del applet. El parámetro `bArray` del método `install` suministra los parámetros de instalación para la inicialización del applet. Los parámetros de instalación se envían a la tarjeta desde el principio con el archivo CAP. El desarrollador de applets define el formato y el contenido de los parámetros de instalación.

Después de que el applet haya terminado el proceso de inicialización y de registro en el JCRE, ya se puede seleccionar y ejecutar. El JCRE identifica al applet en ejecución (una instancia del applet) usando un AID. El applet puede registrarse en el JCRE usando el AID por defecto que se encuentra en el fichero CAP, o puede elegir otro diferente. Los parámetros de instalación se pueden usar para suministrar un AID alternativo.

El método `install` se puede llamar más de una vez para crear múltiples instancias de un applet. Cada instancia del applet se identifica por un único AID.

En el entorno de Java Card, un applet se puede escribir y ejecutar sin saber como se cargan sus clases. La única responsabilidad del applet durante la instalación es implementar el método `install`.

4.1.10.4 Recuperación de errores durante la instalación del applet

El proceso de instalación es transaccional. En el caso de que se produzca un error, tal como un fallo de programación, falta de memoria, daño de la tarjeta, u otros errores, el instalador descarta el fichero CAP y cualquier applet que haya sido creado durante la instalación y recupera el espacio y el estado previo del JCRE.

4.1.10.5 Restricciónes en la instalación

Se debe tener en cuenta que la instalación del applet es diferente de la carga dinámica de clases en tiempo de ejecución, que se supone en una máquina virtual de Java en un entorno de escritorio. La instalación de un applet de Java Card significa simplemente descargar clases a través de un proceso de instalación después de que la tarjeta haya sido fabricada.

Por lo tanto, la instalación de applets de Java Card tiene dos puntos delicados. Primero, los applets que se están ejecutando en la tarjeta solo podrán hacer referencia a clases que ya existan en la tarjeta, ya que no hay forma de descargar clases durante la ejecución normal de código de applets.

En segundo lugar, el orden de carga debe garantizar que cada nuevo paquete descargado haga referencia solo a paquetes que ya están listos en la tarjeta. Por ejemplo, para instalar un applet, el paquete `javacard.framework` debe estar presente en la tarjeta, porque todas las clases de applets deben extenderse de la clase `javacard.framework.Applet`. Una instalación fallaría si hubieran un paquete A y un paquete B que se referenciaran mutuamente.

4.2 OBJETOS DE JAVA CARD

En la tecnología Java Card, el JCRE y los applets crean objetos para representar, guardar y manipular datos. Los applets están escritos usando el lenguaje de programación Java. Los applets ejecutables en la tarjeta son objetos de clases de applet.

Los objetos en la plataforma Java Card están sujetos a las reglas de programación de Java:

- Todos los objetos de la plataforma Java Card son instancias de clases o tipos de array, las cuales tienen la misma clase raíz `java.lang.Object`.
- Los campos en un objeto nuevo o los componentes de un array nuevo, están inicializados a sus valores por defecto (`cero`, `null` o `false`) a menos que sean inicializados en el constructor con otros valores.

La tecnología Java Card soporta tanto objetos persistentes como transitorios. Sin embargo, los conceptos de objetos transitorios y persistentes no son los mismos que en la plataforma Java.

4.2.1 MODELO DE MEMORIA DE JAVA CARD

Una smart card tiene tres clases de memoria: ROM, RAM y EEPROM. La memoria ROM es una memoria de solo lectura y es la menos cara de las tres. Los programas y los datos se "queman" en la ROM durante el proceso de fabricación de la tarjeta. Tanto la memoria RAM como la EEPROM se pueden leer y escribir, pero difieren en muchas características eléctricas. En el caso de pérdida de energía, la memoria RAM pierde su contenido, pero el contenido de la EEPROM se preserva. Las operaciones de escritura en la EEPROM son unas 1000 veces más lentas que las operaciones de escritura en la RAM, y el número posible de escrituras en la EEPROM durante el tiempo de vida de la tarjeta está limitado físicamente. Además, una celda de memoria RAM tiende a ser 4 veces mayor que una celda de EEPROM. Las smart cards actuales suelen ofrecer unos 16k de EEPROM y 1k de RAM.

El modelo de memoria de Java Card, está influenciado por la clase y características físicas de la memoria que usan las smart cards. Un sistema Java Card típico suele colocar el código del JCRE (máquina virtual, clases de la API, y otro software) en ROM. El código de los applets también se puede emplazar en ROM. La RAM se usa para el almacenamiento temporal. La pila en tiempo de ejecución de Java Card se aloja en RAM. Los resultados intermedios, parámetros de métodos y las variables locales se colocan en la pila. Los métodos nativos, como aquellos que llevan a cabo cálculos criptográficos, también salvan sus resultados intermedios en RAM. Los datos con una vida larga se guardan en la EEPROM, como es el caso de las clases del applet que son descargadas.

La mayoría de los objetos de JCRE y applets, representan información que se necesita preservar cuando la tarjeta no esté conectada a una fuente de energía. Debido al ratio de tamaño RAM/EEPROM, en una smart card se hace necesario designar espacio para los objetos en memoria EEPROM.

Sin embargo, se suele acceder con frecuencia a ciertos objetos que poseen datos (los contenidos de sus campos) que no necesitan ser persistentes (así los accesos a memoria son más rápidos). Por ello la tecnología Java Card también soporta objetos transitorios en RAM. Los objetos transitorios se crean invocando las API's de Java Card.

Objetos persistentes y transitorios en la plataforma Java

En la plataforma Java, los objetos se crean en RAM. Los objetos se destruyen automáticamente cuando la máquina virtual de Java existe, o cuando el recolector de basura los elimina. Las propiedades, campos e información de estado de algunos objetos, se puede preservar usando los mecanismos de serialización y de deserialización de objetos. La serialización de objetos graba el estado actual y las propiedades de un objeto en una cadena de bytes. La cadena se puede deserializar más tarde para restaurar el objeto, con el mismo estado y propiedades. El lenguaje Java también soporta la palabra clave `transient`. Los campos se marcan como `transient` para indicar que no forman parte del estado persistente de un objeto. Los campos transitorios no se salvan durante la serialización de un objeto.

La tecnología Java Card no soporta la serialización de objetos o la palabra clave `transient`.

4.2.2 OBJETOS PERSISTENTES

La memoria y datos de los objetos persistentes se preservan en los periodos de tiempo que transcurren entre sesiones CAD. Un objeto persistente tiene las siguientes propiedades:

- Un objeto persistente se crea mediante el operador `new`.
- Un objeto persistente mantiene estados y valores entre las sesiones CAD.
- Cualquier actualización de un solo campo de un objeto persistente es atómica. Es decir, si a la tarjeta se le corta el suministro de energía u ocurre un fallo durante la actualización, el campo se restaura a su valor previo.
- Un objeto persistente se puede referenciar mediante un campo de un objeto transitorio.
- Si un objeto persistente deja de ser referenciado por otros objetos, se vuelve inalcanzable o puede ser eliminado por el recolector de basura.

Cuando se crea la instancia de un applet, como cualquier objeto persistente, el espacio y datos del applet, persisten indefinidamente de una sesión CAD a otra.

4.2.3 OBJETOS TRANSITORIOS

El término transitorio es algo inapropiado. Se puede interpretar incorrectamente cuando lo que se quiere decir es que el objeto es temporal: cuando se le quita la energía a la tarjeta, el objeto transitorio se destruye. De hecho, el término objeto transitorio significa que el contenido de los campos del objeto tienen una naturaleza temporal. Como con los objetos persistentes, el espacio en el que se alojan los objetos transitorios está reservado y no se puede recuperar a menos que se implemente el recolector de basura.

Un applet debería crear un objeto transitorio solo una vez durante su tiempo de vida y debería salvar la referencia del objeto en un campo persistente, como se muestra a continuación:

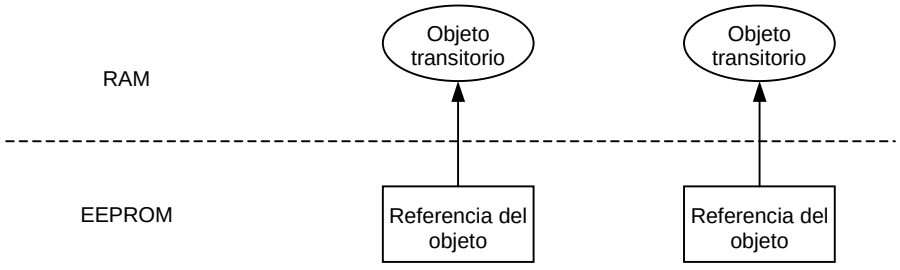


Figura 14: Objetos transitorios

La próxima vez que a la tarjeta se le suministra energía, el applet usa la misma referencia del objeto para acceder al objeto transitorio, aunque se perdieran los datos del objeto de la anterior sesión.

4.2.3.1 Propiedades de los objetos transitorios

En Java Card 2.1, solo los arrays con tipos primitivos o arrays con referencias a `Object` se pueden declarar como transitorios. Los tipos primitivos en la plataforma Java Card son `byte`, `short`, `int` y `boolean`. De aquí en adelante los términos objeto transitorio y array transitorio se usan indistintamente. Un objeto transitorio en la plataforma Java Card tiene las siguientes propiedades:

- Un objeto transitorio se crea invocando las API's de Java Card.
- Un objeto transitorio no mantiene estados ni valores entre sesiones CAD. Los campos de un objeto transitorio se ponen a su valor por defecto (`cero`, `false` o `null`) cuando ocurren ciertos eventos.
- Cualquier actualización de un solo campo de un objeto transitorio es no atómica. Es decir, si la tarjeta pierde su suministro de energía u ocurre un fallo durante una actualización, no se restaura el valor del campo a su valor anterior. Si las escrituras a los campos de un objeto transitorio están incluidas en un transacción, una transacción fallida (porque salga mal) nunca causa la restauración del campo del objeto transitorio a su valor previo.
- Un campo de un objeto persistente puede referenciar a un objeto transitorio.
- Un campo de un objeto transitorio puede referenciar a un objeto persistente.
- Si un objeto transitorio deja de ser referenciado por otros objetos, se vuelve inalcanzable o puede ser eliminado por el recolector de basura.
- Las escrituras a campos de un objeto transitorio no produce penalización en tiempo, porque la RAM tiene un ciclo de escritura más rápido que la EEPROM.

Las propiedades de los objetos transitorios los hace ideales para pequeñas cantidades de datos temporales (de applets) que son frecuentemente modificados, pero que no necesitan ser preservados entre sesiones CAD. El desarrollador de applets debería asegurar que tales datos temporales se almacenan en arrays transitorios. Esto reduce el uso de memoria persistente, garantizando así unos mejores tiempos en escritura, y añadiendo seguridad para proteger a los datos más delicados. Como regla de actuación, si los datos temporales están siendo actualizados múltiples veces por cada APDU procesada, el desarrollador de applets debería mover estos datos a un array transitorio.

4.2.3.2 Tipos de objetos transitorios

Hay dos tipos de objetos transitorios: `CLEAR_ON_RESET` y `CLEAR_ON_DESELECT`. Cada tipo de objeto transitorio está asociado a un evento, que cuando ocurre, hace que el JCRE limpie los campos de los objetos.

Los objetos transitorios `CLEAR_ON_RESET` se usan para mantener datos que necesitan ser preservados entre selecciones de un applet pero no entre reseteos de la tarjeta. Por ejemplo, una clave de sesión principal se debería declarar del tipo `CLEAR_ON_RESET` para que la misma clave se pueda compartir entre applets que sean

seleccionados durante una sesión CAD. Cuando se resetea la tarjeta, se borran los campos de los objetos transitorios `CLEAR_ON_RESET`. Un reset a la tarjeta puede estar causado por una señal de reset enviada a la circuitería de la tarjeta (reset en caliente) o apagando y encendiendo la fuente de alimentación.

Los objetos transitorios del tipo `CLEAR_ON_DESELECT` sirven para mantener datos que se deben preservar en el tiempo en que el applet esté seleccionado pero no entre selecciones del applet o reseteos de la tarjeta. Por ejemplo, la clave de sesión de un applet, necesita ser declarada del tipo `CLEAR_ON_DESELECT` para que cuando el applet sea deseleccionado, el JCRE borre automáticamente la clave de sesión. Esto es por una razón de seguridad. De este modo otro applet no podría descubrir los datos de la clave de sesión e intentar hacerse pasar por el applet anteriormente seleccionado, al que pertenece ese objeto clave.

Debido a que un reset a la tarjeta deselecciona implícitamente al applet actualmente seleccionado, los campos de los objetos `CLEAR_ON_DESELECT` también se borran para los mismos eventos especificados para los del tipo `CLEAR_ON_RESET`. En otras palabras, los objetos `CLEAR_ON_DESELECT` son también objetos `CLEAR_ON_RESET`. Además, los objetos transitorios `CLEAR_ON_DESELECT` tienen propiedades adicionales debido al applet firewall (se verá en la sección 4.7).

4.2.3.3 Creación de objetos transitorios

En la tecnología Java , los objetos transitorios se crean usando uno de los métodos de fábrica incluidos en la clase `JCSystem`, como se muestran a continuación:

Tabla 5: Métodos de la clase `JCSystem` para crear arrays transitorios

Métodos	Resultado de llamar al método.
`public static boolean[] makeTransientBooleanArray (short length, byte event)`	Crea un array transitorio de tipo `boolean`.
`public static byte[] makeTransientByteArray (short length, byte envent)`	Crea un array transitorio de tipo `byte`.
`public static short[] makeTransientShortArray (short length, byte envent)`	Crea un array transitorio de tipo `short`.
`public static Object[] makeTransientObjectArray (short length, byte envent)`	Crea un array transitorio de tipo `Object`.

El primer parámetro, `length`, especifica la longitud del array transitorio requerido en cada llamada al método. El segundo parámetro, `event`, indica que tipo de evento limpia el objeto. De este modo, se especifica el tipo de array transitorio, `CLEAR_ON_RESET` o `CLEAR_ON_ DESELECT`. Se usan dos constantes de la clase `JCSystem` para denotar el tipo de array transitorio:

```
//array transitorio del tipo CLEAR_ON_RESET
public static final byte CLEAR_ON_RESET

//array transitorio del tipo CLEAR_ON_DESELECT
public static final byte CLEAR_ON_DESELECT
```

El siguiente fragmento de código crea un array del tipo `CLEAR_ON_DESELECT`:

```
JCSystem.makeTransientByteArray (BUFFER_LENGTH,
JCSystem.CLEAR_ON_DESELECT);
```

4.2.3.4 Consulta de los objetos transitorios

Un applet quizás necesite acceder a un objeto que está creado por un applet diferente. La clase `JCSystem` provee un método práctico de consulta para que un applet pueda determinar si el objeto al que está accediendo es transitorio:

```
public static byte isTransient(Object theObject)
```

El método `isTransient` devuelve un tipo constante (`CLEAR_ON_RESET` o `CLEAR_ON_DESELECT`) o la constante `JCSystem.NOT_A_TRANSIENT_OBJECT` para indicar que ese objeto es nulo (`null`) o es un objeto persistente.

4.2.4 ACERCA DE LA CREACIÓN Y BORRADO DE OBJETOS

Debido a que la memoria de una smart card es escasa, si no hay suficiente memoria no volátil disponible cuando un applet intenta crear un objeto persistente usando el operador `new`, el JCRE arroja una `SystemException` con el código de la causa (similar al mensaje de detalle que contiene un objeto de excepción de Java) `JCSystem.NO_RESOURCE`.

Cuando un applet llama a uno de los métodos de creación de objetos transitorios y no hay suficiente espacio de memoria RAM disponible, el JCRE arroja una `SystemException` con el código de la causa `JCSystem.NO_TRANSIENT_SPACE`.

Una vez creados, los objetos persistentes y transitorios son alcanzables mientras sean referenciados desde la pila, los campos estáticos de una clase, los campos de otros objetos existentes o desde el JCRE. Cuando todas las referencias a un objeto desaparecen, el objeto se vuelve inalcanzable. Que el espacio que el objeto ocupa pueda ser reclamado, depende de si hay implementado un recolector de basura en la máquina virtual. La tecnología Java Card no requiere una implementación de JCRE que incluya un recolector de basura, debido a que no es factible hacerlo en los niveles bajos de las smart cards.

4.3 ATOMICIDAD Y TRANSACCIONES

Las smart cards están emergiendo como los dispositivos preferidos en aplicaciones como el almacenamiento de datos personales confidenciales y la provisión de servicios de autenticación en entornos móviles o distribuidos. Sin embargo, con las smart cards, hay un peligro de fallo en cualquier instante durante la ejecución de los applets. Los fallos pueden ocurrir debido a un error computacional, o por lo que más suele pasar, el usuario de la smart card quita accidentalmente la tarjeta del CAD provocando un corte en el suministro de energía a la CPU de la tarjeta y finalizando así la ejecución de cualquier applet. El riesgo de una ejecución incompleta presenta un reto para preservar la integridad de las operaciones o de los datos más delicados en una smart card.

El JCRE provee un mecanismo muy robusto para asegurar operaciones atómicas. Este mecanismo está soportado en dos niveles. El primero, la plataforma Java Card asegura que cualquier actualización de un solo campo de un objeto persistente o de un solo campo de una clase, es atómica. El segundo, la plataforma Java Card soporta un

modelo transaccional, en el que un applet puede agrupar un conjunto de cambios en una transacción. En este modelo, la atomicidad de todas las actualizaciones está asegurada.

4.3.1 ATOMICIDAD

En la plataforma Java Card, la atomicidad significa que en cualquier actualización de un solo campo de un objeto persistente (incluyendo un elemento de un array) o de un campo de una clase, se garantiza que la operación se completa satisfactoriamente o se restaura su valor original si ocurre un error durante la actualización. Por ejemplo, un campo de un objeto contiene actualmente el valor 1, y es actualizado al valor 2. La tarjeta se sale accidentalmente del CAD en el momento crítico en que la tarjeta está sobrescribiendo el campo. Cuando la tarjeta vuelve a tener suministro de energía, el campo no tiene un valor aleatorio, sino que se restaura a su valor anterior, 1.

El concepto de atomicidad se aplica a los contenidos del almacenamiento persistente. Define como el JCRE trata un elemento simple en caso de pérdida de energía u otro error durante la actualización de ese elemento. La característica de atomicidad del JCRE no se aplica a los arrays transitorios. La actualización de un elemento de un array transitorio no preserva el valor previo del elemento en caso de pérdida de energía. La próxima vez que la tarjeta se inserte en un CAD, los elementos de un array transitorio estarán puestos a sus valores por defecto (`cero`, `false` o `null`).

4.3.2 ACTUALIZACIONES DE BLOQUES DE DATOS EN UN ARRAY

La clase `javacard.framework.Util` provee un método, `arrayCopy`, que garantiza la atomicidad para las actualizaciones de múltiples elementos de dato en un array:

```
public static short arrayCopy (byte[] src, short srcOff, byte[]
dest, short desOff, short length)
```

El método `Util.arrayCopy` garantiza que todos los bytes están correctamente copiados o que el array de destino tiene restaurados los bytes a sus valores previos. Si el array de destino es transitorio, la característica de atomicidad no se mantiene.

Sin embargo, el método `arrayCopy` requiere escrituras extra en la EEPROM para dar soporte a la atomicidad, y por este motivo este proceso es lento. Un applet quizás no requiera atomicidad para las actualizaciones de arrays. Se provee el método `Util.arrayCopyNonAtomic` para este propósito:

```
public static short arrayCopyNonAtomic (byte[] src, short
srcOff, byte[] dest, shot desOff, short length)
```

El método `arrayCopyNonAtomic` no usa la facilidad de transacción durante la operación de copia aunque la transacción esté en proceso. Por eso, este método se debe usar solamente si los contenidos del array de destino pueden dejarse en un estado de modificación parcial cuando se corta la energía en mitad de una operación de copia. Un método similar, `Util.arrayFillNonAtomic`, rellena de forma no atómica los elementos de un array de bytes con un valor especificado:

```
public static short arrayFillNonAtomic (byte[] bArray, short
bOff, short bLen, byte bValue)
```

4.3.3 TRANSACCIONES

La atomicidad garantiza la modificación atómica de un solo elemento de dato. Sin embargo, un applet puede necesitar la actualización automática de varios campos diferentes y en varios objetos diferentes. Por ejemplo, una transacción de crédito o de débito quizás requiera un applet de monedero (purse) para incrementar el número de transacción, actualizar el balance del monedero, y escribir en un registro de transacciones, todo como una unidad de trabajo atómica.

Nociones usadas en las transacciones en bases de datos, como begin, commit, y rollback aseguran que se completen o no se completen las actualizaciones de múltiples valores. La tecnología Java Card soporta un modelo de transacción similar (con commit y rollback) para garantizar que las operaciones complejas se puedan efectuar atómicamente. Esto implica que todas las operaciones se han completado satisfactoriamente o que los resultados parciales no han tenido efecto. Los mecanismos de transacción protegen los datos frente a eventos como la pérdida de energía en mitad de una transacción o los errores de programación, que quizás causen corrupción en los datos.

4.3.3.1 Declaración de la transacción

Una transacción comienza por la invocación del método `JCSystem.beginTransaction` y termina con la llamada al método `JCSystem.commitTransaction`:

```
//comienzo de una transacción
JCSystem.beginTransaction();
//todas las modificaciones de un conjunto de
//actualizaciones de datos persistentes son temporales
//hasta que la transacción se comprometa o se declare.
...
//declaración de la transacción
JCSystem.commitTransaction();
```

Los cambios en una transacción son condicionales, los campos o elementos de arrays parecen estar actualizados. La lectura de los campos o los elementos de arrays ofrece los últimos valores, pero las actualizaciones no se comprometen hasta que se llama al método `JCSystem.commitTransaction`.

4.3.3.2 Interrupción de la transacción

El applet o el JCRE pueden interrumpir las transacciones. Si un applet encuentra un problema interno, puede cancelar la transacción expresamente llamando al método `JCSystem.abortTransaction`. La interrupción de una transacción hace que el JCRE deshaga cualquier cambio hecho durante la transacción y restaure los campos actualizados o los elementos de arrays a sus valores anteriores. Cuando se invoca al método `abortTransaction` debe haber una transacción en progreso; si no, el JCRE arroja una `TransactionException`.

Cuando el JCRE recupera el control de la programación después de que el applet anterior tuviera una transacción todavía en progreso, (es decir, cuando el applet no declaró o abortó una transacción en curso) el JCRE llama automáticamente al método `abortTransaction`. De forma similar, el JCRE aborta una transacción si la transacción arroja una excepción y el applet no trata dicha excepción.

Si se corta la energía u ocurre un error durante la transacción, el JCRE invoca una facilidad de rollback, interna del JCRE, la próxima vez que se le suministre energía a la tarjeta. Así se restauran los valores de los datos involucrados en la transacción.

En cualquier caso, los objetos transitorios y los persistentes creados durante una transacción que falla (por corte de energía, reseteo de la tarjeta, error de cálculo, o por una interrupción del programa), se borran y el JCRE libera la memoria.

4.3.3.3 Transacciones anidadas

A diferencia de las transacciones de las bases de datos, las transacciones en la plataforma Java Card no pueden ser anidadas. Solo puede haber una transacción en progreso en cada instante. Este requisito se debe a los limitados recursos de computación de las smart cards.

Si se llama a `JCSystem.beginTransaction` mientras una transacción está todavía en progreso, el JCRE lanza una `TransactionException`. Un applet puede descubrir si una transacción está en progreso invocando al método `JCSystem.transactionDepth`. El método devuelve 1 si una transacción está en progreso y un 0 en caso contrario.

4.3.3.4 Capacidad de declaración

Para soportar el rollback (vuelta atrás) de transacciones no declaradas o no comprometidas, el JCRE mantiene un "commit buffer" donde se guardan los contenidos originales de los campos actualizados hasta que la transacción se comprometa. Si ocurriera un fallo antes de que la transacción se complete, los campos participantes en la transacción se restaurarían a sus contenidos originales a partir del "commit buffer".

El tamaño del "commit buffer" varía de una implementación a otra, dependiendo de la memoria disponible en la memoria de la tarjeta. En general, el "commit buffer" alojado en una implementación de JCRE es lo suficientemente grande como para satisfacer las necesidades de la mayoría de los applets (un applet suele acumular decenas de bytes durante una transacción). Sin embargo, debido a que los recursos de las smart cards son limitados, es importante que solo las actualizaciones de una unidad lógica de operaciones se incluyan en una transacción. El poner demasiadas cosas en una transacción quizás no sea posible.

Antes de intentar una transacción, un applet puede comparar el tamaño disponible del "commit buffer" con el tamaño de los datos requeridos en una actualización atómica. La clase `JCSystem` provee dos métodos para ayudar a los applets a determinar cuanta capacidad está disponible en el buffer de las implementaciones de Java Card.

- `JCSystem.getMaxCommitCapacity()` devuelve el número total de bytes del "commit buffer".
- `JCSystem.getUnusedCommitCapacity()` devuelve el número de bytes no usados que quedan en el "commit buffer".

Además, para guardar los contenidos de los campos modificados durante una transacción, el "commit buffer" mantiene bytes adicionales, tales como la localización de los campos. La cantidad de estos bytes adicionales depende del número de campos que están siendo modificados y de la implementación del sistema de transacciones. La capacidad devuelta por los dos métodos es el número total de bytes de datos persistentes, incluyendo los datos adicionales, que pueden ser modificados durante una transacción.

Si se excede la capacidad durante una transacción, el JCRE arroja una `TransactionException`. Aún así, la transacción está todavía en progreso a menos que el applet o el JCRE la interrumpa explícitamente.

4.3.3.5 Excepción `TransactionException`

El JCRE arroja una `TransactionException` si se detectan ciertas clases de problemas (como las transacciones anidadas o como el overflow del "commit buffer") en una transacción.

`TransactionException` es una subclase de `RunTimeException`. Provee un código de causa para indicar la causa de la excepción. Las excepciones de Java Card y los códigos de causa se explican más tarde. Los siguientes son códigos de causa definidos en la clase `TransactionException`:

- `IN_PROGRESS`: se llamó a `beginTransaction` mientras una transacción estaba todavía en progreso.
- `NOT_IN_PROGRESS`: se llamó a `commitTransaction` o `abortTransaction` mientras una transacción no estaba en progreso.
- `BUFFER_FULL`: durante una transacción, se intentó una actualización que ha causado el overflow del "commit buffer".
- `INTERNAL_FAILURE`: ocurrió un problema interno en el sistema de transacción.

Si el applet no captura una `TransactionException`, la captura el JCRE. En último caso, el JCRE aborta la transacción automáticamente.

4.3.3.6 Variables locales y objetos transitorios durante una transacción

Debe tenerse en cuenta que solo las actualizaciones a objetos persistentes participan en una transacción. Las actualizaciones de objetos transitorios y variables locales (incluyendo los parámetros del método) se realizan sin hacer caso de si estaban o no dentro de una transacción. Las variables locales se crean en la pila de Java Card que reside en RAM.

El siguiente fragmento de código demuestra tres operaciones de copia que involucran a un array transitorio `key_buffer`. Cuando la transacción se interrumpe, ni las operaciones de copia del array ni la actualización de los elementos de `key_buffer` en el bucle `for`, están protegidos por la transacción. De forma similar, la variable local `a_local` retiene el nuevo valor 1.

```
byte[] key_buffer = JCSystem.makeTransientByteArray (KEY_LENGTH,
JCSystem, CLEAR_ON_RESET);
JCSystem.beginTransaction();
Util.arrayCopy(src, src_off, key_buffer, 0, KEY_LENGTH);
Util.arrayCopyNonAtomic(src,     src_off,     key_buffer,     0,
KEY_LENGTH);
for (byte i=0; i < KEY_LENGTH; i++) key_buffer[i] = 0;
byte a_local = 1;
JCSystem.abortTransaction();
```

Debido a que las variables locales o los elementos del array transitorio no participan en una transacción, la creación de un objeto y la asignación del objeto a una variable local o al elemento de un array transitorio necesita ser considerada cuidadosamente. Aquí se presenta un código de ejemplo:

```
JCSystem.beginTransaction();
//ref_1 es una instancia (objeto) de campos
ref_1 = JCSystem.makeTransientObjectArray (LENGTH,
JCSystem.CLEAR_ON_DESELECT);
//ref_2 es una variable local
ref_2 = new SomeClass();
//comprobación de estado
if (!condition) JCSystem.abortTransaction();
else JCSystem.commitTransaction();
return ref_2;
```

En el ejemplo, la instancia `ref_1` guarda una referencia a un objeto transitorio, y la variable local `ref_2` guarda una referencia a un objeto persistente. Como se describió previamente, si se interrumpe una transacción, los objetos persistentes y transitorios creados durante una transacción, son automáticamente destruidos, Esto no tiene efecto secundario sobre la instancia `ref_1`, porque su contenido se restaura al valor original si la transacción no se completa normalmente. Sin embargo, ocurre un problema potencial en la siguiente línea, cuando un objeto nuevo se asigna a una variable local. En un fallo de la transacción, el JCRE borra el objeto, sin embargo, `ref_2` todavía apunta a la localización donde el objeto ya no existe. La situación empeora si `ref_2` se usa después como valor de retorno del método. En este caso, el llamante recibe un puntero colgado.

Para evitar generar un puntero colgado, que compromete la seguridad del lenguaje Java, el JCRE asegura que aquellas referencias a objetos creados durante una transacción interrumpida se ponen a `null`. En el ejemplo, si se invoca al método `abortTransaction`, la variable `ref_2` se pone a `null`. Esta solución quizás no sea ideal, pero evita la violación de seguridad y además minimiza la carga del sistema.

Este ejemplo no es aplicable a la mayoría de los applets, porque crear objetos en un método no es recomendable. Cuando sea posible, un applet debería alojar todos los objetos que necesite durante la inicialización del applet. Sin embargo, un implementador de un instalador de Java Card que quizás necesite tratar con una

considerable creación de objetos en una transacción, debería evitar el escenario descrito en el código.

4.4 EXCEPCIONES EN JAVA CARD Y MANEJO DE EXCEPCIONES

Una excepción es un evento que rompe el flujo normal de instrucciones durante la ejecución de un programa. Las excepciones son importantes en el lenguaje Java porque proveen una forma elegante de manejar los errores de un programa.

La plataforma Java Card soporta todas las construcciones para excepciones del lenguaje programación Java. Un applet de Java Card puede usar las palabras clave `throw`, `try`, `catch` o `finally`, y funcionan de la misma manera que en la plataforma Java.

Las clases del JCRE y la máquina virtual de Java Card lanzan excepciones cuando se detectan problemas en tiempo de ejecución o cuando los applets las lanzan por programación. A pesar de que la plataforma Java Card tiene soporte completo para las excepciones al estilo de Java, difieren en su uso debido al entorno restringido de las smart cards.

4.4.1 EXCEPCIONES EN EL PAQUETE `JAVA.LANG`

En general, la plataforma Java Card no soporta todos los tipos de excepciones que se encuentran en los paquetes del núcleo de la tecnología Java, debido a que muchos de ellos no son aplicables al contexto de las smart cards. Por ejemplo, en la plataforma Java Card no se soporta la programación con hilos, y por lo tanto, no se soporta ninguna de las excepciones relacionadas con los hilos.

Sin embargo, el paquete `java.lang` de Java Card soporta algunas clases de excepciones derivadas de la versión Java de ese paquete. De todas las clases de excepciones soportadas, solo se proveen el método `equals` (heredado de la clase raíz `Object`) y un constructor sin parámetros.

En la siguiente tabla se listan todas las clases de excepciones del paquete `java.lang` de la plataforma Java Card:

Tabla 6: Clases de excepción del paquete `java.lang`

`Throwable`	`Exception`	`RuntimeException`
`ArithmeticException`	`ArrayStoreException`	`ArrayIndexOutOfBounds Exception`
`ClassCastException`	`NullPointerException`	`IndexOutOfBounds Exception`
`SecurityException`	`NegativeArraySize Exception`	

La clase `Throwable` define una clase común para todas las clases de excepción de la plataforma Java Card. Esta clase también asegura que las excepciones de Java Card tienen la misma semántica que las excepciones equivalentes de la plataforma Java. Por ejemplo, los applets solo pueden lanzar y capturar objetos que deriven de la clase `Throwable`.

La clase `Exception` extiende la clase `Throwable`. Como en la plataforma Java, es la clase raíz en la plataforma Java Card para todas las excepciones con comprobación. La clase `RuntimeException` deriva de la clase `Exception`, y es la clase raíz para todas las excepciones sin comprobación en la plataforma Java Card. El concepto de excepciones con comprobación y sin comprobación están definidas en la especificación del lenguaje Java. Sus definiciones se dan en la siguiente sección.

El resto de clases de la tabla anterior son excepciones sin comprobación. Las clases de excepciones del paquete `java.lang` proveen un soporte fundamental del lenguaje para el soporte de excepciones de Java. La máquina virtual de Java Card las lanza cuando ocurre un error debido a una violación del lenguaje Java.

4.4.2 EXCEPCIONES DE JAVA CARD

La plataforma Java Card provee una jerarquía de herencia para las excepciones con comprobación y sin comprobación, como se muestra en la Figura 15:

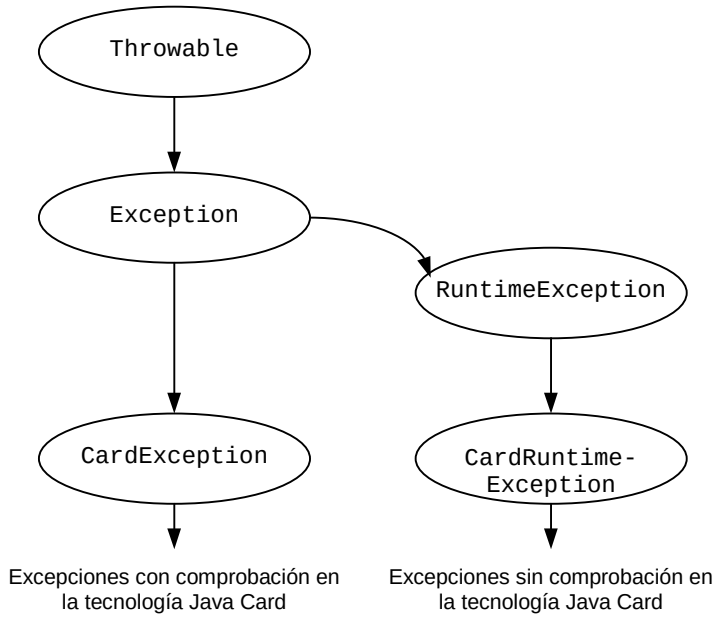


Figura 15: Jerarquía de excepciones de Java Card

Las excepciones con comprobación son subclases de la clase `Exception` y se deben capturar en los métodos que las arrojan o declararlas en una cláusula `throws` en la cabecera del método. El compilador de Java fuerza este requisito. Todas las clases de excepciones de Java Card extienden la clase `CardException`, que deriva de la clase `Exception`.

El applet debe capturar todas las excepciones con comprobación, por dos razones. La primera es que las excepciones con comprobación indican un error de programación en un applet y por ello el applet debería corregirlo. La segunda es que las excepciones con comprobación son una parte importante de la interfaz de un método. Debido a que los métodos no pertenecientes a las API's especifican una excepción con comprobación en la cláusula `throws`, el compilador de Java emite un error si un applet no captura la excepción con comprobación.

Las excepciones sin comprobación, llamadas a menudo excepciones en tiempo de ejecución, son subclases de la clase `RuntimeException` y no se necesitan capturar en un programa ni declarar en una cláusula `throws`. Las excepciones sin comprobación suelen indicar problemas inesperados en tiempo de ejecución, errores de programación o estados erróneos de procesamiento de APDU's. Los niveles más externos del JCRE son los encargados de capturar tales excepciones. Todas las excepciones sin comprobación de la plataforma Java Card deberían extender la clase `CardRuntimeException`, que deriva de la clase `RuntimeException`.

Entonces, para que se necesitan las clases `CardException` y `CardRuntimeException`? porque habilitan un mecanismo de ahorro de recursos que permite que un objeto de excepción se pueda reutilizar múltiples veces, como se explica en las siguientes secciones.

4.4.2.1 Código de causa de las expeciones de Java Card

Las clases de excepciones de Java suministran una cadena con un mensaje que indica un error específico. La plataforma Java Card no soporta la clase `String`, por ello no se pueden proveer mensajes de cadenas en las excepciones. Como camino alternativo para adjuntar información extra a la excepción, las clases de excepciones de Java Card suministran un código numérico de causa. El código de causa se usa para describir detalles opcionales relacionados con el lanzamiento de la excepción. El tipo del código de causa es `short`.

El código de causa se define como un campo de las clases `CardException` y `CardRuntimeException` y por este motivo sus subclases lo heredan. Además, ambas clases definen dos métodos públicos (`getReason` y `setReason`) para recuperar y cambiar un código de causa.

4.4.2.2 Lanzando una excepcion en la plataforma Java Card

Para lanzar una excepción en el sistema Java, un applet crea una instancia de una clase de excepción. El código que permite esto se muestra a continuación:

```
throw new MyException("un mensaje específico de error");
```

Por supuesto, se podría crear un objeto de excepción nuevo cada vez que se lanza una excepción en la plataforma Java Card. Sin embargo, la economía de espacio siempre es muy importante en una smart card. Si un applet crea un objeto cada vez que se lanza una excepción, con el tiempo, el applet irá acumulando una gran cantidad de instancias de excepciones en memoria EEPROM y que además no se usarán más.

Para optimizar el uso de la memoria, todos los objetos de excepciones se deberían crear en el instante de la inicialización y salvar sus referencias permanentemente. Cuando ocurre un evento de excepción, antes de crear un nuevo objeto de excepción, un applet debería hacer lo siguiente:

- Recuperar y reutilizar la referencia del objeto de excepción deseado.
- Rellenar el código de causa en el objeto.

- Lanzar el objeto.

Para dar soporte a los objetos de excepción reutilizables, el JCRE crea una instancia de cada tipo de excepción de las API's de Java Card. Las clases `CardException` y `CardRuntimeException` y cada una de sus subclases, proveen un método estático `throwIt` para que los applets reutilicen la instancia de la excepción:

```
public static void throwIt (short reason)
```

El método `throwIt` lanza la instancia de la excepción, creada por el JCRE, cada vez que se invoca. Además, el applet especificará un código de causa al método `throwIt`. Por ejemplo, para rechazar una APDU de comando, un applet puede lanzar una `ISOException` e indicar que el código de causa es "comando no permitido":

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

Un applet puede crear sus propios objetos de excepciones. Durante la inicialización, el applet realiza una instancia del objeto de excepción y salva la referencia en un campo persistente. Más tarde, el applet reutiliza la instancia para todas las veces que necesite lanzar esa excepción.

4.4.2.3 Excepción `ISOException`

La `ISOException` es una excepción de las API's de Java Card sin comprobación especial. Se lanza durante el tiempo de ejecución para indicar un estado de aviso o de error de procesamiento de la tarjeta. La `ISOException` encapsula el código de causa en una palabra de estado (SW) de una respuesta ISO 7816.

La `ISOException` permite que un applet trate los errores eficientemente. Cuando se procesa un comando de forma satisfactoria, el método se comporta de forma normal. Pero si ocurre un error, el método lanza una `ISOException` con la palabra de estado apropiada.

Típicamente, un applet no maneja una `ISOException`. De forma eventual, el JCRE captura la `ISOException`. Entonces, devuelve a la aplicación que se encuentra en el host, el código de causa que contiene el objeto de excepción mediante un palabra de estado ISO. Éste es el porqué de que la clase de excepciones lleve la palabra ISO en su nombre.

La palabra de estado ISO es parte de un protocolo APDU. Es una forma de que una smart card devuelva el estado de procesamiento de una APDU de comando a la aplicación del host. La plataforma Java Card provee una interfaz `javacard.framework.ISO7816` que define las constantes de palabras de estado más usadas, relacionadas con `ISO 7816-3` e `ISO 7816-4`. Un applet puede definir sus propias palabras de estado y puede usar la `ISOException` para comunicárselas a la aplicación del host.

4.4.2.4 Excepción `UserException`

Cuando un applet encuentra un error de programación que el applet necesita corregir, lanza una `UserException`. A diferencia de la `ISOException`, una `UserException` es una excepción comprobable derivada de `CardException` y por lo tanto el applet debe manejarla. Si un applet necesita crear tipos adicionales de excepciones, puede hacerlo creando clases derivadas de `UserException`.

4.5 APPLETS DE JAVA CARD

4.5.1 INTRODUCCIÓN A LOS APPLETS

Un applet de Java Card es una aplicación de smart card, escrita en el lenguaje de programación Java y conforme a una serie de convenciones para que pueda ejecutarse en el entorno de ejecución de Java Card (JCRE). Un applet ejecutándose en el JCRE es una instancia de la clase del applet extendida de `javacard.framework.Applet`. Como con otros objetos persistentes, un applet creado en la tarjeta vive a lo largo de toda la vida de la tarjeta (algunas smart cards de Java también soportan el borrado de applets). La plataforma Java Card soporta un entorno multiaplicación. Cada instancia de un applet se identifica unívocamente mediante un AID (como ya se ha visto).

4.5.1.1 Instalación y ejecución de applets

Después de que el (los) paquete(s) que define(n) un applet se hayan cargado convenientemente en una smart card de Java y se haya linkado con otros paquetes de la tarjeta, la vida de un applet comienza cuando una instancia del applet se crea y registra en el JCRE. El JCRE es un entorno en el que solo se permite un hilo. Esto significa que solo se ejecuta un applet a la vez. Cuando un applet acaba de instalarse, se encuentra en un estado inactivo. El applet se vuelve activo cuando una aplicación de un host lo selecciona explícitamente.

Los applets, como en cualquier aplicación de smart card, son aplicaciones reactivas. Un vez seleccionado, un applet típico espera a que una aplicación que se ejecuta en el host le envíe un comando. Entonces, el applet ejecuta el comando y devuelve una respuesta al host.

Este diálogo de comando-respuesta continúa hasta que un nuevo applet se seleccione o la tarjeta se quite del dispositivo de aceptación de tarjetas. El applet permanece inactivo hasta que sea seleccionado la próxima vez. Los estados de ejecución se ilustran en la siguiente figura:

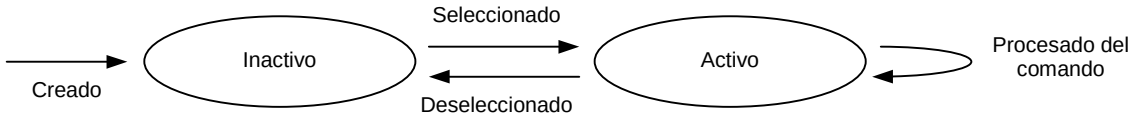


Figura 16: Estados de ejecución de un applet

4.5.1.2 Comunicación con el applet

La comunicación entre un applet y una aplicación de un host, se consigue a través del intercambio de APDU's como se muestra en la siguiente figura:

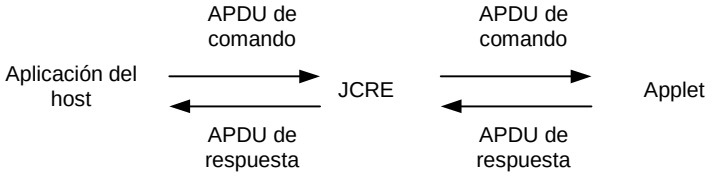


Figura 17: Comunicación entre un applet y una aplicación del host

Una APDU contiene un mensaje de comando o de respuesta. Las aplicaciones del host envían comandos al applet y el applet devuelve respuestas.

Cuando la aplicación del host quiere seleccionar un applet, envía una APDU que especifica el comando SELECT y el AID del applet que se requiere. El JCRE busca un applet en su tabla interna cuyo AID coincida con el especificado en el comando. Si se encuentra una coincidencia, el JCRE selecciona el applet para que se ejecute. Todas las APDU's posteriores (incluyendo la APDU SELECT) se envían al applet actual hasta que se selecciona un nuevo applet.

4.5.2 CLASE `JAVACARD.FRAMEWORK.APPLET`

Cada applet se implementa creando una subclase de la clase `javacard.framwork.Applet`. El JCRE invoca los métodos `install`, `select`, `process` o `deselect` (que están definidos en la clase base `Applet`) cuando quiere instalar, seleccionar, o deseleccionar el applet o preguntarle al applet acerca del procesamiento de una APDU de comando. Los métodos de la siguiente tabla están listados en el orden en que el JCRE los invoca durante la creación y ejecución del applet:

Tabla 7: Métodos de la clase `javacard.framework.Applet`

`public static void`	`install (byte[] bArray, short bOffset, byte bLenght)` El JCRE llama a este método estático para crear una instancia de la subclase `Applet`.
`protected final void`	`register ()` El applet usa este método para registrar esta instancia del applet en el JCRE y asignar a la instancia del applet, el AID por defecto que contiene el fichero CAP.
`protected final void`	`register (byte[] bArray, short bOffset, byte bLength)` El applet usa este método para registrar esta instancia del applet en el JCRE y asignar a la instancia del applet, el AID especificado en el array `bArray`.

`public boolean`	`select ()` El JCRE llama a este método para informar al applet de que ha sido seleccionado.
`public abstract void`	`process (APDU apdu)` El JCRE llama a este método para ordenar al applet el procesado de una APDU de comando entrante.
`public void`	`deselect ()` El JCRE llama a este método para informar al applet actualmente seleccionado que otro (o el mismo) applet será seleccionado.

El JCRE llama al método `install` para crear una instancia el applet. La instancia del applet se registra en el JCRE, usando uno de los dos métodos `register`.

Cuando se recibe a una APDU SELECT, el JCRE comprueba primero si hay un applet seleccionado. Si es así, el JCRE deselecciona el applet actual invocando al método `deselect`. En el método `deselect`, el applet lleva a cabo cualquier trabajo de limpieza o de contabilidad antes de que el applet se vuelva inactivo. Entonces el JCRE selecciona el nuevo applet invocando al método `select`. El nuevo applet seleccionado lleva a cabo cualquier inicialización necesaria en el método `select`.

Después de una selección exitosa, al applet activo se le entrega cada APDU (incluyendo la APDU SELECT) mediante una llamada al método `process`. El método `process` es un método esencial en la clase del applet. Procesa las APDU's de comando y así se consiguen las funcionalidades del applet.

Los métodos `install`, `select`, `deselect` y `process` son métodos de entrada al applet. El JCRE los invoca según el estado de creación o ejecución en el que se encuentre el applet. La clase base `Applet` solo provee el comportamiento por defecto de estos métodos. Un applet necesitará sobrescribir alguno o todos estos métodos para implementar sus funciones. Los detalles de cada uno de estos métodos se especifican en apartados posteriores.

4.5.3 EL MÉTODO `INSTALL`

El JCRE suele llamar al método `install` en el último paso durante la instalación del applet para crear una instancia de dicho applet. El método `install` es similar al método `main` en una aplicación Java. Los argumentos del método `install` llevan los parámetros de instalación el applet. Son análogos a los argumentos suministrados al método `main` en la línea de comandos.

El método `install` crea una instancia del applet usando el operador `new` seguido de una llamada al constructor del applet. En el constructor, un applet suele llevar a cabo los siguientes objetivos:

- Crear los objetos que el applet necesite durante su tiempo de vida.
- Inicializar objetos y las variables internas del applet.
- Registrar la instancia del applet en el JCRE llamando a uno de los dos métodos `register` definidos en la clase base `Applet`.

El registro de la instancia del applet marca el comienzo del tiempo de vida del applet. Un applet se debe registrar en el JCRE para que el JCRE pueda seleccionarlo y

ponerlo en ejecución. El siguiente código muestra un ejemplo de la creación de un applet "cartera" (wallet) usando el constructor por defecto:

```
public class WalletApp extends Applet{
      private Log transaction_log;
      private byte[] wallet_id;
      private byte wallet_balance;
      public static void install (byte[] bArray, short
      bOffset, byte bLength) {
            new WalletApp();
      }
      private WalletApp() {
            //crea un registro de transacciones
            //identificadas con un número específico
            transaction_log = new Log(TRAN_RECORD_NUM);
            //crea un array de bytes para guardar el ID
            //de la cartera
            wallet_id = new byte[ID_LENGTH];
            //inicializa el balance de la cartera
            wallet_balance = INITIAL_BALANCE;
            //registra la instancia del applet en el JCRE
            register()
      }
}
```

Alternativamente, un applet puede definir un constructor que tome los parámetros de instalación:

```
public walletApp(byte[] bArray, short bOffset, byte bLength)
{...}
```

Los parámetros de instalación proveen datos adicionales para inicializar y personalizar el applet. El procesamiento de los parámetros de instalación se explica más adelante.

Si el resultado del método `install` ha sido exitoso, el applet está preparado para ser seleccionado y procesar las APDU's de comando. Solo se puede crear y registrar con éxito una instancia de un applet a partir de una sola invocación al método `install`. Si el JCRE quiere crear múltiples instancias del mismo applet, cada instancia se crea con cada invocación del método `install`.

Si ocurre un fallo durante el método `install` y antes de la invocación del método `register`, cuando el JCRE recupere el control realizará las acciones de limpieza necesarias para recuperar los recursos de la tarjeta. El JCRE borra la instancia del applet además de los objetos creados por el método `install` y recupera su estado previo. No es necesario establecer una transacción en el método `install`, ya que el JCRE asegura que el método `install` es transaccional. El registro del applet significa el final exitoso de la transacción. Por lo tanto, es importante registrar el applet como el último paso durante la creación del applet. Si ocurre cualquier error después del método `register`, el applet permanecerá registrado, pero quizás quede inutilizado.

Se debe prestar atención en que el método `install` de la clase base `Applet` es simplemente un prototipo. Por lo que un applet debe definir un método `install` que tenga el mismo prototipo.

4.5.3.1 Creación de objetos con el constructor del applet

Aunque los objetos y arrays se pueden crear en cualquier punto de la ejecución de un applet, cuando sea posible, es recomendable que tales creaciones ocurran solo durante la inicialización del applet. Cualquier objeto que quizás se requiera durante la ejecución del applet, debería ser preasignado en el constructor para asegurar que el applet nunca fallará debido a la falta de memoria.

El constructor se invoca dentro del método `install`. Así, si el JCRE detecta escasez de recursos y no puede asignar espacio en memoria para el applet durante la creación de objetos o durante algún otro proceso de asignación, el JCRE borrará el applet y recuperará todo el espacio en memoria. Así no se dejará ningún applet creado parcialmente, en un estado de no ejecución.

No obstante, un applet no debería crear más objetos de los necesarios, ya que la memoria ocupada por los objetos no usados no se puede reutilizar o compartir por otros applets o por el JCRE.

4.5.3.2 Registro de la instancia del applet en el JCRE

Para registrar un applet en el JCRE, se usa uno de los dos métodos `register` provistos en la clase base `Applet`.

```
protected final void register ()
protected final void register (byte[] bArray, short bOffset,
byte bLength)
```

El método `register` tiene dos funciones. La primera es que guarda una referencia a la instancia del applet en el JCRE. La segunda es que asigna un AID a la instancia del applet (se recuerda que cada instancia del applet está unívocamente identificada por un AID). El fichero CAP que define las clases de los applets contiene un AID por defecto. Sin embargo, un applet podría elegir tener un AID distinto del AID por defecto. El AID por defecto se puede suministrar en los parámetros de la instalación.

El primer método `register` (el que no tiene parámetros) registra al applet en el JCRE usando el AID por defecto que se encuentra en el fichero CAP. El segundo método `register` (que tiene argumentos) registra la instancia del applet en el JCRE usando el AID especificado en el argumento `bArray`. El argumento `bOffset` especifica el comienzo del offset en `bArray`, y `bLength` especifica la longitud del AID en bytes.

4.5.3.3 Proceso los parámetros de instalación

Normalmente, durante la instalación de un applet, los parámetros de instalación se envían a la tarjeta junto a los ficheros CAP que definen al applet. Entonces, el JCRE provee los parámetros de instalación al applet a través de los argumentos del método `install`. El método `install` acepta tres argumentos:

- `byte[] bArray`: array que contiene los parámetros de instalación.
- `short bOffset`: comienzo del offset en `bArray`.

- `byte bLength`: longitud, en bytes, de los parámetros contenidos `bArray`.

El contenido y formato de los parámetros de instalación están definidos por los diseñadores de applets o por los proveedores de tarjetas. Los parámetros de configuración se pueden usar para especificar el tamaño de un fichero interno, un array, etc. En este sentido, el applet puede asignar la memoria adecuada para soportar el procesamiento anticipado y así evitar el derroche de memoria. Por ejemplo, los valores de inicialización del applet pueden especificar el balance inicial, el ID del titular de la tarjeta y el número de cuenta en una cartera (monedero) electrónica. Otro uso común de los parámetros de instalación es suministrar un AID además del dado por defecto en el fichero CAP. Por ejemplo, se supone que se necesitan dos instancias del applet wallet (cartera): una para uso personal y otra para negocios. En tal caso, el JCRE debe invocar dos veces al método `install`. Cada vez, se crea una instancia del applet wallet con un AID único.

Ahora, supóngase que el diseñador del applet wallet especifica que el array de bytes de los parámetros de instalación consiste en los siguientes campos:

- Un valor de un byte que especifica el número de grabaciones en el registro de transacciones.
- Un array de 4 bytes que especifica la ID de la cartera.
- Un byte que contiene el balance inicial de la tarjeta.
- Un byte que especifica el tamaño del siguiente subarray.
- Un array de tamaño variable de bytes que especifica el AID para esa instancia del applet. Si el array está vacío, el applet usa el AID por defecto del fichero CAP.

Para crear una applet wallet para uso personal con el AID por defecto, los parámetros por defecto podrían ser los bytes [0x10, 0x1,0x2, 0x3, 0x4, 0x32, 0], los cuales serían interpretados por el applet de la siguiente manera:

- número de transacciones en el registro de transacciones = 0x10 = 16
- ID de la cartera = [0x1, 0x2, 0x3, 0x4]
- balance inicial = 0x32 =50
- AID = 0, lo que indica que toma el AID por defecto que contiene el fichero CAP.

Para crear una instancia de la cartera (para controlar los gastos por negocios) con un AID distinto al AID por defecto, los parámetros de instalación podrían ser los bytes [0x10, 0x4, 0x3, ‘x2, 0x1, 0x64, 0xF, ‘B’, ‘A’, ‘N’, ‘K’, ‘_’, ‘w’, ‘a’, ‘l’, ‘l’, ‘e’, ‘t’, ‘_’, ‘B’, ‘T’, ‘S’], los cuales serían interpretados por el applet como:

- número de transacción en el registro de transacciones = 0x10 = 16
- ID de la cartera = [0x4, 0x3, 0x2, 0x1]
- balance inicial = 0x64 = 100

- AID = [‘B’, ‘A’, ‘N’, ‘K’, ‘_’, ‘w’, ‘a’, ‘l’, ‘l’, ‘e’, ‘t’, ‘_’, ‘B’, ‘T’, ‘S’]

El siguiente código demuestra como el applet wallet procesa los parámetros de instalación en el constructor:

```
private WalletApp(byte[] bArray, short bOffset, byte
bLength) {
      //crea un registro de transacciones y especifica el
      //número máximo de grabaciones en el registro
      max_record_num = bArray[bOffset]
      transaction_log = new Log(bArray[bOffset++]);
      //ajusta la ID de la cartera
      wallet_id = new byte[ID_LENGTH];
      Util.arrayCopy(bArray, bOffset, wallet_id, (byte)0,
      ID_LENGTH]);
      //avanza bOffset en un cantidad ID_LENGTH de bytes
      bOffset += ID_LENGTH;
      //inicializa el balance de la cartera
      wallet_balance = bArray[bOffset++];
      //comprueba el AID
      byte AID_len = bArray[bOffset++];
      if (AID_len==0) {
            //registra la instancia del applet en el JCRE
            //usando el AID por defecto
            this.register();
      else {
            //registra la instancia del applet en el JCRE
            //usando el AID especificado en los parámetros
            //de instalación. Los bytes del AID en bArray
            //empiezan desde el índice bOffset y consisten
            //en un número AID_LEN de bytes.
            this.register(bArray, bOffset, AID_len);
      }
}
```

El contenido de `bArray` no pertenece al applet. Por motivos de seguridad, el JCRE limpia el array cuando se regresa del método `install`. Si el applet desea preservar cualquiera de esos datos, debería copiarlos dentro de su propio objeto. En el ejemplo, los bytes del ID de la cartera (contenidos en `bArray`) se copian en el campo `wallet_id`.

La plataforma Java Card soporta parámetros de instalación de hasta 32 bytes. De ese modo, el máximo valor de `bLength` es 32. Se verá más adelante que el JCRE emplea un buffer para transmitir APDU’s. El mínimo tamaño del buffer APDU es 37, incluyendo 5 bytes de cabecera y 32 de datos. El número 32 se elige como el máximo tamaño de los parámetros de instalación para que se puedan transportar en el buffer con una APDU de E/S.

4.5.3.4 Inicialización tardía de un applet

Después del `return` del método `install`, quizás los applets sencillos estén completamente listos para funcionar normalmente. Quizás los applets más complejos necesiten información adicional de personalización antes de que estén listos para ejecutarse de forma normal. Tal información podría no estar disponible en el momento

de la creación del applet o podría exceder la capacidad de los parámetros de instalación (32 bytes). En este caso, se podría requerir una planificación separada para permitirle al applet completar la personalización en el método `process`. En tal planificación, el applet necesita ajustar variables de estado y es el responsable de mantenerse al tanto de esas transiciones de estado. Para recibir información personalizada, el applet intercambia APDU's con el host.

4.5.4 EL MÉTODO SELECT

Un applet permanece en un estado suspendido hasta que se selecciona explícitamente. La selección del applet ocurre cuando el JCRE recibe una APDU SELECT cuyos datos coincidan con el AID del applet. El JCRE informa al applet de su selección invocando a su método `select`.

En el método `select`, el applet puede comprobar si sus condiciones para la selección se han cumplido, y si es así, puede ajustar sus variables y estados internos para manejar las siguientes APDU's. El applet devuelve `true` desde una llamada al método `select` si está preparado para aceptar las APDU's entrantes a través de su método `process`, o puede rehusar a ser seleccionado, devolviendo `false` o lanzando una excepción.

Si la selección falla, el JCRE devuelve la palabra de estado 0x6999 al host. Si el método `select` devuelve `true`, la APDU SELECT de comando se suministra a la llamada posterior de su método `process`, para que el applet pueda responder al host con información relacionada con el applet. Por ejemplo, el applet wallet quizás devuelva la identificación del proveedor, la información de conversión actual u otros parámetros. El host quizás necesite esta información para empezar las transacciones de débito o de crédito. Los diseñadores o proveedores son libres de definir el contenido y el formato de los datos de respuesta.

El método `select` de la clase base `Applet` simplemente devuelve `true`. Un applet puede sobrescribir este método y definir las acciones requeridas durante la selección.

4.5.4.1 Formato y procesado de la ADPU SELECT

La APDU SELECT de comando es la única APDU de comando que está estandarizada en la plataforma Java Card. Asegura la interoperabilidad en la selección de applets de varias implementaciones de la plataforma Java Card. El formato de la APDU se muestra a continuación:

Tabla 8: Estructura del comando SELECT

CLA	INS	P1	P2	Lc	Campo de datos
0x0	0xA4	0x4	0x0	Longitud del AID	Bytes del AID

La porción de datos de la APDU SELECT contiene el AID de algún applet, que suele tiene una longitud de entre 5 y 16 bytes. Para que se seleccione un applet, el campo entero de datos debe coincidir con el AID del applet.

Cuando se recibe una APDU, el JCRE decodifica su cabecera (CLA, INS, P1 y P2) para determinar si es un comando de selección de un applet, y si es así, si el AID de la APDU coincide con el del applet de la tarjeta. Una selección exitosa de un applet implica la deselección del applet actual, la selección del nuevo applet, y el envío de la APDU SELECT al método `process` del nuevo applet. Si la APDU no es para la selección de un applet, el JCRE se la entrega al applet actual para que la procese. En cualquier caso, si ocurre un error durante la selección, el JCRE señala el error devolviendo al host la palabra de estado 0x6999, y no se selecciona ningún applet de la tarjeta. El proceso de la SELECT APDU se muestra en el siguiente diagrama:

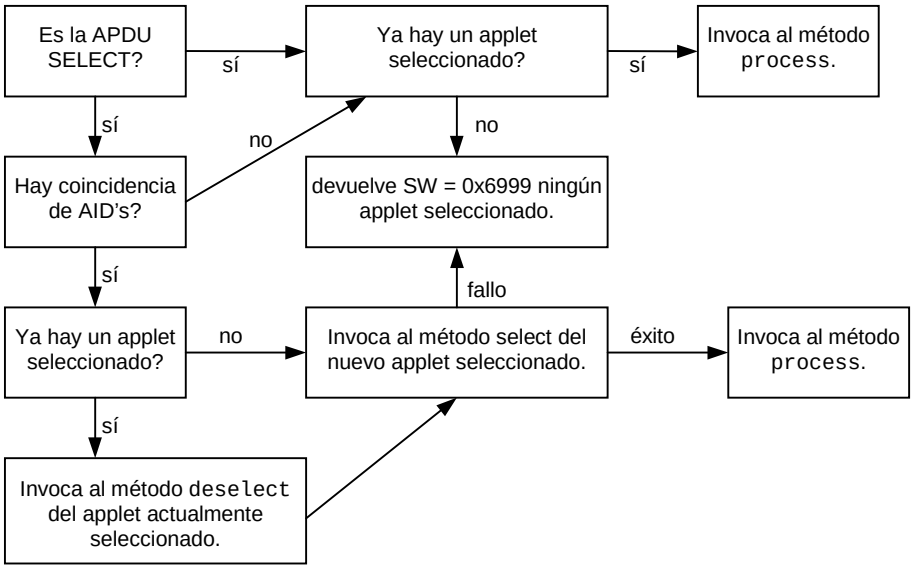


Figura 18: Procesado de la APDU de comando

4.5.4.2 El applet por defecto

Normalmente, los applets se llegan a seleccionar a través de un comando SELECT exitoso. Sin embargo, algunos sistemas de smart cards requieren de un applet por defecto que esté implícitamente seleccionado después de cada reset de la tarjeta.

Para seleccionar al applet por defecto, el JCRE llama al método `select` del applet por defecto y lo marca como el applet seleccionado actualmente. Debido a que no se requiere una APDU SELECT, no se llama al método `process` después de la selección. Si el método `select` del applet por defecto lanza una excepción o devuelve `false`, el applet no se selecciona hasta que la próxima APDU SELECT se procese.

La selección del applet por defecto es una característica opcional del JCRE. Cuando se soporta, la implementación del JCRE debería idear un mecanismo para especificar el applet por defecto.

4.5.5 EL MÉTODO DESELECT

Antes de que se seleccione un applet nuevo, el JCRE desactiva el applet actual llamando a su método `deselect`. Es posible que el nuevo applet seleccionado sea el mismo que el applet actual. En este caso, el JCRE lo deselecciona de todos modos y luego lo vuelve a seleccionar.

El método `deselect` permite al applet que lleve a cabo cualquier labor de limpieza para prepararse a si mismo para entrar a un estado de suspensión y permitir que otro applet se ejecute. La implementación por defecto en la clase `Applet` es un método vacío. Un applet debería sobrescribir este método para realizar las labores de limpieza que se requieran. Por ejemplo, la cartera necesitaría resetear la condición de seguridad o el estado de la transacción, que solamente es válido durante un periodo de selección.

El método `deselect` podría fallar. Sin embargo, el applet actual se deselecciona y el nuevo applet se selecciona a pesar del resultado de la ejecución del método `deselect`. El JCRE también ignora cualquier excepción lanzada desde el método `deselect`.

Además, en un reset o una pérdida de energía, el JCRE deselecciona automáticamente al applet sin que se llame a su método `deselect`. Por lo tanto, un applet no siempre puede contar con que se han llevado a cabo las operaciones de limpieza en el método `deselect`.

4.5.6 EL MÉTODO `PROCESS`

Cuando se recibe una APDU de comando, el JCRE llama al método `process` del applet actual. En el método `process`, se espera a que el applet lleve a cabo la función requerida en la APDU. El método `process` de la clase base `Applet` es un método abstracto. Un applet debe sobrescribir directamente o indirectamente este método. Usualmente, el método `process` está implementado como un despachador. Cuando se recibe una APDU de comando, el método decodifica la cabecera de la APDU y llama a un método de servicio para ejecutar la función requerida.

El JCRE encapsula la APDU en el argumento del método `process`, `apdu` (una instancia de la clase `APDU`). El applet invoca métodos sobre el objeto `apdu` para recibir o devolver datos de la APDU. El manejo de los comandos de una APDU se considera más adelante.

4.5.7 OTROS MÉTODOS DE LA CLASE `JAVACARD.FRAMEWORK.APPLET`

Hay otros dos métodos en la clase `Applet`: `selectingApplet` y `getShareableInterfaceObject`.

El sistema tradicional de las smart cards está orientado a ficheros. Los datos de las aplicaciones se guardan en ficheros. Se debe seleccionar un fichero antes de realizar cualquier acción en los datos del fichero. Es importante tener en cuenta que la APDU de la tabla de la sección 4.5.4.1 en la página 77, es el comando ISO de selección de DF's por nombre (DF = dedicated file). El JCRE puede determinar si el comando es para la selección de applets, mediante la comparación de los datos del comando con el AID de cualquiera de los applets de la tarjeta. Debido a que todas las APDU's están dirigidas al método `process` del applet actualmente seleccionado, el applet llama al método `selectingApplet` para distinguir si el comando de la APDU SELECT se usa para seleccionar ese applet o si se trata de la sección de un DF de ese applet. El método

`selectingApplet` devuelve `true` si la APDU selecciona a ese applet. Si no es así, devuelve `false`.

El método `getShareableInterface` está dirigido para la compartición de objetos entre applets. El JCRE lo invoca cuando otro applet requiere la interfaz de un objeto compartido de ese applet. Este método se describe más adelante en la sección 4.7.

4.6 TRABAJANDO CON APDU'S

En esta sección se describen las técnicas de manejo de las APDU's en un applet. Las APDU's son paquetes de datos, son el protocolo de comunicación del nivel de aplicación entre el software de la aplicación de la tarjeta y el software de la aplicación del host.

En los applets de Java Card, el mecanismo de comunicación con el host es diferente de las técnicas de red usadas en las aplicaciones Java. Muchas de las diferencias se deben a la naturaleza del protocolo APDU. La tecnología Java Card provee la clase `javacard.framework.APDU`, que define una interfaz poderosa y a la vez simple, para que el manejo de las APDU's por parte de los applets sea fácil.

En esta sección se comienza con una introducción de la clase `APDU`, en la que se explica como se encapsula una APDU en un objeto `APDU` y como el JCRE se lo se entrega al applet. Seguidamente, se explica como se procesa una APDU en un applet usando la clase `APDU` (como recibir una APDU de comando, como interpretar y ejecutar el comando de la APDU, y como devolver datos al host).

4.6.1 CLASE `APDU`

La clase `APDU` de las API's de Java Card provee una interfaz poderosa y flexible para manejar la APDU's, cuyas estructuras de comando y respuesta se ajustan a la especificación ISO 7816-4. Las APDU's se transmiten entre el host y la tarjeta gracias al protocolo de transporte (que se encuentra en el nivel más bajo). Hoy en día se usan principalmente dos protocolos de transporte en los sistemas de smart card: el protocolo T=0 y el protocolo T=1.

En la mayoría de los sistemas de smart card, no hay una clara separación entre el sistema operativo de la smart card y las aplicaciones. Las aplicaciones deben tener en cuenta el protocolo de transporte usado por el sistema subyacente. En la plataforma Java Card la clase `APDU` está diseñada cuidadosamente para que la complejidad y las diferencias entre los protocolos T=0 y T=1 se oculten a los desarrolladores de applets. En otras palabras, usando la clase `APDU`, se pueden escribir los applets para que trabajen correctamente sin tener en cuenta si la plataforma usa el protocolo T=0 ó el T=1.

La clase `APDU` también provee una manera orientada a objetos de tratar las APDU's. Los applets reciben y envían APDU's invocando los métodos definidos en la clase `APDU`. Por lo tanto, los desarrolladores de applets pueden concentrar sus esfuerzos en el proceso de los contenidos de los mensajes APDU en lugar de los detalles de cómo se construyen y transmiten las APDU's.

4.6.1.1 El objeto APDU

Como se describió en la sección 4.5, los applets no se comunican directamente con las aplicaciones del host. Ellas interaccionan con el JCRE, que por turnos usa la interfaz de E/S serie para comunicarse con el host. El JCRE crea un objeto `APDU`, una instancia de la clase `APDU`, que encapsula los mensajes APDU en un array interno de bytes, llamado buffer APDU.

En el entorno Java Card, el objeto `APDU` se puede ver como un objeto de comunicación. Cuando se recibe una APDU desde el host, el JCRE escribe la cabecera en el buffer APDU. Entonces invoca al método `process` del applet actualmente seleccionado y entrega al applet el objeto `APDU` como un parámetro del método. En el interior del método `process`, si la APDU entrante tiene datos, el applet puede llamar a los métodos del objeto `APDU` para recibir los datos. Después de procesar el comando, si el applet quiere enviar datos al host, llama otra vez a los métodos del objeto `APDU` para hacerlo. Los datos de la respuesta también se escriben en el buffer APDU. Entonces el JCRE envía los datos de respuesta al host.

4.6.1.2 Tamaño del buffer APDU

Para conseguir la interoperatividad entre las implementaciones de la plataforma Java Card, se requiere un buffer APDU de por lo menos 37 bytes, 5 bytes de cabecera más el IFSC por defecto (Information Field Size on Card). Una smart card con más memoria puede alojar un buffer APDU mayor.

IFSC está definido en la ISO 7816-3 para el protocolo T=1. El porqué del tamaño mínimo del buffer APDU está determinado por el tamaño por defecto del IFSC se explica en la sección 4.6.4.

4.6.2 INTERFAZ ISO 7816

Para facilitar el manejo de las APDU's de comando, la interfaz ISO 7816 de las API's de Java Card define un conjunto de constantes comunes relacionado con la ISO 7816-3 y la ISO 7816-4. Las constantes definidas en la interfaz ISO 7816 se pueden dividir en tres grupos:

- Constantes que se usan para indexar dentro del buffer APDU. Estas constantes empiezan por el prefijo `OFFSET`. Declaran el offset de cada byte de la cabecera de la APDU. Por ejemplo, `OFFSET_CLA` representa el offset del byte CLA del buffer APDU. Un applet usa estas constantes para acceder a los campos de la cabecera de la APDU.
- ISO 7816-4 (palabras de estado definidas para la respuesta). Estas constantes comienzan con el prefijo `SW`. Son las palabras de estado definidas en la ISO 7816-4 más comúnmente utilizadas. La palabra de estado es un campo obligatorio de una APDU de respuesta. Todas las constantes de palabra de estado son de tipo short, que comprende dos bytes.
- Constantes CLA e INS. La interfaz ISO 7816 también define el byte de codificación CLA e INS para las APDU's de los comandos SELECT y EXTERNAL AUTHENTICATE.

4.6.3 TRABAJANDO CON APDU'S EN LOS APPLETS

Los applets manejan los comandos APDU en el método `process`, como se describe en los siguientes pasos. Junto a la descripción de cada paso, se explica el uso de los métodos de la clase `APDU`.

4.6.3.1 Recuperar la referencia del buffer.

El primer paso para procesar una APDU es que el applet recupere una referencia al buffer APDU invocando el método `getBuffer`. El buffer APDU es un array de bytes cuya longitud puede ser determinada usando `apdu_buffer.length`.

```
public void process(APDU apdu) {
      //recupera el buffer APDU
      byte[] apdu_buffer = apdu.getBuffer();
}
```

Nótese que el JCRE exige que la referencia al objeto `APDU` o la referencia al buffer APDU no se pueda guardar en variables de la clase, variables de instancias, o un array. En cambio, un applet solo debería guardar las referencias en variables locales y parámetros de métodos, que son datos temporales en el ámbito de un método. Esta exigencia se debe a medidas de seguridad, ya que un applet podría retener impropiamente la referencia de una APDU de datos, perteneciente a otro applet.

4.6.3.2 Examinar la cabecera de la APDU de comando.

Cuando se invoca al método `process`, sólo los 5 primeros bytes de la cabecera están disponibles en el buffer APDU. Los 4 primeros son la cabecera de la APDU [CLA, INS, P1, P2] y el quinto byte (P3) es un campo adicional de longitud. El significado de P3 está implícitamente determinado según el caso al que pertenezca el comando (ver sección 3.2.4.3):

- Para el caso 1, P3 = 0.
- Para el caso 2, P3 = Le, la longitud de los datos salientes de respuesta.
- Para los casos 3 y 4, P3 = Lc, la longitud de los datos entrantes de comando.

Los bytes restantes del buffer están indefinidos y el applet no debería leerlos ni escribir en ellos.

Cuando un applet obtiene el buffer APDU, debería examinar primero la cabecera APDU para determinar si el comando está bien formado y si el comando se puede ejecutar:

- El comando está bien formado: los bytes de la cabecera están codificados correctamente.
- El comando puede ser ejecutado: el comando está soportado por el applet y las condiciones internas y de seguridad son las apropiadas para el comando.

Si la comprobación falla, el applet debería terminar la operación lanzando una `ISOException`.

Las constantes definidas en la interfaz ISO 7816, se deberían usar como índices dentro del buffer APDU para acceder a los bytes de la cabecera. Las constantes son las siguientes:

Tabla 9: Offsets definidos en la interfaz `ISO7816`, para la cabecera de una APDU

Nombre de la constante	Significado	Valor
`OFFSET_CLA`	offset del buffer APDU para el campo CLA	`OFFSET_CLA = 0`
`OFFSET_INS`	offset del buffer APDU para el campo INS	`OFFSET_INS = 1`
`OFFSET_P1`	offset del buffer APDU para el campo P1	`OFFSET_P1 = 2`
`OFFSET_P2`	offset del buffer APDU para el campo P2	`OFFSET_P2 = 3`

Por ejemplo, el siguiente fragmento de código examina el byte CLA:

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

El uso de las constantes hace que el código del applet sea más legible.

4.6.3.3 Recibir los datos de la APDU de comando

Además de especificar una instrucción para que la lleve a cabo el applet, la cabecera de la APDU [CLA, INS, P1, P2] especifica la estructura (o caso) de la APDU (si la APDU tiene datos entrantes y si se esperan datos en la respuesta). Si es de caso 3 o de caso 4, la APDU de comando tiene datos entrantes que forman parte de la instrucción. El applet puede saber el número de bytes de datos a partir del campo Lc (el quinto byte del buffer APDU).

```
short data_length = (short) (apdu_buffer[ISO7816.OFFSET_LC] &
0xFF);
```

Los tipos de datos enteros en el lenguaje de programación Java son con signo, es decir, que el bit más significante determina si es un número positivo o negativo. Sin embargo, el campo Lc debería ser interpretado como un valor sin signo, porque no tiene sentido tener una longitud negativa. En el fragmento de código anterior, al byte Lc se le hace una AND con la constante 0xFF. Esto se hace para convertir un byte con signo a un valor sin signo.

Para leer datos dentro del buffer APDU, el applet invoca al método `setIncomingAndReceive` de la clase `APDU`:

```
public short setIncomingAndReceive() throws APDUException
```

Como su nombre indica, el método `setIncomingAndReceive` cumple dos objetivos. El primero es que ajusta el JCRE al modo de recepción de datos. La comunicación hacia y desde la tarjeta es half-duplex. Es decir, los datos se envían desde el host hacia la tarjeta o desde la tarjeta hacia el host, pero no las dos cosas al mismo tiempo. En este paso se dan instrucciones al JCRE para que se trate al quinto byte del buffer APDU como el campo Lc y prepara al JCRE para aceptar los datos entrantes. Lo siguiente, es solicitar al JCRE la recepción de los bytes de datos del comando,

empezando a partir del offset `ISO7816.OFFSET_DATA` (=5) en el buffer APDU. En la siguiente figura se muestra lo comentado:

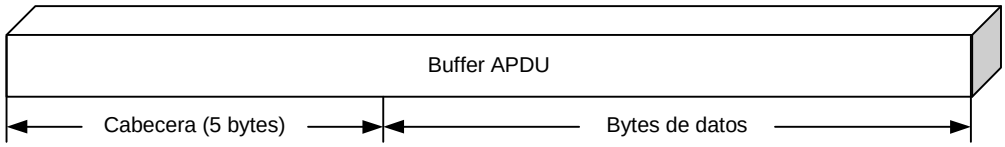


Figura 19: Buffer APDU tras invocar al método `setIncomingAndReceive`

El método `setIncomingAndReceive` devuelve el número de bytes que lee. Devuelve 0 si no hay datos disponibles. Si se llama al método `setIncomingAndReceive` cuando el JCRE está todavía en modo recepción de una llamada previa al mismo método, da lugar a una `APDUException` con el código de causa `APDUException.ILLEGAL_USE`.

4.6.3.3.1 Para recibir un comando con gran cantidad de datos

En la mayoría de los casos, el método `setIncomingAndReceive` es suficiente para leer el número de bytes de datos del buffer APDU especificado en el campo Lc. Sin embargo, para una APDU de comando que tiene más datos de los que pueden caber en el buffer APDU, la llamada al método `setIncomingAndReceive` debe ser seguida por una o más llamadas al método `receiveBytes`:

```
public short receiveBytes(short bOff) throws APDUException
```

Para los datos largos de una APDU de comando, un applet puede procesar los datos por trozos y entonces llamar al método `receiveBytes` para leer datos adicionales en el buffer APDU. Con el método `receiveBytes`, se puede especificar el offset del buffer APDU donde se reciben los datos. Esto permite al applet controlar como se usa el buffer APDU en el proceso de los datos entrantes. Como se muestra en la Figura 20, el applet quizás haya procesado datos procedentes de la llamada previa al método `setIncomingAndReceive` o el método `receiveBytes`, excepto una pequeña cantidad de bytes.

El applet puede mover estos bytes (los que quedan por leer) al comienzo del buffer y entonces recibir el próximo grupo de bytes, para que sean añadidos a los bytes que todavía están en el buffer. Esta característica es importante en instancias donde los datos, que se necesitan procesar como un todo, se leen mediante invocaciones al método.

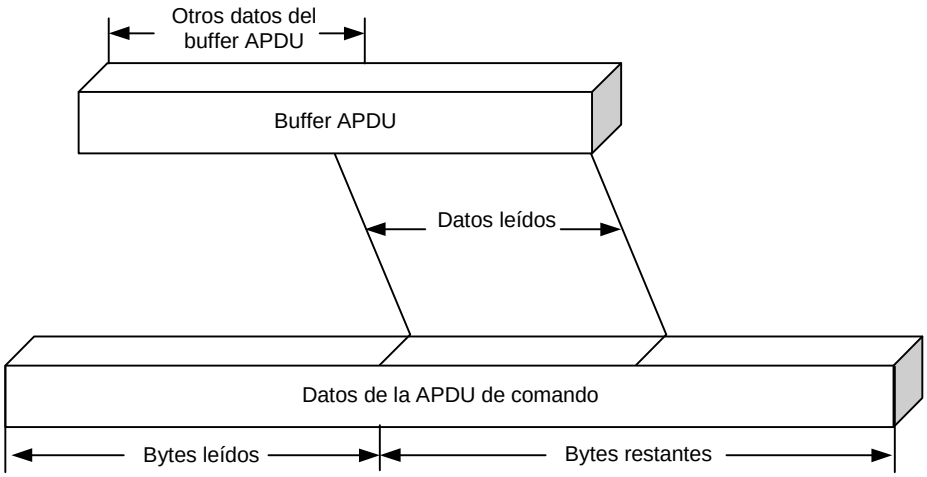


Figura 20: Invocación del método `receiveBytes`

Como el método `setIncomingAndReceive`, el método `receiveBytes` garantiza la entrega de forma síncrona, recuperando tantos bytes como sea posible. Sin embargo, dependiendo de cuantos bytes envíe el host como un grupo y dependiendo de la implementación del JCRE, puede que ambos métodos lean menos bytes que el espacio disponible en el buffer APDU.

Como regla general, tanto el método `setIncomingAndReceive` como `receiveBytes`, están optimizados. Si todos los datos del comando caben en el buffer APDU comenzando en el offset `ISO7816.OFFSET_CDATA` (=5), debería ser suficiente una sola invocación del método `setIncomingAndReceive` para obtener todos los datos. No es necesario llamar el método `receiveBytes`. La mayoría de las APDU's de comando se encuentran en esta categoría.

Cuando están disponibles más datos, que caben en el buffer APDU, el applet debe llamar al método `receiveBytes`. Si los bytes restantes caben en el espacio disponible a partir del offset especificado en el buffer APDU, el método `receiveBytes` garantiza la entrega de todos los datos restantes. De otra manera, el método lee tantos bytes como quepan en el buffer, y posiblemente menos. El applet debería llamar repetidamente al método `receiveBytes`, procesando o moviendo los bytes del buffer de datos de la APDU en cada llamada, hasta que se lean todos los datos disponibles. El siguiente ejemplo incluye el método `receiveBytes` dentro de un bucle `while`:

```
public void process(APDU apdu) {
     byte[] apdu_buffer = apdu.getBuffer();
     short total_bytes = (short)
     (apdu_buffer[ISO7816.OFFSET_LC] & 0xFF);
     //lee datos del interior del buffer APDU
     short read_count = apdu.setIncomingAndReceive();
     //consigue el número de bytes restantes
     short bytes_left = (short) (total_bytes –
     read_count);
     while (true) {
          //procesa los datos del buffer APDU
          //o copia los datos en un buffer interno
          //...
```

```
            //comprueba si todavía quedan datos restantes
            //si no, salta fuera del bucle
            if (bytes_left <= 0) break;
            //si hay datos restantes, lee más datos
            read_count = apdu.receiveBytes((short)0);
            bytes_left -= readcount;
        }
        //ejecuta otras tareas y responde al host
        //...
}
```

4.6.3.4 Procesar la APDU de comando y generar los datos de la respuesta

La cabecera de la APDU [CLA, INS, P1, P2] identifica una instrucción que el applet debería llevar a cabo. Cuando se ejecuta la instrucción, el applet debería procesar los datos del comando en el buffer APDU si el comando es del caso 3 o del caso 4. Para reducir el uso de memoria, el applet suele usar el buffer APDU como un buffer de trabajo para mantener los resultados intermedios o los datos de respuesta.

4.6.3.5 Devolver datos en la APDU de respuesta

Después de completar la instrucción especificada en la APDU de comando, el applet puede devolver datos al host. Ya que el camino de comunicación subyacente es half-duplex, para enviar datos, en primer lugar, el applet debe llamar al método `setOutgoing` para indicar que ahora desea enviar datos de respuesta.

```
public short setOutgoing() throws APDUException
```

El método `setOutgoing` ajusta el JCRE al modo de envío de datos reseteando la dirección de transferencia hacia fuera. A diferencia del método correspondiente, `setIncomingAndReceive` para leer datos, el método `setOutgoing` no envía ningún byte, sólo ajusta el modo de transferencia. Una vez que se llame al método `setOutgoing`, se descartará cualquier dato entrante restante, y el applet no podrá continuar recibiendo datos.

El método `setOutgoing` devuelve el número de bytes de datos de respuesta (Le) que el host espera como respuesta de la APDU de comando. En el caso 4 del protocolo T=0, debido a que el campo Le efectivo no se puede determinar, se asume que el máximo valor permitido para el campo Le es de 255 bytes. Para otros casos se devuelve el campo Le efectivo de la APDU de comando. Sin embargo, no es necesario para el applet el conocer qué protocolo de transporte se está usando.

Aunque la aplicación del host solicite que se le devuelvan Le bytes de datos, el applet podría enviar más o menos datos que los indicados en ese número. Después de invocar al método `setOutgoing`, el applet debe llamar al método `setOutgoingLegth` para indicar al host cual es la cantidad total de datos de respuesta (sin incluir las SW) que se enviarán realmente:

```
public void setOutgoingLength(short length) throws APDUException
```

El host asume que la longitud por defecto de los datos de respuesta es 0, por eso no se necesita llamar a este método si el applet no envía ningún dato. El applet no puede enviar más de 256 bytes al host, o la invocación del método `setOutgoingLength`

resultará en una `APDUException` con el código de causa `APDUException.BAD_LENGTH`.

Lo próximo, para enviar realmente los datos de la respuesta, es que el applet llame al método `sendBytes`:

```
public void sendBytes(short bOff, short len) throws APDUException
```

El método `sendBytes` envía los `len` bytes de datos del buffer APDU a partir del offset especificado en `bOff`. Por lo tanto, el applet debe generar o copiar la respuesta en el buffer APDU antes de invocar este método.

Los métodos `setOutgoing`, `setOutgoingLength`, y `sendBytes` se deben invocar en el orden correcto; si no, el JCRE lanzará una `APDUException`. El siguiente fragmento de código demuestra sus usos en el applet:

```
public void process(APDU apdu) {
      //recibe y procesa la APDU de comando
      ...
      //ahora está preparado para enviar los
      //datos de respuesta.
      //primero ajusta el JCRE al modo de envío de datos y
      //obtiene la longitud esperada de la respuesta (Le).
      short le = apdu.setOutgoing();
      //informa al host de que el applet enviará
      //realmente 10 bytes
      apdu.setOutgoingLength((short)10);
      //prepara los datos en el buffer APDU
      //empezando en el offset 0
      //...
      //al final envía los datos
      apdu.sendBytes((short)0, (short)10);
}
```

Si el applet necesita enviar más datos, puede actualizar el buffer APDU con los nuevos datos y llamar repetidamente al método `sendBytes` hasta que se envíen todos los bytes.

Para reducir la sobrecarga, la clase `APDU` provee el método `setOutgoingAndSend` para enviar los datos salientes:

```
public void setOutgoingAndSend(short bOff, short len) throws APDUException
```

El método `setOutgoingAndSend` combina los métodos `setOutgoing`, `setOutgoingLength`, y `sendBytes` en una única llamada e implementa las siguientes tareas:

- Ajusta el modo de transferencia, al modo de envío.
- Ajusta la longitud de los datos de respuesta a `len`.
- Envía los bytes de datos desde el buffer APDU a partir del offset `bOff`.

El método `setOutgoingAndSend` es análogo al método `setIncomingAndReceive`. Ambos métodos ajustan el modo de transferencia y envían o reciben datos en una sola llamada. Sin embargo, hay una pega usando el método `setOutgoingAndSend`: los datos deben caber completamente en el buffer APDU. Es decir, a diferencia del método `setIncomingAndReceive`, no se pueden enviar más datos o no se puede alterar el contenido del buffer APDU después de que el applet llame al método `setOutgoingAndSend`. Debido a que la mayoría de las APDU's de respuesta de un applet encapsulan un campo de datos que contiene pocos bytes y caben bien dentro del buffer APDU, el método `setOutgoingAndSend` provee la forma más eficiente para enviar una respuesta corta, es la forma que provoca menor sobrecarga por protocolo.

4.6.3.5.1 Envío de datos desde otras localizaciones

Los métodos `sendBytes` y `setOutgoingAndSend` envían datos desde el buffer APDU. Esto es práctico si los datos de la respuesta ya están listos en el buffer APDU. Si los datos se guardan en un buffer local del applet o en un fichero, el applet necesita copiar los datos en el buffer APDU. Para reducir la sobrecarga por el movimiento de datos, el applet puede llamar al método `sendBytesLong`:

```
public void sendBytesLong(byte[] outData, short bOff, short len)
throws APDUException
```

El método `sendBytesLong` envía los `len` bytes de datos a partir del offset `bOff` desde el array de bytes `outData`. Al igual que el método `sendBytes`, se puede llamar al método `sendBytesLong` repetidamente, pero solo se puede llamar si los métodos `setOutgoing` y `setOutgoingLength` se han invocado primero.

4.6.3.5.2 Envío de una respuesta larga

Para enviar una respuesta larga, el applet puede llamar repetidamente al método `sendBytes` o al método `sendBytesLong`. De hecho, se puede llamar a ambos métodos sucesivamente para enviar datos desde varias localizaciones. El siguiente código demuestra como un applet envía una respuesta larga:

```
//la APDU de comando Get_Account info requiere que el appet
//envíe el nombre del titular de la cuenta, el número de la
//cuenta y la fecha de expiración
//además, se requiere que el applet envíe el valor de
//dispersión, calculado a partir del número de la cuenta,
//y la fecha de expiración para asegurar que los datos
//se transmiten correctamente.
//
//el applet guarda el nombre del titular de la cuenta,
//el número de cuenta y la fecha de expiración en
//arrays de bytes separados
private byte[] name; //20 bytes
private byte[] account_number; //9 bytes
private byte[] expiration_date; //4bytes
//después de procesar la APDU de comando, el applet está
//preparado para enviar los datos de respuesta
//el número de bytes de los datos de respuesta
//total_bytes = 20 (nombre) + 9 (número de cuenta)
```

```
//+ 4 (fecha de expiración) + 8 (valor de dispersión)
//= 41 bytes
short total_bytes = (short)41;
//paso 1:
//ajustar la dirección de transferencia hacia fuera
short le = apdu.setOutgoing();
//paso 2:
//informar al host del número real de bytes de la respuesta
apdu.setOutgoingLength(total_bytes);
//paso 3:
//enviar el nombre
apdu.sendBytesLong(name, (short)0, (short) name.length);
//enviar la fecha de expiración
apdu.sendBytesLong(expiration_date, (short)0, (short)
expiration_date.length);
//ahora calcula el valor de dispersión en el buffer APDU.
//asume que el valor de dispersión se genera en el offset 0
//...
//envía el valor de dispersión
apdu.sendBytes((short)0, HASH_VALUE_LENGTH);
//y termina
return;
```

Una APDU de respuesta consta de un campo de datos opcional, seguido por las palabras (2 bytes cada una) de estado obligatorias. El JCRE determina la palabra de estado apropiada después de que el applet vuelva del método `process`. Para reducir la sobrecarga por protocolo, el JCRE envía la última parte de los datos de respuesta junto a los bytes de estado. Normalmente, el método `sendBytes` o el método `sendBytesLong` envían datos y finalizan de forma síncrona, para que el applet pueda actualizar el buffer APDU inmediatamente. Sin embargo el applet no debería alterar el buffer APDU después de la última invocación a uno de esos dos métodos. En el ejemplo anterior, el JCRE no envía el valor de dispersión inmediatamente después de la última invocación al método `sendBytes`. Si el applet necesita llevar a cabo otras tareas antes de dejar el método `process`, debería dejar intacto el buffer APDU, o posiblemente la última porción de datos de datos esté corrupta cuando se envíe.

4.6.3.6 Devolución de las palabras de estado

Una invocación al método `process` de un applet implica el intercambio de una APDU de comando y una APDU de respuesta entre el host y el applet. En el método `process`, en primer lugar, el applet lee la APDU de comando recibida y luego escribe los datos de respuesta para que sean enviados. Se alcanza el estado final cuando se ajusta la palabra de estado de la APDU de respuesta. La palabra de estado informa al host del estado del procesamiento de la instrucción de la APDU de comando. En este paso puede ocurrir uno de estos tres resultados:

- En un regreso normal del método `process`, el JCRE envía automáticamente los bytes de estado de terminación normal (0x9000) al host. Para los comandos del caso 2 o del caso 4, el JCRE adjunta las palabras de estado a los datos de respuesta del applet.
- Si ocurre un error, en cualquier punto durante el proceso del comando, el applet termina la operación y lanza una `ISOException` invocando al

método estático `ISOException.throwIt(reason)`. El applet asigna al parámetro `reason`, la palabra de estado. Si el applet no trata la excepción `ISOException`, la captura el JCRE. El JCRE recupera el código de causa y se lo envía como si fuera la palabra de estado de la misma manera que si enviase la palabra 0x9000 de terminación normal. El applet también puede usar la palabra de estado para indicar el aviso de que se ha completado el comando, pero existe un problema potencial.

La interfaz ISO7816 incluye la mayoría de las palabras de estado definidas (en la ISO 7816-4) para la respuesta. Por ejemplo, la constante `SW_CLA_NOT_SUPPORTED` indica que el applet no soporta el byte CLA de una APDU de comando. Las constantes de la interfaz ISO7816, usadas en las palabras de estado, ofrecen una manera práctica para que un applet especifique palabras de estado que cumplan con la ISO 7816-4 y para que el código del applet sea más legible y mantenible.

Antes de que se lance una excepción, quizás el JCRE se haya cambiado al modo de envío, y los bytes de datos quizás se hayan transmitido. Para finalizar la respuesta, el JCRE adjunta la palabra de estado a los datos que el applet ya haya enviado.

- Si el sistema Java Card subyacente detectase un error, el comportamiento del JCRE está indefinido. Por ejemplo, el JCRE podría lanzar una excepción, como una `APDUException` o una `TransactionException`. Quizás el JCRE no implemente un manejador por cada tipo de excepción o quizás, no encuentre el código de causa en el objeto de excepción que capture. En ambos casos el JCRE devuelve una `ISO7816.SW_UNKNOWN` (0x6F00) sin especificar un diagnóstico preciso. Si el error es más severo, el JCRE podría decidir silenciar la tarjeta, bloquear al applet para que no se ejecute, o llevar a cabo tantas operaciones como sean necesarias para asegurar la seguridad y la integridad del applet y de la tarjeta.

4.6.4 PROCESADO DE UNA APDU ESPECÍFICA DEL PROTOCOLO

En la plataforma Java Card, los desarrolladores de applets programan en la capa de aplicación, usando la clase `APDU` para manejar las APDU's. La clase `APDU` provee una interfaz simple y común para los applets sin tener en cuenta el protocolo de transporte subyacente (T=0 ó T=1) usado. Sin embargo, algunos sistemas de smart cards fueron diseñados empleando un protocolo de transporte específico. Para ser compatible con tales sistemas, los applets deben utilizar atributos específicos de cada protocolo para que se puedan comunicar con una gran cantidad de dispositivos de aceptación de tarjetas. Por lo tanto, la clase `APDU` también define métodos que puede usar un applet para descubrir el protocolo subyacente, para reservar espacio para recibir o enviar datos, y solicitar tiempo de proceso adicional.

Esta sección explica por qué están definidos estos métodos y como se usan. Sin embargo, todos los métodos definidos en la clase `APDU` se pueden utilizar ya que son independientes del protocolo. Es decir, no es necesario el conocimiento de los detalles del protocolo de transporte para usar los métodos.

4.6.4.1 Método `getProtocol`

```
public static byte getProtocol()
```

El método `getProtocol` devuelve el tipo de protocolo ISO 7816 soportado en la plataforma Java Card.. El resultado puede ser `APDU.PROTOCOL_T0` para el protocolo T=0 ó `APDU.PROTOCOL_T1` para el protocolo T=1.

4.6.4.2 Método `getInBlockSize`

```
public static short getInBlockSize()
```

El método `getInBlockSize` devuelve el tamaño configurado de bloque entrante. El protocolo T=1 es un protocolo orientado a bloque. Una APDU de comando se transmite en un solo bloque o en algunos bloques si se soporta el mecanismo de encadenamiento. El mecanismo de encaminamiento permite que se transmita una APDU de comando grande en bloques sucesivos. Cada bloque se compone de tres campos: un campo de prólogo, un campo de información y un campo de epílogo. El campo de prólogo y epílogo limitan el bloque, y el campo de información transporta la APDU.

En el protocolo T=1, el tamaño de bloque devuelto por el método corresponde al parámetro IFSC (Information Field Size on Card). El IFSC especifica la máxima longitud del campo de información en un bloque que una tarjeta puede aceptar. El valor del IFSC varía de una tarjeta a otra. El valor por defecto del IFSC es de 32 bytes. Este es el porqué de que el tamaño mínimo del buffer APDU esté ajustado a 37 bytes (5 bytes de la cabecera más los 32 bytes del IFSC). Típicamente, el buffer APDU es algo más grande, para que le permita al applet reservar algunos bytes de datos en el buffer APDU y recibir después los bytes de datos del comando, sin el peligro de causar un overflow.

El protocolo T=0 es un protocolo orientado a bit. No tiene un requerimiento de longitud máxima y solo necesita recibir un bit en cada instante. De este modo, el método `getInBlockSize` devuelve 1 para el protocolo T=0.

Un applet puede usar el método `getInBlockSize`, independientemente del protocolo, para indicar el máximo número de bytes que se pueden recibir en el buffer mediante una sola operación del nivel subyacente de E/S. Los métodos `receiveBytes` y `setIncomingAndReceive` consisten en una o más de estas operaciones de E/S.

Además, un applet puede comprobar `InBlockSize` para asegurar que queda suficiente espacio en el buffer APDU cuando se invoque al método `receiveBytes`. Por ejemplo, para optimizar el uso del espacio, un applet podría conservar `n` bytes del comienzo del buffer APDU para guardar los datos intermedios sin tener que utilizar un buffer distinto. Para hacer esto, el applet necesita averiguar si los bytes restantes del buffer APDU son suficientes para mantener por lo menos a uno de los bloques de datos que el host puede enviar.

4.6.4.3 Método `getOutBlockSize`

```
public static short getOutBlockSize()
```

El método `getOutBlockSize` es análogo al método `getInBlockSize`. Devuelve el tamaño configurado de bloque saliente. En el protocolo T=1, este tamaño corresponde con el IFSD (Information Field Size for interface Device), que es la máxima longitud del campo de información de un bloque que el host puede aceptar. El valor inicial definido en la ISO 7816-3 es de 32 bytes. En el protocolo T=0, el método devuelve 258,contando 2 bytes de estado. Por ello la cantidad máxima de bytes que se pueden enviar desde el applet hasta el host, es de 256.

El IFSD especifica el atributo en el lado del host. A diferencia de `InBlockSize`, normalmente, un applet no comprueba `OutBlockSize`. Para el protocolo T=1, si el JCRE no puede enviar todos los datos en un bloque (el número total de datos de la respuesta alcanza el límite dado por IFSD) el JCRE puede dividir los datos en bloques y enviarlos usando el mecanismo de encadenamiento. Sin embargo, si el protocolo subyacente T=1 no soporta el encadenamiento, cuando el JCRE no pueda responder al host con una gran cantidad de datos, lanza una `APDUException`. El siguiente código de ejemplo demuestra como un applet puede trabajar alrededor de esta limitación cuando no se permite el encadenamiento de bloques.

```
//un applet wallet guarda los registros de las
//transacciones de las tres últimas transacciones
//
//por simplicidad, cada registro grabado se guarda en un
//array de bytes y el tamaño total del registro grabado
//es menor de 256 bytes
//
//para responder a una APDU de comando READ_TRANSACTION_LOG
//el applet wallet envía al host cada registro
//de transacción grabado
//
//comprueba el protocolo
if (apdu.getProtocol() == APDU.PROTOCOL_T0) {
      //sin problema, los registros de transacciones se
      //pueden enviar de una sola vez
      //...
} else {
      //consigue el tamaño de bloque
      short out_block_size = apdu.getOutBlockSize();
      //comprueba si el tamaño total del registro de
      //transacciones es más pequeño que el tamaño del
      //bloque de salida
      if (TOTAL_LOG_RECORD_SIZE <= out_block_size) {
            //sin problema, envía todos los registros
            apdu.setOutgoingLength(TOTAL_LOG_RECORD_SIZE);
            apdu.sendBytesLong(log_record_1, (short)0,
            (short) log_record_1.length);
            apdu.sendBytesLong(log_record_2, (short)0,
            (short) log_record_2.length);
            apdu.sendBytesLong(log_record_3, (short)0,
            (short) log_record_3.length);
            return;
```

```
      } else {
            //envía una grabación, pero informa al host de
            //que hay más grabaciones
            apdu.setOutgoingLength((short)
            log_record_1.length);
            apdu.sendBytesLong(log_record_1, (short)0,
            (short)log_record_1.length);
            //le dice al host que hay dos registros grabados
            //el host puede dar otra APDU de comando
            //para recuperar las grabaciones restantes
            ISOException.throwIt(SW_2_MORE_RECORDS);
      }
}
```

4.6.4.4 Método `setOutgoingNoChaining`

```
public short setOutgoingNoChaining() throws APDUException
```

Este método se usa para ajustar la dirección de transferencia de datos hacia fuera sin usar el encadenamiento de bloques y para obtener la longitud esperada de los datos (campo Le) de la APDU de respuesta. Los applets deberían utilizar este método en lugar de `setOutgoing` para que sean compatibles con las especificaciones EMV.

El método `setOutgoingNoChaining` se puede usar de la misma manera que el método `setOutgoing`. Una vez que se invoca, se descarta cualquier dato entrante restante. El applet debe invocar al método `setOutgoingLength` para informar al host del número real de bytes que se enviarán.

4.6.4.5 Método `waitExtension`

```
public byte waitExtension()
```

Cuando el host no recibe ninguna respuesta en un tiempo máximo de tiempo definido en la ISO7816-3, considera que la tarjeta no escucha y que su tiempo ha acabado. Un applet llama al método `waitExtension` para solicitar al host tiempo adicional de proceso, para que no finalice su tiempo mientras el applet está llevando a cabo una operación larga (un número significativo de escrituras en EEPROM u operaciones criptográficas complejas).

Un applet puede llamar al método `waitExtension` en cualquier instante durante el proceso de una APDU de comando. No es necesario llamar a este método si la tarjeta tiene un temporizador hardware que automáticamente envía un `waitExtension` al host.

4.6.5 PASOS A SEGUIR PARA PROCESAR LAS APDU'S

El contenido de esta sección trata acerca de la utilización de métodos de la clase `APDU` para examinar la cabecera de la APDU, leer los datos del comando y enviar datos de respuesta.

A continuación se expone una metodología, con los pasos que hay que seguir para que un applet pueda manejar cada caso de APDU de comando. En todos los casos,

el applet puede lanzar una `ISOException` con el código de causa adecuado, para indicar errores.

4.6.5.1 Caso 1: Sin datos de comando, ni datos de respuesta.

1. Se llama al método `process` del applet. El applet examina los 4 primeros bytes del buffer APDU y determina si es un comando del caso 1. El campo P3 (el quinto byte del buffer APDU) está a 0.
2. El applet lleva a cabo la solicitud especificada en la cabecera de la APDU.
3. El applet vuelve del método `process`.

4.6.5.2 Caso 2: Sin datos de comando, con datos de respuesta.

1. Se llama al método `process` del applet. El applet examina los 4 primeros bytes del buffer APDU y determina si es un comando del caso 2. El campo P3 se interpreta como el campo Le.
2. El applet lleva a cabo la solicitud especificada en la cabecera de la APDU.
3. El applet envía los datos de respuesta. La respuesta puede ser corta o larga y se trata de forma diferente, según el tamaño de los datos.

4.6.5.2.1 Respuesta corta (los datos de la respuesta caben en el buffer APDU):

1. El applet llama al método `setOutgoingAndSend` y especifica la longitud real de los datos de la respuesta.
2. El applet vuelve del método `process`.

4.6.5.2.2 Respuesta larga (los datos de la respuesta no caben en el buffer APDU):

1. El applet llama al método `setOutgoing` y obtiene el campo Le.
2. El applet llama al método `setOutgoingLength` para informar al host de la longitud real de los datos de respuesta.
3. El applet llama a los métodos `sendBytes` o `sendBytesLong` (repetidamente si es necesario) para enviar grupos de bytes de respuesta.
4. El applet vuelve del método `process`.

4.6.5.3 Caso 3: Recepción de datos del comando, sin datos de respuesta.

1. Se llama al método `process` del applet. El applet examina los 4 primeros bytes del buffer APDU y determina que es un comando de clase 3. El campo P3 se interpreta como el campo Lc.
2. El applet llama al método `setIncomingAndReceive` y repite la llamada al método `receiveBytes` para recibir los bytes si es necesario. Cada grupo de bytes de datos del comando se procesan o copian en un buffer interno, a medida que se van recibiendo.

3. El applet vuelve del método `process`.

4.6.5.4 Caso 4: Recepción de datos del comando y envío de datos de la respuesta.

El caso 4 es una combinación de los casos 3 y 2. En primer lugar, el applet recibe los datos del comando como se describe en el caso 3. Seguidamente, el applet envía los datos de la respuesta como se describe en el caso 2. Al final, el applet vuelve del método `process`.

4.7 EL APPLET FIREWALL Y LA COMPARTICIÓN DE OBJETOS

La plataforma Java Card es un entorno multiaplicación. Múltiples applets de diferentes suministradores pueden coexistir en una sola tarjeta, y se pueden descargar applets adicionales después de la fabricación de la tarjeta. A menudo, un applet guarda información altamente confidencial, como dinero electrónico, huellas dactilares, claves privadas usadas en criptografía, etc. El compartir estos datos confidenciales entre applets debe estar cuidadosamente limitado.

En la plataforma Java Card, el aislamiento del applet se consigue a través del mecanismo del applet firewall. El applet firewall confina al applet en su propia área designada. Así, un applet queda reservado del acceso a sus contenidos o del comportamiento de objetos que pertenecen a otros applets.

Para soportar aplicaciones que cooperen entre sí, en una sola tarjeta, la tecnología Java Card provee mecanismos de compartición de objetos bien definidos.

El applet firewall y los mecanismos de compartición afectan a la forma en la que se escriben los applets. Esta sección explica el comportamiento de los objetos, excepciones, y applets en presencia del firewall y se explica como los applets pueden compartir datos de forma segura, usando las API's de Java Card.

4.7.1 APPLET FIREWALL

El applet firewall provee protección contra los problemas más frecuentes de seguridad: errores del desarrollador y fallos en el diseño que quizás permitan que se fuguen datos a otro applet. También provee protección contra la piratería. Quizás un applet sea capaz de obtener la referencia de un objeto de una localización pública accesible, pero si el objeto pertenece a otro applet de un paquete diferente, el firewall se encarga de impedir el acceso al objeto. De este modo, un applet que no funcione correctamente, o un applet hostil, no puede afectar a las operaciones de otros applets o a las del JCRE.

4.7.1.1 Los contextos

El applet firewall divide al sistema de objetos de Java Card en espacios de objetos, separados y protegidos, llamados contextos. El applet firewall es la frontera entre un contexto y otro. Cuando se crea la instancia de un applet, el JCRE le asigna un contexto. El contexto es esencialmente un contexto de grupo. Todas las instancias de applets de un sólo paquete de Java, comparten el mismo contexto de grupo. No hay

firewall entre dos instancias de applets que pertenezcan a un mismo de contexto de grupo. Se permite el acceso a objetos, entre applets del mismo contexto de grupo. Sin embargo, el firewall deniega el acceso a un objeto de un contexto de grupo diferente.

Además, el JCRE mantiene su propio contexto JCRE. El contexto JCRE es un sistema de contexto dedicado que tiene privilegios especiales: al contexto JCRE se le permite el acceso a cualquier contexto de un applet, pero lo contrario, el acceso desde el contexto de un applet al contexto JCRE, está prohibido por el firewall. Las divisiones del sistema de objetos de Java Card se ilustran en la siguiente figura:

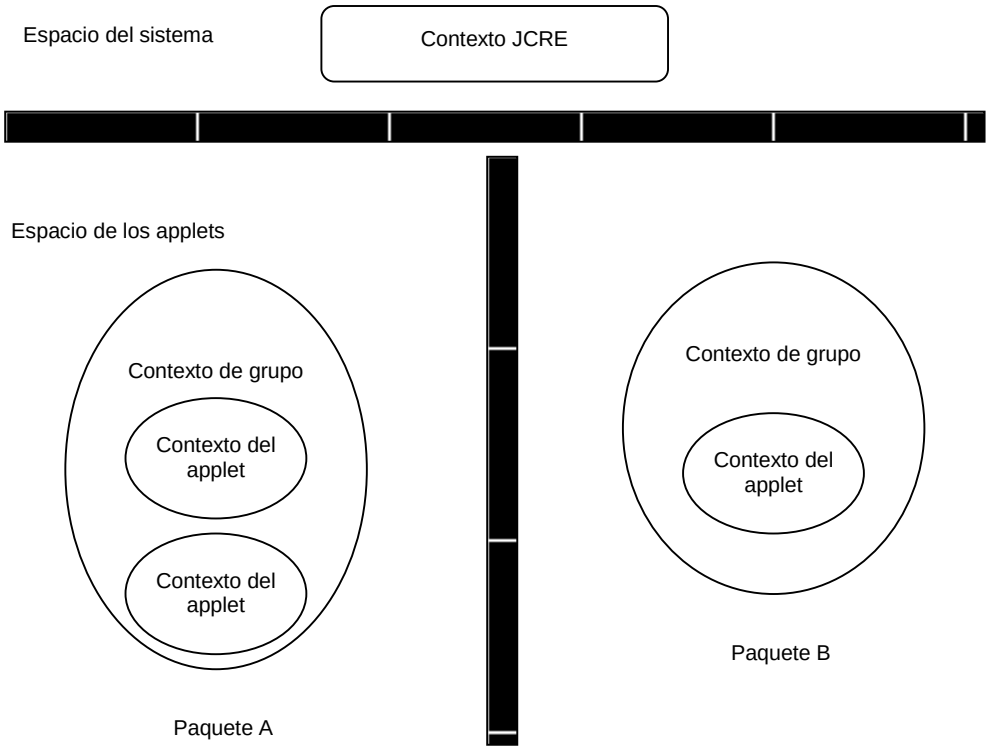


Figura 21: Divisiones del sistema de objetos de Java Card

4.7.2 LA PROPIEDAD DEL OBJETO

En cualquier instante, sólo hay un contexto activo en la máquina virtual: el contexto JCRE o el contexto de grupo de un applet. Cuando se crea un objeto nuevo, se le asigna un contexto propio, el contexto actualmente activo. El objeto puede ser accedido desde ese contexto, es decir, por todas las instancias de applets que pertenezcan a su contexto propio. También se dice que un objeto pertenece al applet activo del contexto actual, cuando se instancia a ese objeto. Si el contexto JCRE es el contexto actualmente activo, el objeto pertenece al JCRE.

En los applets, los arrays de tipos primitivos se pueden inicializar cuando se declaran. El convertidor crea e inicializa tales arrays estáticos (ver sección 4.1.3). Debido a que se crean estáticamente antes de que cualquier instancia de un applet se instancie en la tarjeta, se puede asignar la propiedad de estos arrays a cualquier instancia de un applet que pertenezca a su paquete de definición. Cualquier applet de este paquete puede acceder a estos arrays. Es decir, el contexto propio de estos arrays, es el contexto de grupo del paquete.

4.7.3 EL ACCESO A LOS OBJETOS

Cuando se accede a un objeto, están asegurados los controles de acceso del lenguaje Java. Por ejemplo, un método privado de una instancia no se puede invocar desde fuera del objeto. Además, el contexto propio del objeto se compara con el contexto actualmente activo. Si los contextos no coinciden, se le deniega el acceso, y la comparación provoca una `SecurityException`. En el siguiente ejemplo se asume que el objeto `b` pertenece al contexto `A`. Las siguientes operaciones de acceso al objeto `b` desde el contexto `A'` haría que la máquina virtual de Java Card lanzase una `SecurityException`:

- Conseguir y ajustar los campos:
```
short a = b.field_a; //consigue el campo del objeto b
b.field_a = 5; //ajusta el campo del objeto b
```
- Invocar un método público de una instancia:
```
b.virtual_method_a();
```
- Invocar a una interfaz del método:
```
b.interface_method_a(); //si b es un tipo de interfaz
```
- Lanzarlo como una excepción:
```
throw b;
```
- Hacerle el cast de un tipo dado:
```
(givenType)b;
```
- Determinar si es de un tipo dado:
```
if (b instanceof givenType)
```
- Acceder al elemento de un array:
```
short a = b[0]; //si el objeto b es un tipo de array
b[0] = 5;
```
- Obtener la longitud del array:
```
int a = b.length; //si b es un tipo de array
```

4.7.4 EL ARRAY TRANSITORIO Y SU CONTEXTO:

Al igual que un objeto persistente, solamente se puede acceder a un array transitorio del tipo `CLEAR_ON_RESET` cuando el contexto actualmente activo es el contexto propio del array.

Los arrays transitorios del tipo `CLEAR_ON_DESELECT` son recursos específicos del applet. Solo se pueden crear cuando el contexto actualmente activo es el contexto del applet actualmente activo. Debido a que los applets del mismo paquete comparten un contexto de grupo, los arrays transitorios del tipo `CLEAR_ON_DESELECT` también se pueden acceder desde todos los applets de su contexto propio. Sin embargo, tales accesos solo se conceden si uno de esos applets es el applet actualmente seleccionado.

4.7.5 LOS CAMPOS ESTÁTICOS Y LOS MÉTODOS

Sólo las instancias de clases pertenecen a los contextos y no las clases mismas (clases no instanciadas o estáticas). No se llevan a cabo comprobaciones de contexto

cuando se accede a un campo estático o cuando se invoca a un método estático. Es decir, los campos y métodos estáticos son accesibles desde cualquier contexto. Por ejemplo, cualquier applet puede invocar al método estático `throwIt` de la clase `ISOException`:

```
If (apdu_buffer[ISO7816.OFFSET_CLA] != EXPECTED_VALUE)
ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
```

Por supuesto, las reglas de acceso de Java todavía se aplican a los campos y métodos estáticos. Por ejemplo, los campos y métodos estáticos con el modificador `private` solo son visibles desde sus clases de definición.

Cuando se invoca a un método estático, se ejecuta en el contexto del llamante. Esto sugiere que aquellos objetos creados dentro de un método estático se les asigna el contexto del llamante (el contexto actualmente activo).

Los campos estáticos son accesibles desde cualquier contexto. Sin embargo, los objetos (incluyendo los arrays) referenciados en campos estáticos, son como objetos ordinarios. Tales objetos pertenecen al applet (o al JCRE) que los creó, y se aplican las reglas estándares de acceso al firewall.

4.8 LA COMPARTICIÓN DE OBJETOS A TRAVÉS DE CONTEXTOS.

El applet firewall confina las acciones de un applet a su contexto designado. El applet no puede llegar más allá de su contexto. Esto significa que no podrá acceder a objetos que pertenecen al JCRE o a otro applet de un contexto diferente. Pero en situaciones donde los applets necesiten ejecutarse de manera conjunta, la tecnología Java Card provee mecanismos bien definidos y seguros que se cumplen por los siguientes medios:

- Privilegios del JCRE.
- Objetos de punto de entrada al JCRE.
- Arrays globales.
- Interfaces que pueden ser compartidas.

Esencialmente, los mecanismos de compartición permiten a un objeto el acceso a objetos que pertenecen a otro contexto, bajo ciertas condiciones.

4.8.1 CAMBIO DE CONTEXTO

Se recuerda que solo hay un contexto activo en cualquier instante de la ejecución de la máquina virtual de Java Card. La máquina virtual comprueba todos los accesos a objetos para determinar si se permite el acceso. Normalmente, se deniega el acceso si el contexto propio perteneciente al objeto al que se intenta acceder no es el mismo que el contexto actualmente activo. Cuando se aplica el mecanismo de compartición, la máquina virtual de Java Card permite el acceso llevando a cabo un cambio de contexto.

Los miembros de un objeto consisten en instancias de métodos y campos. El acceso a instancias de campos de un objeto de un contexto diferente no causa un cambio

de contexto. Solo el JCRE puede acceder a las instancias de campos de un objeto de un contexto diferente. Los cambios de contexto solo ocurren en la invocación y en la vuelta de las instancias de métodos de un objeto que pertenece a un contexto diferente, así como en las excepciones de salida desde aquellos métodos.

Durante el cambio de contexto debido a la invocación de un método, se salva el contexto actual y el nuevo contexto llega a ser el contexto actualmente activo. El método invocado está ejecutándose ahora en el nuevo contexto y tiene los derechos de acceso del contexto actual. Cuando el método se sale mediante una salida normal o por una excepción, se restaura el contexto original (el contexto del llamante) y pasa a ser el contexto actualmente activo. Por ejemplo, si un applet invoca a un método de un objeto de entrada al JCRE, el cambio de contexto se produce desde el contexto del applet, al contexto JCRE. Cualquier objeto creado por el método invocado está asociado al contexto JCRE.

Debido a que las invocaciones de métodos se pueden anidar, los cambios de contexto también se pueden anidar en la máquina virtual de Java. Cuando la máquina virtual de Java Card empieza a funcionar después de un reset, el contexto JCRE siempre es el contexto actualmente activo.

4.8.2 PRIVILEGIOS DEL JCRE

En la plataforma Java Card, el JCRE se comporta como el director de la tarjeta. Debido a que es el contexto del sistema, el contexto JCRE tiene privilegios especiales. Puede invocar un método de cualquier objeto o acceder a la instancia de un campo de cualquier objeto de la tarjeta. Tales privilegios del sistema, permite al JCRE el control de los recursos del sistema y la gestión de los applets. Por ejemplo, cuando el JCRE recibe una APDU de comando, invoca al método `select`, `deselect`, o `process` del applet actualmente seleccionado.

Cuando el JCRE invoca al método de un applet, el contexto JCRE se cambia por contexto del applet. El applet toma ahora el control y pierde los privilegios del JCRE. Cualquier objeto creado después del cambio de contexto pertenece al applet y está asociado al contexto del applet actual. Cuando el método finaliza, se restaura el contexto JCRE.

4.8.3 LOS OBJETOS DE PUNTO DE ENTRADA AL JCRE

El JCRE puede acceder al contexto de cualquier applet, pero no se permite que los applets accedan al contexto JCRE. Una computadora segura debe tener un camino para los usuarios sin privilegios (aquellos que están restringidos a un subconjunto de recursos) para solicitar los servicios del sistema que sean llevados a cabo por rutinas del sistema con privilegios. En la plataforma Java Card, este requisito se cumple usando los objetos de punto de entrada al JCRE.

Los objetos de punto de entrada al JCRE son objetos normales que pertenecen al contexto JCRE, pero que han sido marcados como contenedores de métodos de entrada al sistema. Normalmente el firewall debería proteger por completo a tales objetos del acceso de cualquier applet. Con la designación de punto de entrada se permite invocar a los métodos públicos de tales objetos desde cualquier contexto. Cuando esto ocurre, se

lleva a cabo un cambio de contexto hacia el contexto JCRE. De este modo, estos métodos son puertas o pasarelas a través de las cuales los applets solicitan servicios privilegiados del JCRE. Hay que señalar que sólo son accesibles los métodos públicos de los objetos de punto de entrada al JCRE a través del firewall. Los campos de estos objetos están aún protegidos por el firewall.

El objeto `APDU`, es quizás el objeto de punto de entrada al JCRE más frecuentemente utilizado. Hay dos categorías de objetos de punto de entrada al JCRE:

- Los objetos temporales de punto de entrada al JCRE: como todos los objetos de punto de entrada al JCRE, los métodos de los objetos temporales de punto de entrada al JCRE se pueden invocar desde cualquier contexto. Sin embargo las referencias a estos objetos no se pueden guardar en las variables de clase, las instancias de variables ni en los arrays de campos (incluyendo los arrays transitorios) de un applet. El JCRE detecta y limita los intentos de guardar las referencias de estos objetos como parte de las funciones del firewall para prevenir la reutilización no autorizada de esas referencias. El objeto `APDU` y todos los objetos de excepción pertenecientes al JCRE son ejemplos de objetos de punto de entrada al JCRE.

- Los objetos permanentes de punto de entrada al JCRE: como todos los objetos de punto de entrada al JCRE, los métodos de los objetos permanentes de punto de entrada al JCRE se pueden invocar desde cualquier contexto. Además, las referencias a estos objetos se pueden guardar y reutilizar libremente. Las instancias de los AID's pertenecientes al JCRE, son ejemplos de los objetos permanentes de punto de entrada al JCRE. El JCRE crea una instancia de AID para encapsular el AID de un applet cuando se crea la instancia de ese applet.

Solo el JCRE puede designar por sí mismo objetos de punto de entrada y si son temporales o permanentes. Los implementadores del JCRE son responsables de implementar el JCRE, incluyendo el mecanismo por el que: se designan los objetos de punto de entrada al JCRE y llegan a ser temporales o permanentes.

4.8.4 ARRAYS GLOBALES

Los objetos de punto de entrada permiten a los applets acceder a los servicios particulares del JCRE, invocando sus respectivos métodos de punto de entrada. Los datos encapsulados en los objetos de punto de entrada al JCRE no son directamente accesibles por los applets. Pero en la plataforma Java Card, la naturaleza global de algunos datos, requiere que sean accesibles desde cualquier applet y contexto JCRE.

Para acceder a los datos globales de una forma flexible, el firewall permite que el JCRE designe arrays de tipos primitivos como globales. Los arrays globales suministran esencialmente, un buffer de memoria compartida para aquellos datos a los que se puede acceder desde cualquier applet y desde el JCRE. Debido a que solo hay un contexto ejecutándose a la vez, el acceso sincronizado no es un problema.

Los arrays globales son un tipo especial de objeto de punto de entrada al JCRE. El applet firewall permite que desde cualquier contexto se acceda a los campos públicos (los componentes de un array) de tales arrays. Los métodos públicos de los arrays

globales se tratan de la misma manera que los métodos de otros objetos de punto de entrada al JCRE. El único método de la clase array, es el método `equals`, que es heredado de la clase raíz `Object`. Como ocurre cuando se invoca cualquier método de un objeto de punto de entrada al JCRE, la invocación del método `equals` de un array global, hace que el contexto actual se cambie por el contexto JCRE.

Solo los arrays de tipos primitivos se pueden designar como globales, y solo el JCRE puede designar arrays globales. Todos los arrays globales deben ser objetos temporales de punto de entrada al JCRE. Por lo tanto, las referencias a estos arrays no pueden ser guardadas en variables de clase, instancias de variables, o los componentes de un array (incluyendo los arrays transitorios).

El único array global necesario en las API's de Java Card es el buffer APDU y el parámetro de array de bytes en el método `install` de un applet. Normalmente, una implementación de JCRE pasa el buffer APDU como el parámetro de array de bytes al método `install`. Debido a que cualquiera puede ver y acceder a los arrays globales, el JCRE limpia el buffer APDU siempre que se seleccione un applet o antes de que el JCRE acepte una nueva APDU de comando. Así se previene el que se puedan filtrar los datos delicados a otro applet a través del buffer APDU, que es global.

4.8.5 MECANISMO PARA COMPARTIR INTERFACES DE OBJETOS

La tecnología Java Card también permite compartir objetos entre applets, mediante el mecanismo para compartir interfaces de objetos.

4.8.5.1 Interfaz compartida

Una interfaz compartida es simplemente una interfaz que se extiende directa o indirectamente, de la interfaz `javacard.framework.Shareable`.

```
public interface Shareable {}
```

Esta interfaz es similar en concepto a la interfaz `Remote` usada en la facilidad RMI. La interfaz no define ni métodos ni campos. Su único propósito es ser extendida por otras interfaces y etiquetar a aquellas interfaces que tienen propiedades especiales.

Una interfaz compartida define un conjunto de métodos que están disponibles para otros applets. Una clase puede implementar cualquier número de interfaces que se pueden compartir y puede extender otras clases que implementen las interfaces que se pueden compartir.

4.8.5.2 Objeto de interfaz compartida

Un objeto de una clase que implementa una interfaz compartida se llama objeto de interfaz compartida (SIO = Shareable Interface Object). Para su propio contexto, un SIO es un objeto normal a cuyos campos y métodos se puede acceder. Para cualquier otro contexto, el SIO es una instancia de un tipo de interfaz compartida, y solo son accesibles los métodos definidos en la interfaz compartida. El firewall protege todos los campos y los otros métodos del SIO.

4.8.5.3 Conceptos que hay detrás del mecanismo de interfaces que se pueden compartir

Los applets guardan datos en objetos. Compartir datos entre applets significa que un applet realiza un objeto que está disponible para otros applets, con el fin de compartir los datos encapsulados en el objeto.

En el mundo orientado a objetos, el comportamiento de un objeto se expresa a través de sus métodos. El paso de mensajes, o la invocación de métodos, soporta la interacción y comunicación entre objetos. El mecanismo de interfaces que se pueden compartir, permite a los applets el envío de mensajes evitando la vigilancia del firewall. El propio applet crea un objeto de interfaz compartida e implementa los métodos definidos en la interfaz compartida. Estos métodos representan la interfaz pública del propio applet, a través de la cual cualquier applet puede enviar mensajes y por lo tanto acceder a los servicios provistos por este applet.

El escenario de compartición ilustrado en la siguiente figura, se puede describir como una relación cliente / servidor.

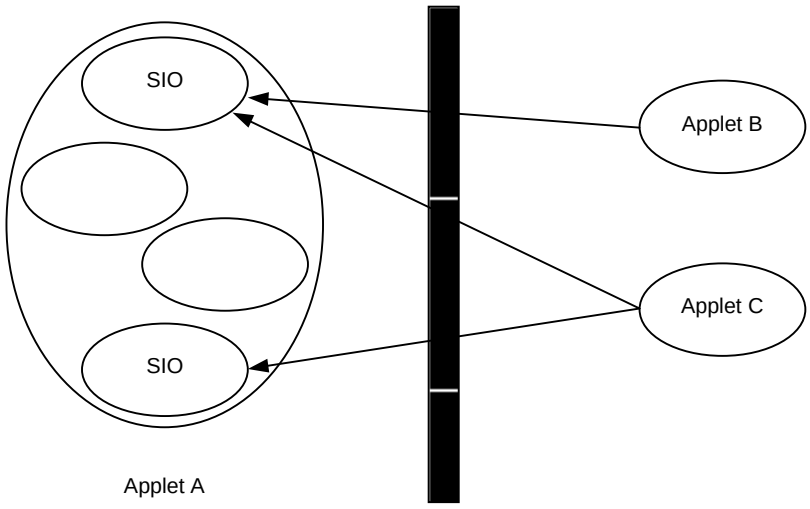


Figura 22: Mecanismo de objetos de interfaz compartida

El applet A (que provee SIO's) es un servidor, y el applet B y C (que usan los SIO's del applet A) son clientes. Quizás un applet sea un servidor para algunos applets y también un cliente para otros applets.

En el lenguaje de programación Java, una interfaz define un tipo de referencia que contiene una colección de firmas y constantes de métodos. Un applet cliente ve un SIO como un contenedor de un tipo de interfaz de que se puede compartir. No se muestra el tipo de clase del SIO que implementa la interfaz compartida. Es decir, solo se presentan al applet cliente los métodos definidos en la interfaz compartida; no se revelan las instancias de campos ni otros métodos. En este sentido, un applet servidor puede proveer un acceso controlado a los datos que quiere compartir.

Cuando interacciona con un applet cliente diferente, el applet servidor podría llevar un sombrero diferente. Esto requeriría que el applet servidor personalizase sus servicios según el applet cliente, sin tener la puerta abierta de par en par. Un applet servidor puede hacerlo definiendo múltiples interfaces. Cada interfaz declara los

métodos convenientes para un grupo de applets cliente. Si los métodos de las interfaces son distintos, un applet servidor podría elegir la clase, cada una de las cuales implementa una interfaz. Pero a menudo los servicios coinciden; un applet servidor puede definir una clase que implemente múltiples interfaces. Por lo tanto, un SIO de esa clase puede jugar muchos papeles.

4.8.5.4 Un ejemplo de compartición de objetos entre applets

Esta sección utiliza un applet wallet (cartera) y un applet air-miles que proporcionan un ejemplo de compartición de objetos entre applets. El applet wallet guarda dinero electrónico. El dinero se puede gastar para comprar cosas.

El applet air-miles proporciona incentivos para viajar. De forma similar al applet wallet, el applet air-miles también guarda valores (las millas que consigue el titular de la tarjeta). Por motivos de marketing, por cada dólar gastado usando el applet cartera, se le da una milla al applet air-miles.

Se supone que el applet wallet y el applet air-miles están en contextos diferentes (están definidos en paquetes diferentes). Los siguientes pasos indican como interaccionan en presencia del firewall.

1. El applet air-miles crea un objeto de interfaz compartida (SIO).
2. El applet wallet realiza una petición al SIO del applet air-miles.
3. El applet wallet va solicitando la suma de millas, invocando a un método de servicio del SIO.

En este caso, el applet air-miles es un servidor que otorga millas a petición del applet wallet (que es el cliente), como se muestra en la siguiente figura:

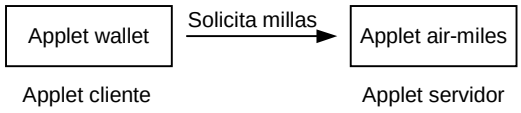


Figura 23: Compartición de objetos entre el applet wallet y el applet air-miles

Lo próximo es como implementar el applet wallet y el applet air-miles. Los ejemplos de código son breves (se omiten los métodos y campos no pertenecientes a la discusión y también se ignoran las comprobaciones de condiciones durante una transacción).

4.8.5.5 Crear un objeto de interfaz compartida

Para crear un SIO, el applet servidor (el applet air-miles) debe definir primero una interfaz compartida que extienda `javacard.framework.Shareable`.

```
package com.fasttravel.airmiles;
import javacard.framework.Shareable;
public interface AirMilesInterface extends Shareable {
     public void grantMiles (short amount);
}
```

Lo próximo que debe hacer el applet servidor, es crear una clase proveedora del servicio (una clase proveedora del servicio puede ser la misma clase del applet) que implemente la interfaz compartida. Entonces, el applet servidor puede crear uno o más objetos de la clase proveedora del servicio y puede compartir tales objetos (SIO) con otros applets de diferente contexto.

```
package com.fasttravel.airmiles;
import javacard.framework.*;
public class AirMilesApp extends Applet implements
AirMilesInterface {
      private short miles;
      public void grantMiles (short amount) {
            miles = (short)(miles + amount);
      }
}
```

Antes de que un cliente pueda solicitar un SIO, debe encontrar un camino para identificar al servidor. En la plataforma Java Card, cada instancia de applet está unívocamente identificada por un AID.

Se recuerda de la sección 4.5 que cuando se crea la instancia de un applet, se registra en el JCRE usando uno de los dos métodos `register`. El método sin parámetros registra al applet en JCRE usando el AID por defecto, definido en el fichero CAP. El otro método `register` permite al applet especificar otro AID diferente al AID por defecto. El JCRE encapsula los bytes del AID en un objeto AID (que pertenece al JCRE) y asocia ese objeto AID al applet. Durante la compartición del objeto, este objeto AID lo usa el cliente para especificar el servidor.

4.8.5.6 Solicitar un objeto de interfaz compartida

Antes de solicitar un SIO del applet servidor, el applet cliente debe obtener el objeto AID asociado al applet servidor. Para hacerlo, el applet cliente llama al método `lookupAID` de la clase `JCSystem`:

```
public static AID lookupAID (byte [] buffer, short offset, byte
length)
```

El applet cliente debe saber de antemano los bytes del AID del applet servidor, y suministrárselos al parámetro `buffer`. El método `lookupAID` devuelve el objeto AID (perteneciente al JCRE) del applet servidor o devuelve `null` si el applet servidor no está instalado en la tarjeta. Debido a que el objeto AID es un objeto permanente de punto de entrada al JCRE, el applet cliente puede solicitarlo una vez y guardarlo en una localización permanente para su uso posterior.

Seguidamente, el applet cliente llama al método `JCSystem.getAppletShareableInterfaceObject`, usando el objeto AID para identificar al servidor:

```
public static Shareable getAppletShareableInterfaceObject (AID
server_aid, byte parameter)
```

El applet servidor interpreta el segundo parámetro del método `getAppletShareableInterfaceObject`. Se puede utilizar para seleccionar un SIO, si el servidor tiene más de uno disponible. Alternativamente, el parámetro se puede

usar como un testigo de seguridad, que lleva información secreta compartida entre el servidor y el cliente.

En el método `getAppletShareableInterfaceObject`, el JCRE busca el applet servidor, comparando el `server_aid` con los AID's de los applets que están registrados en el JCRE. Si no se encuentra el applet servidor, el JCRE devuelve un `null`. Si lo encuentra, el JCRE invoca al método `getShareableInterfaceObject` del applet servidor.

```
public Shareable getShareableInterfaceObject (AID client_aid,
byte parameter)
```

Se advierte que en el método `getShareableInterfaceObject`, el JCRE cambia el primer argumento por el objeto `client_aid` y pasa el mismo parámetro de byte suministrado por el applet cliente. El applet servidor usa ambos parámetros para determinar si proveer servicios al applet que los ha solicitado, y si es así, qué SIO exportar.

El método `getShareableInterfaceObject` está definido en la clase base `javacard.framework.Applet`. La implementación devuelve `null` por defecto. Una clase de applet debe sobrescribir este método si intenta compartir cualquier SIO. A continuación se muestra como el applet air-miles implementa el método `getShareableInterfaceObject` (ver la sección 4.8.5.10):

```
public class AirMilesApp extends Applet
implements AirMilesInterface {
      short miles;
      public Shareable getShareableInterfaceObject
      (AID client_aid, byte parameter) {
            //autentifica al cliente, se explica más tarde
            //...
            //devuelve el objeto de interfaz compartida
            return this;
      }
      public void grantMiles (short amount) {
            miles = (short)(miles + amount);
      }
}
```

Cuando el applet servidor devuelva el SIO, el JCRE se lo entrega al applet cliente que lo solicitó. El proceso de solicitud de una interfaz compartida se muestra a continuación:

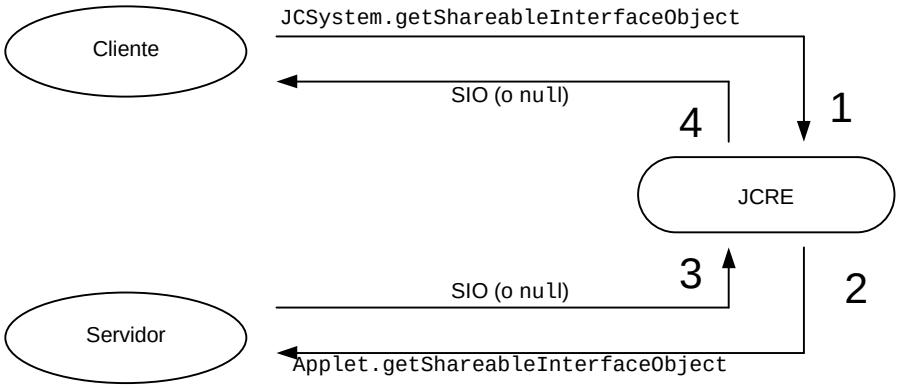


Figura 24: Solicitud de un SIO

4.8.5.7 Usar un objeto de interfaz compartida

Para permitir que el servidor devuelva cualquier tipo de interfaz compartida usando una sola interfaz, los métodos `JCSystem.getAppletShareableInterfaceObject` y `Applet.getShareableInterfaceObject` tienen que devolver el tipo `Shareable` (el tipo base de todos los objetos de interfaz compartida). El applet cliente debe hacer un cast (con un tipo apropiado de subinterfaz) al SIO devuelto, y guardarlo en una referencia de objeto de ese tipo. Por ejemplo, el applet wallet realiza un cast del tipo `AirMilesInterface`, al SIO devuelto:

```
AirMilesInterface sio = (AirMilesInterface)
JCSystem.getAppletShareableInterfaceObject (server_aid,
parameter);
```

Después de que el applet cliente reciba el SIO, invoca los métodos de la interfaz compartida para acceder a los servicios del servidor. Sin embargo, el applet cliente solo puede ver los métodos definidos en la interfaz compartida.

Por ejemplo, en el código anterior, aunque el `sio` realmente apunte al applet air-miles (su clase de applet implementa el `AirMilesInterface`), todas las instancias de campos y las interfaces de métodos no compartidas (como los métodos `process`, `select`, y `deselect`) están protegidas por el firewall.

A continuación de presenta un ejemplo de cómo el applet wallet solicita las millas en una transacción de débito.

```
package com.smartbank.wallet;
import javacard.framework.*;
import com.fasttravel.airmiles.AirMilesInterface;
public class WalletApp extends Applet {
     private short balance;
     //asignado previamente o en la personalización
     //del applet
     private byte[] air_ miles_aid_bytes =
     SERVER_AID_BYES;
     //el método process llama a este método cuando se
     //recibe una APDU DEBIT de comando
     private void debit (short amount) {
```

```
            if (balance < amount)
                  ISOException.throwIt(SW_EXCEED_BALANCE);
            //actualiza el balance
            balance = (short)(balance – amount);
            //pregunta al servidor para que conceda millas
            requestMiles(amount);
      }
      private void requestMiles (short amount) {
            //obtiene el objeto AID del servidor
            AID air_miles_aid = JCSystem.lookupAID(
            air_miles_aid_bytes, (short)0, (byte)
            air_miles_aid_bytes.length);
            if (air_miles_aid == null)
            ISOException.throwIt(SW_AIR_MILES_APP_NOT_EXIST);
            //solicita el SIO del servidor
            AirMilesInterface sio = (AirMilesInterface)
            (JCSystem.getAppletShareableInterfaceObject
            (air_miles_aid, SECRET));
            if(sio == null)
            ISOException.throwIt(SW_FAILED_TO_OBTAIN_SIO);
            //pregunta al servidor para conceder las millas
            sio.grantMiles(amount);
      }
}
```

Cuando ocurre un error, un applet puede lanzar una `ISOException` invocando al método estático `throwIt`. El método `throwIt` lanza el objeto `ISOException` perteneciente al JCRE. Tal objeto es un objeto de punto de entrada al JCRE y se puede acceder a él desde cualquier contexto de un applet.

4.8.5.8 Cambios de contexto durante la compartición de objetos

El JCRE, el applet cliente, y el applet servidor residen en contextos separados. Los cambios de contexto deben ocurrir para permitir la compartición de objetos. El applet cliente llama al método `JCSystem.getAppletShareableInterfaceObject` para solicitar un SIO. Un mecanismo interno del método cambia el contexto del applet cliente por el contexto JCRE. Entonces, el JCRE invoca al método `getShareableInterfaceObject` del applet servidor. Tal invocación provoca otro cambio de contexto para que el contexto del applet servidor sea el actual. A la vuelta de los dos métodos, se restaura el contexto del applet cliente.

Seguidamente, el applet cliente puede solicitar servicio del applet servidor invocando a uno de los métodos de interfaz compartida del SIO recibido. Durante la invocación, la máquina virtual de Java Card lleva a cabo un cambio de contexto. El contexto del applet servidor es el contexto que llega a ser el actualmente activo.

Debido a que la ejecución está ahora en el contexto del applet servidor, el código consigue el acceso a los recursos protegidos del applet servidor (instancias de campos, otros métodos y hasta objetos que pertenecen al contexto del applet servidor). Cuando se completa el método de servicio, se restaura el contexto del applet cliente. Los cambios de contexto durante la compartición de objetos, se ilustran en la siguiente figura:

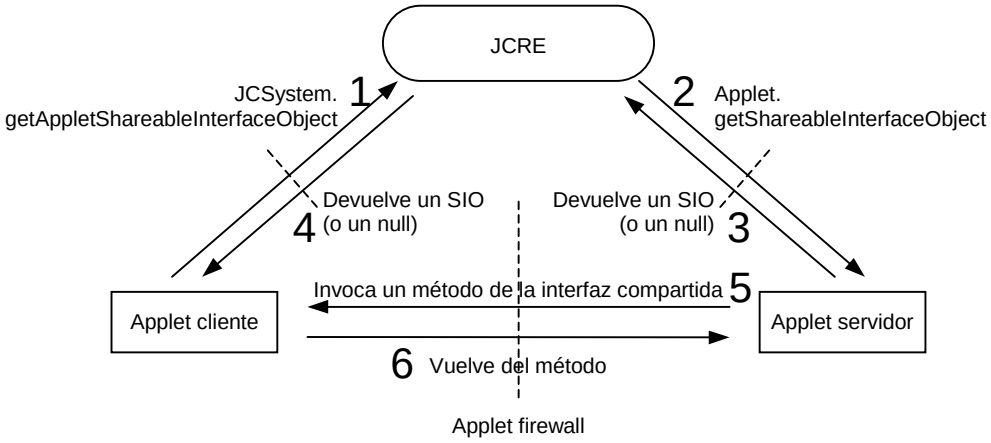


Figura 25: Cambio de contexto durante la compartición de objetos

4.8.5.9 Tipos de parámetros y tipos de valores de retorno, de los métodos de interfaz compartida

En el lenguaje de programación Java, los parámetros del método y los valores de retorno pueden ser de tipos primitivos o referencias. Pero en la plataforma Java Card, el paso de objetos (incluyendo arrays) como parámetros o valores de retorno en un método de interfaz compartida está permitido de manera limitada. Por ejemplo, si el applet wallet pasa uno de sus propios objetos como un parámetro en el método `grantMiles`, el applet firewall impide que el applet air-miles acceda a ese objeto. Igualmente, si el applet air-miles devuelve a uno de sus objetos como valor de retorno, el firewall impide que el applet wallet acceda a ese objeto. Aunque esto quizás parezca incómodo, es el applet firewall el que realmente realiza este trabajo.

Para evitar este problema, se pueden pasar los siguientes tipos de valores a un método de interfaz compartida como parámetros y valores de retorno:

- Valores primitivos: éstos se pasan fácilmente a la pila. Los tipos primitivos soportados en la plataforma Java Card son `boolean`, `byte`, `short`, y (opcionalmente) `int`.
- Campos estáticos: los campos públicos estáticos son accesibles desde cualquier contexto. Sin embargo, los objetos referenciados por estos campos estáticos están protegidos por el firewall.
- Objetos de punto de entrada al JCRE: se pueden acceder a los métodos públicos de estos objetos desde cualquier contexto.
- Arrays globales: se pueden acceder a ellos desde cualquier contexto. Por ejemplo, el buffer APDU se puede utilizar para este propósito.
- SIO's: se puede acceder a los métodos de interfaz compartida de estos objetos desde cualquier contexto. Un SIO que vuelve de la llamada realizada por un cliente, permite al contexto del servidor regresar al contexto del cliente para obtener algún dato o servicio. Sin embargo, el desarrollador debería ser cuidadoso para evitar los cambios excesivos de contexto (que quizás reduzca el rendimiento) y la anidación profunda de

cambios de contexto (que quizás cause un crecimiento del espacio requerido para la pila).

El código de la próxima sección muestra un ejemplo del paso de objetos y arrays a través del firewall.

4.8.5.10 Autenticar un applet cliente

Para prevenir que un cliente no autorizado consiga acceder a los datos protegidos, el applet servidor debería autenticar al applet cliente antes de conceder un SIO y antes de ejecutar un método de servicio del SIO.

Para determinar si se concede un SIO, el servidor puede comprobar el AID del solicitante (el applet cliente). Por ejemplo, el applet air-miles puede imponer las siguientes comprobaciones en el método `getShareableInterfaceObject`.

```
public class AirMilesApp extends Applet implements
AirMilesInterface {
      public Shareable getShareableInterfaceObject (AID
      client_aid, byte parameter) {
            //se asume que los bytes del AID ya son
            //conocidos
            if (client_aid.equals (wallet_app_aid_bytes,
            (short)0, (byte)(wallet_app_aid_bytes.length))
            == false) return null;
            //examina la información secreta para
            //autentificar al applet wallet
            if (parameter != SECRET) return null;
            //concede el SIO
            return (this);
      }
}
```

Cuando un método de interfaz compartida se invoca de nuevo, el servidor debería verificar el applet cliente otra vez. Esta precaución es necesaria porque el applet cliente que originalmente solicita el SIO podría romper el contrato y compartir el SIO con una tercera parte que no tiene el permiso oportuno. El applet servidor debe excluir este escenario para proteger los datos delicados de un cliente que no es de confianza. Para averiguar el AID del llamante real, el applet servidor puede invocar al método `JCSystem.getPreviousContextAID` (explicado en la sección 4.8.5.11). El código para detectar la identidad del llamante se añade al método `grantMiles` de la siguiente manera:

```
public void grantMiles (short amount) {
      //consigue el AID del llamante
      AID client_aid = JCSystem.getPreviousContextAID();
      //comprueba si el applet wallet ha invocado a este método
      if (client_aid.equals(wallet_app_aid_bytes, (short)0,
      (byte)(wallet_app_aid_bytes.length)) == false)
      ISOException.throwIt(SW_UNAUTHORIZED_CLIENT);
      //concede las millas
      miles = (short)(miles + amount);
}
```

La seguridad en el plan de autenticación presentado en el código anterior requiere que se controle la carga del applet bajo estrictas medidas de seguridad para prevenir la suplantación de un applet usando el AID de otro applet. Quizás tal plan no sea suficiente para un applet que requiera un alto grado de seguridad. En este caso, se definen métodos adicionales de autenticación, tales como el intercambio de texto cifrado.

El siguiente código implementa un plan de autentificación llamado reto-respuesta (challenge-response) en el ejemplo del applet wallet y el applet air-miles. Cuando se invoca al método `grantMiles` , el applet air-miles genera una frase aleatoria de reto y se la envía al applet wallet. El applet wallet encripta el reto y devuelve la respuesta al applet air-miles. Verificando la respuesta, el applet air-miles autentifica al applet wallet y suma las millas solicitadas a su balance.

Para dar soporte a este plan, el método `grantMiles` se actualiza para aceptar dos parámetros adicionales (un objeto de autentificación y un buffer).

```
public interface AirMilesInterface extends Shareable {
      public void grantMiles (AuthenticationInterface
      authObject, byte[] buffer, short amount);
}
```

El applet air-miles autentifica al applet wallet invocando al método `challenge` del objeto de autenticación. El buffer se usa para pasar el reto y los datos de respuesta.

```
public class AirMilesApp extends Applet implements
AirMilesInterface {
      public void grantMiles (AuthenticationInterface authObject,
      byte[] buffer, short amount) {
            //genera una frase aleatoria de reto en el buffer
            generateChallenge(buffer);
            //reta al applet cliente
            //la respuesta se devuelve en el buffer
            authObject.challenge(buffer);
            //comprueba la respuesta
            if (checkResponse(buffer) == false)
                  ISOException.throwIt(SW_UNAUTHORIZED_CLIENT);
            miles = (short)(miles + amount);
      }
}
```

Hay que advertir que el objeto de autenticación lo crea el llamante (el applet wallet) y pertenece también al llamante. El applet firewall requiere que tal objeto sea un objeto SIO:

```
public interface AuthenticationInterface extends
Shareable {
      public void challenge(byte[] buffer);
}
public    class    WalletApp    extends    Applet    implements
AuthenticationInterface {
      public void challenge(byte[] buffer) {
            //consigue la respuesta
            //el reto y la respuesta se llevan en el buffer
            getResponse(buffer);
      }
```

```
      public void process(APDU apdu) {
            if (getCommand(apdu) == DEBIT) debit(apdu);
      }
      private void debit(APDU apdu) {
            short amount = getDebitAmount(apdu);
            //actualiza el balance
            balance = (short)(balance – amount);
            //pregunta al applet air-miles para que conceda
            //millas
            requestMiles(apdu.getBuffer(), amount);
      }
      private void requestMiles(byte[] buffer, short amount) {
            //obtiene el AID del objeto
            AID air_miles_aid = JCSystem.lookupAID
            (air_miles_aid_bytes, (short)0, (byte)
            air_miles_aid_bytes.length);
            //solicita el SIO del applet air-miles
            AirMilesInterface sio = (AirMilesInterface)
            (JCSystem.getAppletShareableInterfaceObject
            (air_miles_aid, SECRET));
            //pregunta al applet air-miles para que conceda
            //las millas
            sio.grantMiles(this, buffer, amount);
      }
}
```

En el código de ejemplo, el buffer usado para pasar los datos del reto y de la respuesta, es el buffer APDU. El buffer APDU es un array global al que se puede acceder desde cualquier contexto.

4.8.5.11 Método `getPreviousContextAID`

Durante la compartición de objetos, un servidor puede averiguar el AID del applet llamante invocando al método `JCSystem.getPreviousContextAID`:

```
public AID getPreviousContextAID()
```

Este método devuelve el objeto AID (perteneciente al JCRE) asociado a la instancia del applet que estuvo activa en el momento del último cambio de contexto. En el segundo código de ejemplo de la sección 4.8.5.10, cuando el applet wallet llama al método de interfaz compartida `grantMiles`, ocurre un cambio de contexto. El método `getPreviousContextAID` devuelve el AID del applet wallet que estuvo activo antes del cambio de contexto.

Ahora se considera un escenario más complejo, como se muestra a continuación:

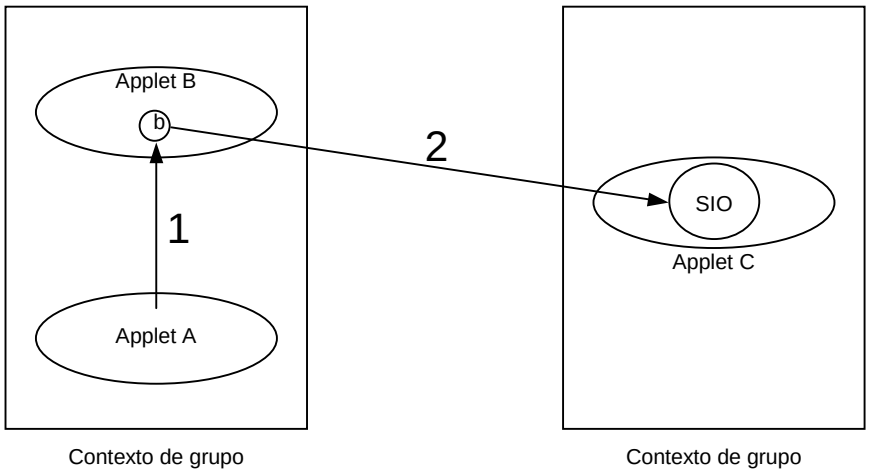


Figura 26: Compartición de objetos entre los applets A, B y C

Se supone que las dos instancias de applet, `A` y `B`, comparten un contexto de grupo. No ocurre ningún cambio de contexto si el applet `A` llama a un método del objeto `b` (que pertenece al applet `B`). Este acto se permite a pesar de que el objeto `b` es un SIO.

Cuando el objeto `b` accede a un SIO del applet `C`, la ejecución se mueve dentro de un nuevo contexto. El applet activo en el último cambio de contexto fue el applet `B`, y por ello, el método `getPreviousContextAID` devuelve el AID del applet `B`.

4.8.5.12 Pasos a seguir para compartir y usar objetos compartidos

Para concluir, el proceso de compartir objetos entre un applet servidor y un applet cliente, se resume aquí:

1. Si un applet servidor `A` quiere compartir un objeto con otro applet, primero define una interfaz compartida `SI` (Shareable Interface). Una interfaz compartida extiende la interfaz `javacard.framework.Shareable`. Los métodos definidos en la interfaz compartida `SI` representan los servicios que el applet `A` desea hacer accesible a otros applets.

2. Entonces el applet `A` define una clase `C` proveedora de servicios, que implementa la interfaz compartida `SI`. `C` provee realmente implementaciones para los métodos definidos en `SI`. Quizás `C` defina otros métodos y campos, pero éstos están protegidos por el applet firewall. Solamente los métodos definidos en `SI` son accesibles por otros métodos.

3. El applet `A` crea una instancia del objeto `O` de la clase `C`. `O` pertenece al applet `A`, y el firewall permite que `A` acceda a los campos y métodos de `O`.

4. Si el applet cliente `B` quiere acceder al objeto `O` del applet `A`, invoca el método `JCSystem.getAppletShareableInterface` para solicitar un objeto de interfaz compartida del applet `A`.

5. El JCRE busca el applet `A` en su tabla interna de applets. Cuando lo encuentra, invoca al método `getShareableInterfaceObject` del applet `A`.

6. El applet `A` recibe la solicitud y determina si quiere compartir el objeto `O` con el applet `B`. Si el applet `A` está de acuerdo en compartirlo con el applet `B`, `A` responde a la solicitud con una referencia de `O`.

7. El applet `B` recibe del applet `A` la referencia del objeto, le hace un cast del tipo `SI`, y la guarda en una referencia de objeto `SIO`. Aunque el `SIO` realmente se refiere al objeto `O` de `A`, `SIO` es del tipo `SI`. Solamente los métodos de la interfaz compartida definidas en el `SI`, son visibles por `B`. El firewall evita que `B` acceda a los otros campos y métodos de `O`. El applet `B` puede solicitar un servicio del applet `A`, invocando a uno de los métodos de interfaz compartida del `SIO`.

8. Antes de llevar a cabo el servicio, el método de interfaz compartida puede autentificar al cliente (`B`), para determinar si le concede el servicio.

5 CRIPTOGRAFÍA EN EL ENTORNO JAVA CARD

Hay ciertos campos de aplicación de las tarjetas inteligentes en los que se requiere cierto nivel de seguridad (por ejemplo, las tarjetas destinadas a operaciones bancarias ó a control de acceso). Por ello, en este capítulo se explica cómo obtener cierto grado de seguridad haciendo uso de la criptografía en los applets y teniendo en cuenta las características de seguridad de la tecnología Java Card.

5.1 USO DE LA CRIPTOGRAFÍA EN LA PROGRAMACIÓN

Además de su uso como lugar de almacenamiento de datos seguro y como dispositivo de procesado, las smart cards pueden actuar como prueba para una autorización y como un módulo de encriptación. Para tales aplicaciones, las funciones criptográficas de las smart cards son de gran importancia.

5.1.1 REPASO DE CONCEPTOS DE CRIPTOGRAFÍA

La criptografía es el arte y la ciencia de escribir de forma secreta para mantener seguros los mensajes de interés. Esto tiene que ver con la codificación y decodificación de la información. El criptoanálisis es lo contrario de la criptografía. Es el arte y la ciencia de descifrar la información secreta codificada. El término criptología se refiere a una rama de las matemáticas que abarca la criptografía y el criptoanálisis.

Aunque la criptografía no asegura por sí misma la seguridad, los sistemas y los programas se benefician a menudo del uso de la criptografía. Cuando se aplica correctamente, la criptografía provee los siguientes objetivos de seguridad:

- La confidencialidad, que asegura la privacidad. Evita que las personas no autorizadas puedan ver la información secreta.
- La integridad, que asegura que los datos son correctos. Esto significa que los datos originales no se pueden cambiar o sustituir sin saberlo.
- La autenticación, que asegura que la identidad es la verdadera. Así se sabe si las personas con las que se comunica electrónicamente son realmente las que dicen ser.

Esta sección explica conceptos básicos de criptografía que son necesarios para entender como usar las API's de criptografía de Java Card.

5.1.1.1 Codificación y decodificación

Un método común para proteger la información secreta es codificarla en el punto de envío y decodificarla en el punto de recepción. Un mensaje es un texto plano. La codificación es el proceso de disfrazar un mensaje para ocultar su significado. El mensaje codificado se llama texto cifrado. La decodificación es el proceso contrario al de codificación. Toma el texto cifrado y lo traduce, obteniendo el texto plano original. El proceso de codificación y de decodificación se muestra en la siguiente figura:

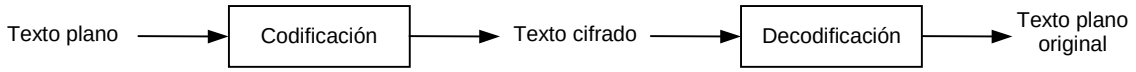


Figura 27: Proceso de codificación y decodificación

El algoritmo matemático que se aplica para llevar a cabo la transformación se le llama algoritmo de cifrado. Generalmente, hay dos algoritmos afines: uno para codificar y otro para decodificar.

Los buenos algoritmos de cifrado usan claves para codificar y decodificar los datos. Conceptualmente, se puede pensar en una clave como si fuera un valor secreto, como una contraseña o un PIN de una tarjeta bancaria. En la práctica, una clave solo es una secuencia de números. Una clave es un parámetro para la fórmula matemática del algoritmo de cifrado, así que si se codifica con el mismo texto plano y diferentes claves, se obtienen textos cifrados diferentes. A la inversa, el texto plano se puede recuperar a partir del texto cifrado usando la clave apropiada. El uso de claves en un algoritmo de cifrado permite publicar el algoritmo criptográfico sin perder seguridad. El secreto de los datos codificados se consigue manteniendo guardada la clave secreta.

Hay dos tipos generales de algoritmos de cifrado: los simétricos y los asimétricos. Los algoritmos de cifrado simétricos son algoritmos en los que la clave de codificación se puede calcular a partir de la clave de decodificación y viceversa. En la mayoría de los casos, la clave de codificación es la clave de decodificación. Por ello, se puede aplicar la misma clave para codificar y para decodificar.

La mayoría de los algoritmos más comunes de cifrado que se utilizan son DES y DESede. DES representa un estándar de codificación de datos (Data Encryption Standard). Es un estándar mundial desde hace 20 años. Las claves usadas en DES son de un tamaño de 56 bits (que se almacenan en 8 bytes). La codificación y la decodificación se llevan a cabo usando la misma clave.

DESede se conoce mejor como triple DES. Resulta de tres operaciones del algoritmo de cifrado, conectadas en cascada alternando la codificación y la decodificación, como se muestra a continuación:

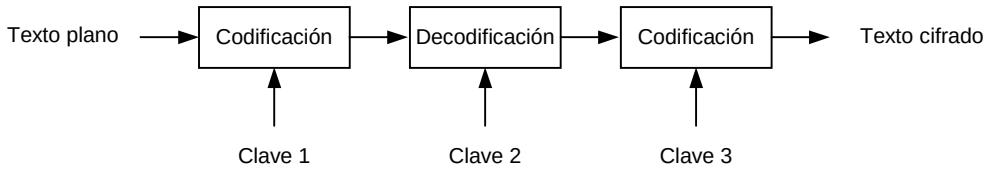


Figura 28: Proceso de codificación del algoritmo DESede

La decodificación del texto cifrado con DESede se realiza invirtiendo el orden de las operaciones, es decir, decodificando – codificando – decodificando:

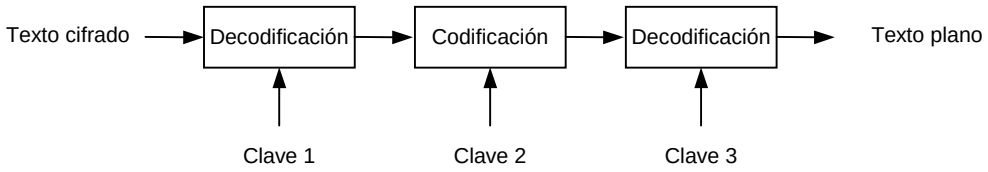


Figura 29: Proceso de decodificación del algoritmo DESede

En lugar de usar una clave de 8 bytes como en el algoritmo de cifrado de DES, DESede aplica tres claves. Por ello es considerablemente más seguro que DES. DESede viene con algunas variaciones, de acuerdo con la elección de las claves. Una versión utiliza tres claves DES distintas. Otra versión usa una clave de tamaño doble, donde la clave 1 y 3 en el algoritmo DESede son idénticas. Si las tres claves son iguales, la seguridad de DESede es la misma que la seguridad ofrecida por DES.

Los algoritmos de cifrado simétricos necesitan que todos los participantes guarden y distribuyan la clave secreta de forma apropiada. Si otros descubren la clave, el sistema entero se puede romper fácilmente.

Los algoritmos de cifrado asimétricos están diseñados para suplir las deficiencias de los algoritmos de cifrado simétricos. Estos algoritmos de cifrado usan dos claves diferentes: una clave pública que se puede distribuir públicamente y una clave privada que se mantiene en secreto.

Como se muestra a continuación, un desconocido puede utilizar la clave pública, que está disponible para cualquiera, para codificar un mensaje Pero sólo las personas autorizadas, que posean la clave privada correspondiente, pueden decodificar el mensaje.

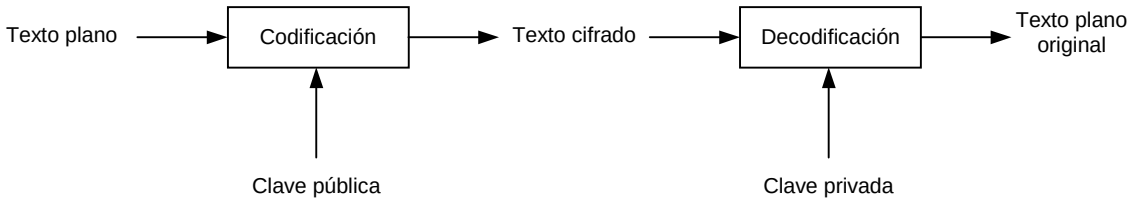


Figura 30: Codificación y decodificación con un algoritmo de cifrado asimétrico

En muchos algoritmos de cifrado asimétricos, el proceso inverso también funciona. Es decir, un mensaje se puede codificar usando la clave privada y lo puede decodificar cualquiera gracias a la clave pública.

El algoritmo RSA (cuyo nombre se debe a los nombres de sus tres inventores, Ronald Rivest, Adi Shamir y Leonard Adleman) es el algoritmo asimétrico mejor conocido y más versátil, y que está actualmente en uso.

Desde el punto de vista computacional, los algoritmos de cifrado asimétricos hacen un uso más intensivo del procesador que los algoritmos de cifrado simétricos. Por ello no se suelen usar para codificar mensajes largos. Muchos protocolos de seguridad combinan los algoritmos de cifrado simétrico y asimétrico. Un ejemplo de ello es el protocolo SSL (Secure Socket Layer), que está soportado en la mayoría de los exploradores Web. En el comienzo de un diálogo SSL, todos los participantes

intercambian datos para ponerse de acuerdo en la clave de sesión. Inicialmente, los datos intercambiados se codifican utilizando un algoritmo de cifrado asimétrico ya que estos datos no son muy voluminosos. En el siguiente paso, se usa la clave de sesión en un algoritmo de cifrado simétrico para codificar y decodificar el resto del diálogo. Como su nombre indica, el tiempo de vida útil de la clave de sesión es de una sola sesión. La clave de sesión ya no es válida cuando finaliza la sesión. Para comenzar un diálogo nuevo, todos los participantes necesitan ponerse de acuerdo en la clave de sesión.

La tecnología de codificación (usando los algoritmos simétricos o asimétricos) cifra los mensajes en un formato secreto. Solamente las personas autorizadas, con ayuda de computadoras, pueden decodificar los datos. Cuando se usa correctamente, la codificación asegura la confidencialidad. La codificación también se puede usar conjuntamente con la asimilación de mensajes para asegurar la integridad y establecer la autentificación.

5.1.1.2 Asimilación de un mensaje

La asimilación de un mensaje es el resultado de una función de dispersión segura de un solo sentido, que toma datos de tamaño arbitrario y produce un valor de dispersión de longitud fija, como se muestra a continuación:

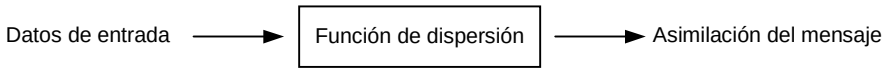


Figura 31: Cálculo de la asimilación de un mensaje

Una función de dispersión está diseñada para que sea fácil calcular un valor de dispersión pero que sea difícil realizar el cálculo inverso. La función también está diseñada para que sea raro calcular la misma dispersión a partir de dos cadenas de datos de entrada diferentes.

Basada en estas propiedades, la asimilación del mensaje es un mecanismo importante que ayuda a conseguir la integridad de los datos. Normalmente, se envía una asimilación del mensaje junto con los datos. El receptor calcula la asimilación del mensaje a partir de los datos recibidos y la compara con la asimilación del mensaje recibida. Cualquier diferencia indicaría un ataque o un error de transmisión.

Hoy en día se utilizan dos populares funciones de dispersión, el MD5 y el SHA_1. El MD5 fue publicado por Ronald Rivest en 1990/1991. Está basada en un algoritmo previo, el MD4. Tanto el MD4 y el MD5, generan un valor de dispersión de 128 bits. SHA, representa un algoritmo de dispersión muy seguro, produce una asimilación del mensaje que tiene una longitud de 160 bits. Primero fue publicado por MIST en 1992. Después del descubrimiento de ciertas debilidades, fue desarrollado un nuevo algoritmo mejorado, conocido como SHA-1. Hoy en día, los términos SHA y SHA-1 se usan para designar al mismo algoritmo.

Se puede usar una asimilación del mensaje para verificar la integridad de los datos. Aunque si la función de dispersión se conoce de antemano, cualquiera podría calcular el valor de dispersión a partir de los datos. Esto pone a la asimilación del mensaje en una posición vulnerable en caso de ataque. Un intruso podría interceptar los datos y la asimilación del mensaje, y reemplazarlos por nuevos datos y una asimilación

del mensaje calculada a partir de los datos nuevos. Si esto ocurriese, el receptor no tendría forma de detectar que los datos han sido cambiados.

Se ha desarrollado una variante de la función de dispersión de un solo sentido que requiere una clave. Produce un valor de dispersión que se basa en los datos de entrada y la clave. Esta idea se refiere a los mensajes de códigos de autentificación, o MAC (Message Authentication Code). Normalmente, un MAC se calcula codificando los datos usando para ello un algoritmo de cifrado simétrico con una clave dada. Un MAC se puede verificar si el que envía y el que recibe conocen la clave.

5.1.1.3 Firma digital

Una firma digital se calcula codificando una asimilación del mensaje, mediante el uso un algoritmo de cifrado asimétrico. Puede ayudar a verificar la identidad de la persona que envía los datos así como la integridad de los datos.

El que envía, calcula la asimilación del mensaje a partir de los datos que quiere enviar y los firma codificando la asimilación del mensaje con su propia clave privada. A la asimilación del mensaje codificada, se le llama firma. La siguiente figura muestra el proceso de firma:

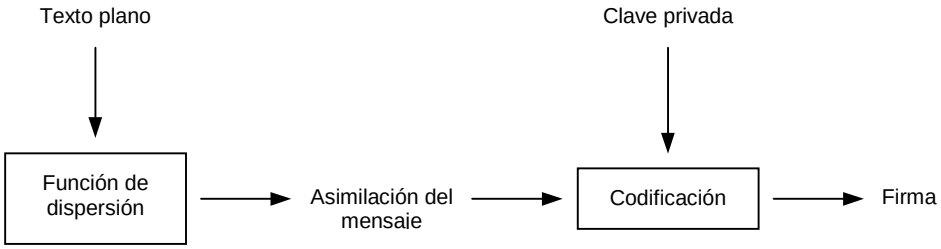


Figura 32: Proceso de generación de una firma

Entonces el transmisor envía los datos reales y la firma al receptor. El receptor verifica la firma, descodificándola con la clave pública del que envía. Así, el receptor obtiene la asimilación del mensaje. Entonces, el receptor puede calcular la asimilación del mensaje a partir de los datos recibidos y comparar las dos asimilaciones del mensaje. Si ambos valores coinciden, el receptor puede verificar que en efecto son los datos del transmisor y que no han sido alterados durante la transmisión. La siguiente figura lo explica:

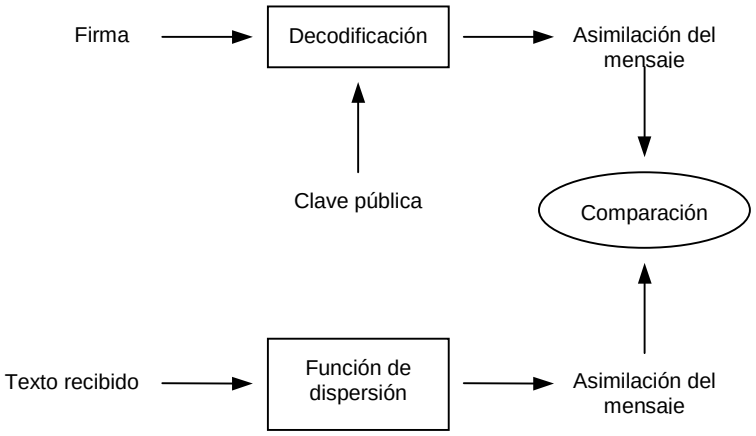


Figura 33: Proceso de verificación de una firma

5.1.1.4 Datos aleatorios

Los números aleatorios son cruciales en la criptografía. Normalmente se añaden a los datos de entrada como relleno para asegurar la unicidad de una sesión durante la autenticación o para generar claves criptográficas seguras. La seguridad de estos procedimientos criptográficos confían en la aleatoriedad de estos números.

Un verdadero número aleatorio no debería estar influenciado por condiciones externas, tales como la temperatura y el voltaje suministrado. Algunos computadores utilizan un generador hardware de números aleatorios, que mide un circuito electrónico inestable o procesos radiactivos u otros procesos físicos aleatorios. De este modo se puede conseguir la independencia de efectos externos. Tal hardware quizás no esté siempre disponible ni sea práctico. Por lo tanto, los computadores tienen que recurrir a los algoritmos software. Los resultados son generadores de números pseudoaleatorios que pueden producir salidas aleatorias bastante buenas para la mayoría de las aplicaciones. Los algoritmos pseudoaleatorios utilizan algoritmos deterministas. Para generar números aleatorios, se inicializan usando un número aleatorio (semilla). Un ejemplo de semilla puede ser la hora actual. Si se conocen el algoritmo y los valores iniciales, los números aleatorios son predecibles. Este es el porqué de que tales números aleatorios se denominen pseudoaleatorios.

5.1.2 LA CRIPTOGRAFÍA EN LAS APLICACIONES DE SMART CARD

De forma aproximada, hay dos motivos (que se explicarán más adelante) para que las aplicaciones de smart card usen la criptografía:

- Para asegurar la seguridad de las aplicaciones y para asegurar la seguridad de las comunicaciones de datos entre la tarjeta y el host.
- Para funcionar como un testigo seguro, a través del cual se puede incrementar la seguridad de otros sistemas.

5.1.2.1 Asegurar la seguridad de las aplicaciones

Considere que el applet wallet guarda dinero electrónico. El primer reto para construir un applet seguro, es la autentificación. Por ejemplo, la cartera debe verificar al comerciante a quien se le paga el dinero. Y todavía más importante, debe confirmar que

el dinero añadido al balance de la tarjeta viene de una fuente legítima. A menudo se requiere autenticación mutua. Tanto la cartera de la smart card como la entidad del host pueden aplicar mecanismos de autentificación criptográfica para asegurar que ellos están hablando con una parte autorizada para ello.

El segundo reto es la confidencialidad de los datos delicados que se deben proteger. Por ejemplo, en una transacción de crédito o de débito, los datos intercambiados entre el applet y el host podrían contener una suma de dinero en formato digital, el número de cuenta del banco, la contraseña para acceder a la cuenta, etc. Para asegurar la privacidad, los datos se pueden codificar antes de que se transmitan y se pueden decodificar en el receptor. Algunas veces es necesario guardar los datos en formato codificado para prevenir que alguien los consiga introduciéndose en la tarjeta.

El tercer reto es garantizar la integridad de los datos. Cuando se transfiere dinero desde la cuenta de un banco a un monedero electrónico, nadie quiere que alguien fisgonee en la transmisión, se dé cuenta de que involucra dinero, y transfiera la mitad de la cantidad (si usted es afortunado) a su propia cuenta y deje la otra mitad en la suya. Para ayudar a prevenir esto durante un intercambio de APDU's, una MAC calculada se sitúa en una APDU después de los datos reales:

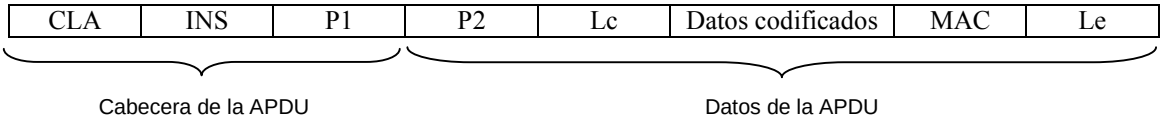


Figura 34: APDU de comando con una MAC

El valor MAC, se calcula a partir de los datos reales. Para proteger la privacidad, el MAC se puede calcular después de que los datos reales se codifiquen primero.

5.1.2.2 Funcionar como un testigo seguro

Cuando las smart cards funcionan dentro un aparato (que es el host), son ideales para actuar como comprobación de una autorización y como módulos de encriptación,. Estos aparatos pueden ser: teléfonos móviles, ordenadores conectados a una red, etc. Cuando se usan en tales aplicaciones, la criptografía se aplica para implementar requisitos funcionales. Por ejemplo, una aplicación de autenticación podría conservar una información confidencial (como una clave criptográfica) compartida con un servidor de red. Cuando el titular de la tarjeta solicita el acceso a la red, el servidor genera un reto en forma de número aleatorio, y se lo envía a la tarjeta como texto plano. Entonces, la aplicación de la tarjeta codifica el reto y le responde al servidor con el texto cifrado. El servidor decodifica los datos codificados y los compara con el reto que generó anteriormente. Si hay coincidencia, el servidor concede el acceso a la red al titular de la tarjeta. Este plan de autenticación se conoce como reto-respuesta. Una implementación real de reto–respuesta, requiere una mayor robustez que el simple concepto descrito en esta sección, para prevenir los ataques de seguridad.

5.1.2.3 Consideraciones de seguridad adicionales

Cuando se diseña una estrategia de seguridad, es importante evaluar los recursos de cálculo disponibles en la tarjeta. En general, no es posible desplegar mecanismos de computación orientados a la criptografía de datos voluminosos en la tarjeta. También es

importante darse cuenta de que la seguridad de un applet se debe evaluar en el contexto del conjunto de la seguridad de la infraestructura del sistema, además de considerar los beneficios de aplicar correctamente la criptografía. La infraestructura del sistema consiste en la tarjeta, la máquina host, y la red entera por la que estas máquinas están conectadas. Por lo tanto, cuando se diseña un applet, se debe considerar muchos factores del sistema entero para decidir si la criptografía es conveniente para la aplicación. Si es conveniente, se debe elegir qué mecanismos criptográficos son los más apropiados.

5.1.3 LAS API'S DE CRIPTOGRAFÍA DE JAVA CARD

Las API's de criptografía de Java Card (llamadas crypo API's para abreviar) se han revisado significativamente en la versión 2.1 de Java Card para cumplir con los requisitos de la regulación de exportación de Estados Unidos y para proveer un conjunto de características extensibles y flexibles.

Las crypto API's de Java Card están modeladas después de la arquitectura de criptografía de Java (JCA = Java Cryptography Architecture). El objetivo es estar exactamente alineado con la plataforma Java.

5.1.3.1 Principios de diseño.

Las crypto API's están diseñadas siguiendo estos principios:

- Independencia e interoperatividad de la implementación.
- Independencia del algoritmo y poder extender el algoritmo.

La independencia de la implementación y la independencia del algoritmo son objetivos complementarios. La intención es permitir a los usuarios de las AIP's la utilización de los servicios criptográficos, tales como las firmas digitales y la asimilación de mensajes, sin tener que preocuparse de las implementaciones o de los algoritmos usados para implementar estos servicios. La interoperatividad entre implementaciones permite a los applets escritos usando las crypto API's interoperar con todas las implementaciones de Java Card (que den un soporte correcto para la criptografía).

El poder extender el algoritmo, significa que se puedan añadir algoritmos criptográficos adicionales a las API's y que los algoritmos existentes en las API's se puedan borrar fácilmente de las API's. Tales cambios solo requieren una revisión de la versión anterior. De este modo las nuevas implementaciones de las API's serán compatibles con las implementaciones previas.

5.1.3.2 Arquitectura

La arquitectura de las crypto API's está diseñada para reflejar los principios de diseño a través de los siguientes mecanismos:

- Separando la interfaz de la implementación.
- Haciendo frecuente el uso de los métodos que vienen de fábrica.
- Estandarizando nombres para especificar algoritmos criptográficos.

- Definiendo API's que tengan poco peso (poca carga para la tarjeta).

Las crypto API's de Java Card (la mayoría son clases abstractas e interfaces) definen tipos de servicios criptográficos y la interfaz para acceder a los servicios. Un proveedor de JCRE's extiende una clase base de las API's para suministrar una implementación de un servicio criptográfico particular o para implementar el algoritmo simétrico o asimétrico de una interfaz. Por ejemplo, para proveer la función de dispersión usando el algoritmo MD5, el proveedor de un JCRE puede crear la implementación de una clase `MessageDigestMD5`, que extienda la clase `MessageDigest` de las API's. La clase `MessageDigestMD5` sobrescribe los métodos de la clase base para implementar el algoritmo de dispersión MD5 .

Sin conocer el nombre real de la clase de una implementación y donde se localiza, los applets usan los métodos que vienen de fábrica para solicitar un objeto de servicios de criptografía. Por ejemplo, para solicitar una asimilación del mensaje usando el algoritmo MD5, el código sería como el siguiente:

```
MessageDigest md5;
md5 = MessageDigest.getInstance (MessageDigest.ALG_MD5, false);
```

La instancia real devuelta por un método que viene de fábrica es normalmente algún descendiente de la clase por la que se preguntó. En el ejemplo, quizás se devuelva una instancia de `MessageDigestMD5`. Sin embargo, en su programa esto no importa. Se trata como una asimilación de un mensaje y se trabaja con métodos de la interfaz de la clase base. Como resultado, el applet no está atado a ninguna implementación particular.

Para soportar los métodos que vienen de fábrica, debe haber una convención de nombres que un applet pueda usar para especificar un algoritmo junto con los parámetros del algoritmo. Esta convención de nombres se consigue a través de los parámetros de selección. Los parámetros de selección son, esencialmente, constantes que especifican un nombre estándar para cada algoritmo criptográfico o para cada clave criptográfica y longitud de la clave. En el ejemplo de la asimilación del mensaje, la constante `MessageDigest.ALG_MD5` es un parámetro de selección que especifica el algoritmo MD5.

Además, los parámetros de selección proveen una forma flexible de extender los algoritmos criptográficos y los servicios. En el futuro, tales algoritmos se pueden soportar fácilmente añadiendo sus nombres a la lista de parámetros de selección de algoritmos. Una implementación de JCRE puede soportar los más nuevos algoritmos añadidos siempre que se suministren sus implementaciones. Debido a que las constantes solo se añaden o se borran de las API´s, el soportar algoritmos adicionales solo requiere una revisión de una versión menor. Las nuevas implementaciones de las API's serán compatibles con las implementaciones previas.

Gracias a la separación de la interfaz y de la implementación, las crypto API's son ligeras en cuanto al tamaño del código, mientras que soportan un conjunto extensible de servicios criptográficos. Dependiendo de la memoria disponible en la tarjeta, se podría elegir que en una implementación particular de JCRE se incluyeran todos, algunos o ninguno de los algoritmos posibles. Sin embargo, esto impone una condición para el desarrollador de applets, que tiene que saber los tipos de servicios criptográficos soportados en las tarjetas en las que los applets funcionarán. El JCRE

lanzará una `CryptoException` si se elige un servicio criptográfico o un algoritmo del servicio que no está disponible en la tarjeta.

5.1.3.3 Estructura de los paquetes

Las crypto API's de Java Card están estructuradas en dos paquetes: `javacard.security` y `javacardx.crypto`.

El paquete `javacard.security` se compone de varias interfaces para implementar claves simétricas o asimétricas, la clase constructora de claves, las clases de autenticación (asimilación de mensajes y firmas), y las clases para la generación de números aleatorios. Las interfaces y las clases del paquete `javacard.security` se muestran en la siguiente tabla:

Tabla 10: Paquete `javacard.security`

Clase o Interfaz	Descripción
`Key`	Interfaz base para todas las claves.
`SecretKey`	Interfaz base para las claves usadas en los algoritmos simétricos.
`DESKey`	Representa un clave de 8/16/24 bytes para DES o dos claves para DESede o tres claves para DESede.
`PrivateKey`	Interfaz base para las claves privadas usadas en algoritmos asimétricos.
`PublicKey`	Interfaz base para las claves públicas usadas en algoritmos asimétricos.
`RSAPrivateKey`	Se usa para firmar datos utilizando el algoritmo RSA en la forma módulo/exponente.
`RSAPrivateCrKey`	Usado para firmar datos utilizando el algoritmo RSA en la forma Chinese Remainder Theorem.
`RSAPublicKey`	Usado para verificar firmas en los datos firmados mediante uso del algoritmo RSA.
`DSAKey`	Interfaz base para las implementaciones de clave pública y privada del algoritmo DSA.
`DSAPrivateKey`	Usado para firmar en los datos utilizando el algoritmo DSA.
`DSAPublicKey`	Usado para verificar firmas utilizando el algoritmo DSA.
`KeyBuilder`	Clase generadora de objetos clave.
`MessageDigest`	Clase base abstracta para algoritmos de dispersión.
`Signature`	Clase base abstracta para algoritmos de firma.
`RandomData`	Clase base abstracta para generación de datos aleatorios.
`CryptoException`	Representa una excepción relacionada con la criptografía.

El paquete `javacardx.crypto` es una extensión. Consta de la clase `Cipher`, que permite el uso de una codificación robusta, y la interfaz `KeyEncryption` que permite la implementación de una clave para acceder a los datos codificados con la clave. Las clases del paquete `javacardx.crypto` están sujetas al control de exportación:

Tabla 11: Paquete `javacardx.crypto`

Clase o Interfaz	Descripción
`Cipher`	Provee la funcionalidad de un algoritmo de cifrado para la codificación y la decodificación. La clase `Cipher` es una clase base abstracta.
`KeyEncryption`	Permite la implementación de una clave para acceder a los datos codificados.

5.1.4 CÓDIGOS DE EJEMPLO

En esta sección se proporcionan códigos de ejemplo para mostrar como generar la asimilación de un mensaje, como construir una clave criptográfica, como calcular y verificar una firma, como codificar y decodificar datos en los applets y como usar la clase `RandomData`.

5.1.4.1 Cálculo de una asimilación de un mensaje

La asimilación de un mensaje es una dispersión única y fiable de un mensaje, que permite saber al receptor que el mensaje recibido es exactamente el mismo que el que ha sido enviado.

El primer paso, para calcular la asimilación, es crear un objeto de asimilación de mensajes. Para hacerlo, se puede llamar al método de fábrica `getInstance` de la clase `MessageDigest`.

```
public static MessageDigest getInstance(byte algorithm, boolean
externalAccess)
```

El parámetro `algorithm` especifica el algoritmo deseado, usando uno de los parámetros de selección de algoritmos. Las API's de Java Card 2.1 soportan tres posibles algoritmos: SHA1, MD5, y RIPE MD-160. Sus correspondientes parámetros de selección se llaman: `ALG_SHA`, `ALG_MD5`, y `ALG_RIPEMD160`.

El segundo parámetro, `externalAccess`, está especialmente diseñado para el entorno Java Card. Como se describe en la sección 4.7, cuando se crea el objeto de asimilación de mensajes, se puede acceder a dicho objeto desde el applet actualmente activo y desde cualquier applet del mismo paquete. Un applet definido en un paquete distinto podría acceder al objeto de asimilación de mensajes a través de un método de interfaz compartida. Si el parámetro `externalAccess` está puesto a `false` (significa que el objeto de asimilación de mensajes devuelto no está dirigido a un acceso externo), solo pueden acceder al objeto de asimilación de mensajes aquellos applets que pertenezcan a su mismo contexto y cuando uno de estos applets sea el applet actualmente seleccionado.

Si un applet necesita usar la instancia de una asimilación de un mensaje a lo largo de múltiples sesiones CAD, no debería llamar al método `getInstance` cada vez que necesite al objeto. En su lugar, debería solicitar una instancia de una asimilación de un mensaje en el constructor del applet y guardarla en una localización permanente (como un campo de instancia) para que se pueda usar en otra sesión. Esto previene la pérdida potencial de memoria cuando el recolector de basura no está disponible en la tarjeta. El siguiente ejemplo muestra como calcular la asimilación de un mensaje usando el algortimo SHA:

```
public class MyApp extends Applet {
      private MessageDigest sha;
      public MyApplet() {
            sha = MessageDigest.getInstance (MessageDigest.
            ALG_SHA, false);
            ...
      }
}
```

En el próximo ejemplo se suponen que se tienen tres arrays de bytes, `m1`, `m2`, y `m3`, que forman la entrada total del mensaje de dispersión que se quiere calcular. Se debería calcular la dispersión a través de las siguientes llamadas:

```
//todos los datos del array m1 se introducen en el
//mensaje de dispersión
sha.update(m1, (short)0, (short)(m1.length));
//se introducen 8 bytes más de datos que vienen del
//array m2, empezando con un offset igual a 0
sha.update(m2, (short)0, (short)8);
//se envían todos los datos del array m3 como el último
//bloque y guarda el valor de dispersión en un array
//de bytes, digest, empezando con un offset igual a 0
sha.doFinal(m3, (short)0, (short)(m3.length), digest, (short)0);
```

El método `update` permite ir introduciendo los datos de entrada para generar la dispersión. Pero cuando se quiere introducir el último bloque de datos, se debería llamar al método `doFinal`. Esto informa de que es el final de los datos de entrada y que el sistema debería llevar a cabo las operaciones finales, tales como el relleno. El método `doFinal` completa y devuelve el cálculo de la dispersión en un array de salida especificado. En el ejemplo, escribe el valor de dispersión en el comienzo del array de bytes `digest`. En el método `doFinal`, la salida puede superponerse sobre los datos de entrada, por ello se puede reutilizar el array de bytes `m3` como salida.

```
sha.doFinal(m3, (short)0, (short)(m3.length), m3,(short)0);
```

Después de realizar la llamada al método `doFinal`, el objeto de asimilación de mensajes se resetea automáticamente. De este modo se puede usar para calcular nuevos valores.

Si todos los datos de entrada caben en un array de bytes, se debería saltar el método `update` y llamar solamente al método `doFinal`. Esto es debido a que el método `update` usa almacenamiento temporal para resultados intermedios del cálculo de la dispersión. Se debería usar solamente si todos los datos de entrada requeridos para la dispersión no se pueden contener en un array de bytes.

Nota: Debido a que el método `uptate` quizás produzca el consumo de recursos adicionales o baje el rendimiento, use solamente el método `doFinal` siempre que sea posible.

En cualquier punto durante el cálculo de la dispersión y antes del método `doFinal`, se puede llamar al método `reset` para ignorar las entradas previas y resetear el estado actual, y así realizar un nuevo cálculo.

La clase `MessageDigest` también define dos métodos de interrogación que un applet puede invocar en la instancia de una asimilación de mensajes. El método `getAlgorithm` devuelve el algoritmo particular usado en la implementación de la instancia, y el método `getLength` devuelve el número de bytes del valor de dispersión de aquel algoritmo.

5.1.4.2 Construcción de una clave criptográfica

Una clave criptográfica es un valor secreto que se suministra a un algoritmo de cifrado para codificar y decodificar datos. Las API's de criptografía de Java Card definen un extenso conjunto de interfaces para implementar claves simétricas y asimétricas. Para construir una clave, se llamaría al método de fábrica `buildKey` de la clase `KeyBuilder`:

```
public static Key buildKey (byte keyType, short keyLength, boolean keyEncryption);
```

La clase `KeyBuilder` define un conjunto de parámetros de selección que se pueden elegir para seleccionar un tipo de clave y la longitud de la clave. Por ejemplo, para crear una clave privada RSA de 64 bytes de longitud ($64 \cdot 8 = 512$ bits), se llama al método `buildKey` de la siguiente manera:

```
Key rsa_private_key;
rsa_private_key = KeyBuilder.buildKey (KeyBuilder.TYPE_RSA_PRIVATE, KeyBuilder.LENGTH_RSA_512, false);
```

El método `buildKey` devuelve un objeto del tipo de la interfaz `Key`. El tipo real del objeto es una clase que implementa la interfaz de la clave, del tipo de la clave solicitada. En el ejemplo, la clase que implementa la clave debería implementar la interfaz `RSAPrivateKey`. Para poder invocar los métodos de la interfaz `RSAPrivateKey`, se le hace un cast del tipo `RSAPrivateKey` al objeto de clave:

```
RSAPrivateKey rsa_private_key;
rsa_private_key = (RSAPrivateKey) KeyBuilder.buildKey (KeyBuilder.TYPE_RSA_PRIVATE, KeyBuilder.LENGTH_RSA_512, false);
```

El método `buildKey` devuelve un objeto de clave con el tipo de clave solicitada, pero el objeto de clave no está inicializado.

Una clave privada RSA (una clave pública se inicializa de la misma forma) se inicializa y está lista para usarse, cuando se ajustan el módulo y el exponente de la clave. Para ajustarlos, se llaman a los métodos `setModulus` y `setExponent` de la interfaz `RSAPrivateKey`:

```
public void setExponent(byte[] buffer, short offset, short length)
public void setModulus(byte[] buffer, short offset, short length)
```

Por ejemplo, para inicializar una clave DES, se llama al método `setKey` de la interfaz `DESKey`:

```
public void setKey(byte[] keyData, short kOff)
```

Los datos de la clave se especifican en el array `keyData` empezando a partir del offset `kOff`. Después de la finalización del método `setKey`, un applet no necesita retener los datos en el array `keyData`, porque los datos se copian a un buffer interno.

Por razones de seguridad, una clave quizás pueda estar asociada a un objeto de cifrado, que puede decodificar internamente los datos de la clave. Quizás se habrá dado cuenta de que el método `buildKey` tiene un tercer parámetro, además de `keyType` y `keyLength`. Si este parámetro está puesto a `true`, se requiere la implementación de la

clave para implementar la interfaz `javacardx.crypto.keyEncryption`. La interfaz `keyEncryption` define los métodos `setKeyCipher` y `getKeyCipher`.

Si el objeto de clave está asociado con un objeto de cifrado a través del método `setKeyCipher`, los datos de entrada de la clave se deberían proveer en texto cifrado. De otra manera, si el objeto de clave no implementa la interfaz `KeyEncryption` o si el objeto de cifrado asociado está puesto a `null`, los datos de entrada de la clave se deberían proveer en texto plano.

En cualquier caso, cuando se quiera recuperar los datos de la clave se invoca a un método y los datos de clave se devuelven en texto plano. Por ejemplo, el método `getKey` de la interfaz `DESKey` devuelve el texto plano de la clave DES.

5.1.4.3 Firmar y verificar una firma

Una firma digital provee dos servicios de seguridad: autentificación e integridad. La clase `javacard.security.Signature` está diseñada para usarse de forma similar a la clase `MessageDigest`. Para crear un objeto `Signature`, se llama al método de fábrica `getInstance` de la clase `Signature`:

```
Signature signature;
signature = Signature.getInstance(Signature.ALG_DSA_SHA,false);
```

Para especificar un algoritmo, se puede usar alguno de los parámetros de selección de algoritmos de la clase `Signature`. Un parámetro de selección de algoritmo especifica la asimilación del mensaje y el algoritmo de codificación. La clase `Signature` soporta un extenso juego de posibles algoritmos de firma. En el ejemplo se firma y verifica una asimilación SHA, usando el algoritmo DSA. El segundo parámetro, `externalAccess`, indica si se puede acceder al objeto `Signature` devuelto, desde otro contexto que no sea el suyo.

Debido a que una firma usa una clave, primero se necesita inicializar el objeto `Signature`. Para hacerlo, se llama a uno de los dos métodos `init`:

```
public void init (Key thekey, byte theMode);
public void init (Key thekey, byte theMode, byte[] bArray, short
bOff, short bLen);
```

En un algoritmo asimétrico, para firmar y verificar no se usa la misma clave. Por lo tanto, se necesita especificar como se usa la clave en el segundo parámetro `theMode`. Hay dos modos, que están definidos en la clase `Signature`:

- `MODE_SIGN`: indica modo de firma.
- `MODE_VERIFY`: indica modo de verificación.

El segundo método `init` también permite especificar los datos de inicialización del algoritmo mediante el array de bytes `bArray`. Un ejemplo de datos de inicialización es el vector (IV) para DESede en el modo CBC.

Para calcular una firma, primero se introducen los datos usando el método `update`. El último bloque de datos se introduce con una llamada al método `sign`. Aquí

hay un código de ejemplo que calcula la firma de los datos contenidos en los arrays `s1`, `s2` y `s3`.

```
//se introducen los datos del array s1 y s2
signature.update(s1, (short)0, (short)(s1.length));
signature.update(s2, (short)0, (short)(s2.length));
//se introducen los datos del array s3, que son
//los últimos y genera la firma en el array sig_buffer
signature.sign(s3,  (short)0,  (short)(s3.length),  sig_buffer,
(short)0);
```

Para verificar una firma, primero se introducen los mismos datos de entrada usando el método `update`. Con el último bloque de datos se llama al método `verify`. El método `verify` verifica la firma calculada a partir de los datos de entrada comparándola con la que se le suministra. Si ambas coinciden, devuelve `true`. El siguiente ejemplo muestra como verificar la firma calculada en el ejemplo anterior:

```
signature.update(s1, (short)0, (short)(s1.length));
signature.update(s2, (short)0, (short)(s2.length));
//introduce el último bloque de datos del array s3 y
//verifica la firma calculada comparándola con la firma
//dada en el buffer sig_buffer
if    (signature.verify(s3,    (short)0,    (short)(s3.length),
sig_buffer, sig_offset, sig_length) != true) {
     ISOException.throwIt(SW_WRONG_SIGNATURE);
}
```

Al igual que los métodos de la clase `MessageDigest`, el método `update` sólo se debería usar si todos los datos de entrada no caben en un array de bytes. Después de llamar a los métodos `sign` o `verify`, el objeto `Signature` se resetea al estado previo a su inicialización mediante una llamada al método `init`. En el método `sign` también se puede usar el mismo array para los datos de entrada y de salida, y quizás los datos se superpongan. Los métodos de interrogación `getAlgorithm` y `getLength` están definidos también en la clase `Signature`.

5.1.4.4 Codificación y decodificación de datos

La codificación es una herramienta que sirve para proteger la privacidad de los datos. La codificación codifica los datos que se encuentran en texto plano y los transforma texto cifrado. La decodificación recupera el texto plano original a partir del texto cifrado.

La clase `javacardx.crypto.Cipher` provee servicios de codificación y decodificación con algoritmos simétricos o asimétricos. Al igual que en las clases `MessageDigest` y `Signature`, para crear un objeto `Cipher` se puede llamar al método de fábrica `getInstance` y suministrarle dos parámetros. El primer parámetro especifica un algoritmo de cifrado. El segundo parámetro, `externalAccess`, indica si se puede acceder externamente al objeto `Cipher`, desde un contexto distinto al suyo (ver la clase `keyBuilder`).

Entonces se inicializa el objeto `Cipher` con una clave apropiada y se especifica si la clave se usa para codificar o para decodificar. Para hacerlo, se llama a uno de los dos métodos `init`:

```
public void init (Key theKey, byte theMode);
public void init (Key theKey, byte theMode, byte[] bArray, short
bOff, short bLen);
```

El siguiente ejemplo crea un objeto `Cipher` con el algoritmo DES en el modo CBC. A los datos de entrada no se les incluirá relleno. El objeto `Cipher` se inicializa con una clave DES para codificar.

```
Cipher cipher;
cipher = Cipher.getInstance(Cipher.ALG_DES_CBC_NO_PAD, false);
cipher.init(des_key, Cipher.MODE_ENCRYPT);
```

También se puede inicializar el objeto `Cipher` para decodificar. Para ello se elige el parámetro de selección `MODE_DECRYPT`, que especifica que la clave suministrada en el método `init` es para decodificar.

El siguiente paso para codificar los datos, es usar los métodos `update` y `doFinal`:

```
public short update(byte[] inBuf, short inOffset, short
inLength, byte[] outBuff, short outOffset);
public short doFinal(byte[] inBuf, short inOffset, short
inLength, byte[] outBuff, short outOffset);
```

Ambos métodos toman el texto plano del buffer de entrada y escriben el texto cifrado calculado en el buffer de salida. Se debería llamar al método `update` para introducir los dados de entrada y llamar al método `doFinal` para introducir el último bloque de datos. La versión final del texto cifrado se calcula en el buffer de salida del método `doFinal`. Después de realizar la llamada al método `doFinal`, el objeto `Cipher` se resetea al estado en que estaba cuando fue previamente inicializado a través de una llamada al método `init`. Para decodificar datos, también se llaman a los métodos `update` y `doFinal`. Pero en este caso, el texto cifrado se introduce en el buffer de entrada y el texto plano se escribe en el buffer de salida. Además, debido a que el método `update` supone una sobrecarga por guardar los resultados intermedios, solamente se debería llamar si todos los datos de entrada no caben en un array de bytes.

5.1.4.5 Generación de datos aleatorios

Los números aleatorios se necesitan habitualmente para procedimientos criptográficos. Para crear un generador de números aleatorios, se llama al método `getInstance` de la clase `javacard.security.RandomData`, especificándole un algoritmo. El parámetro de selección de algoritmo puede ser `RandomData.ALG_PSEUDO_RANDOM` para utilizar el algoritmo de generación de números pseudoaleatorios o `RandomData.ALG_SECURE_RANDOM` para utilizar el algoritmo de generación de números aleatorios criptográficamente seguros.

Como otras clases basadas en algoritmos de las API's de criptografía de Java Card, la clase `javacard.security.RandomData` es una clase base abstracta. Así que una clase que la implemente debe extender la clase base abstracta `RandomData`. El objeto `RandomData` devuelto por el método `getInstance` es un objeto de la clase de implementación, que implementa el algoritmo deseado.

La semilla del objeto `RandomData` para números pseudoaleatorios se inicializa a un valor interno por defecto, mientras que el objeto `RandomData` para números aleatorios seguros consigue inicializar la semilla con un valor completamente aleatorio. Se podría introducir la semilla del generador de números aleatorios a través de una llamada al método `setSeed`.

Finalmente, para conseguir un número aleatorio, se llama al método `generateData` como se muestra a continuación:

```
RandomData random_data = RandomData.getInstance
(RandomData.ALG_SECURE_RANDOM);
//la semilla se suministra en el array de bytes seed
random_data.setSeed(seed, seed_offset, seed_length);
//se escribe un número aleatorio en el array de bytes random_num
random_data.generateData(random_num, random_num_offset,
 random_num_length);
```

5.2 SEGURIDAD EN LA PLATAFORMA JAVA CARD

El procesamiento seguro es la razón fundamental para usar las smart cards. Por ello, las consideraciones respecto a seguridad son las más importantes para los desarrolladores de smart cards. En esta sección se verán las características de seguridad en la plataforma Java Card, los mecanismos que hacen cumplir las características de seguridad y diversas consideraciones de seguridad a la hora de diseñar e implementar los applets.

La seguridad se debe considerar desde el punto de vista del sistema en su totalidad. La plataforma Java Card está construida en lo alto de la plataforma de la smart card. Esta plataforma consiste en el hardware de la smart card y el sistema operativo nativo, y el sistema del host, con el que la tarjeta se comunica.

5.2.1 CARACTERÍSTICAS DE LA SEGURIDAD EN LA PLATAFORMA JAVA CARD

Las características de la seguridad en la plataforma Java Card son una combinación de los fundamentos de la seguridad del lenguaje Java y de las protecciones adicionales de seguridad definidas en la plataforma Java Card.

5.2.1.1 La seguridad en el lenguaje Java

La plataforma Java Card soporta un subconjunto de especificaciones del lenguaje de programación Java y de la máquina virtual de Java, apropiadas para las aplicaciones de las smart cards. Por lo tanto, la plataforma Java Card hereda las características de seguridad construidas en el subconjunto soportado del lenguaje Java. La seguridad del lenguaje Java forma la base de la seguridad de la plataforma Java Card:

- El lenguaje Java está fuertemente tipado. No se pueden hacer conversiones ilegales de datos, tales como convertir enteros en punteros.
- El lenguaje Java impone restricciones en los accesos a los arrays.

- El lenguaje Java no tiene aritmética de punteros. Así, no hay manera de hacer punteros que permitan a los programas maliciosos curiosear en la memoria.
- Las variables se deben inicializar antes de que sean usadas.
- El nivel de acceso de todas las clases, métodos y campos, está estrictamente controlado. Por ejemplo, no se puede invocar a un método privado desde fuera de su clase de definición.

5.2.1.2 Características adicionales de la seguridad en la plataforma Java Card

Los distribuidores de tarjetas desean una plataforma de computación segura que cumpla los requisitos especiales del sistema de la smart card. A continuación se muestran las características adicionales de seguridad, definidas en la plataforma Java Card:

- Modelos de objetos transitorios y persistentes: En la plataforma Java Card, los objetos por defecto se guardan en memoria persistente. Por seguridad y razones de rendimiento, la plataforma Java Card permite que los datos temporales (tales como las claves de sesión) se guarden en objetos transitorios en la memoria RAM. El tiempo de vida de tales objetos se puede declarar como `CLEAR_ON_RESET` o `CLEAR_ON_DESELECT`. Los contenidos de un objeto transitorio se ajustan al valor por defecto del objeto (`cero`, `false` o `null`) cuando la tarjeta se resetea o cuando el applet actualmente seleccionado se deselecciona.
- Atomicidad y transacciones: En la plataforma Java Card, los datos se guardan en objetos que se encuentran en memoria persistente. Podría ocurrir un fallo o una pérdida de energía durante las operaciones de escritura en la tarjeta. Para asegurar la integridad de la tarjeta, se definen tres características de seguridad en la plataforma Java Card. La primera es que la plataforma Java Card garantiza que una simple actualización del campo de un objeto persistente o de una clase, será atómico. Cuando ocurre un error durante una actualización, la plataforma asegura que el contenido del campo será restaurado a su valor previo. La segunda es que el método `arrayCopy` de la clase `javacard.framework.Util`, garantiza la atomicidad para actualizaciones de bloques de múltiples elementos de un array. Aquí, la atomicidad significa que o se copian todos los bytes correctamente o se restauran todos los valores de los bytes del array de destino. La tercera es que la plataforma Java Card soporta el modelo de transacciones en el que un applet puede actualizar atómicamente varios campos diferentes de diferentes objetos persistentes. O todas las actualizaciones de la transacción se llevan a cabo correctamente o se restauran todos los campos persistentes a sus valores previos.
- El applet firewall: La seguridad y la integridad del sistema (el JCRE) y de cada applet residente en la smart card de Java, se protegen gracias al applet firewall. El applet firewall asegura el aislamiento del applet y separa el espacio del sistema del espacio del applet. En el esquema del firewall, cada applet se ejecuta en un contexto (espacio del objeto). Los applets no pueden acceder a los objetos de otros applets a menos que estén definidos en el

mismo paquete (y por ello comparten el mismo contexto) o mediante mecanismos de compartición de objetos seguros y bien definidos, que la plataforma soporta.

- Compartición de objetos: La compartición de objetos en el sistema Java Card se consigue de las siguientes maneras. La primera es que el JCRE es un usuario privilegiado que tiene acceso completo a los applet y a los objetos creados por los applets. La segunda es que un applet gana el acceso a los servicios y recursos del JCRE a través de los objetos de punto de entrada al JCRE. La tercera es que los applets de diferentes contextos pueden compartir objetos que son instancias de clases que implementan una interfaz compartida. Tales objetos se llaman objetos de interfaz compartida. Finalmente, los applets y el JCRE pueden compartir datos a través de arrays globales.
- Métodos nativos en applets: Los métodos nativos no se ejecutan en la máquina virtual de Java y por eso no están sujetos a las protecciones de seguridad de la plataforma Java Card. Por lo tanto, está prohibido que los postissuance applets (los applets que se cargan en la tarjeta después de que la tarjeta haya sido distribuida) contengan métodos nativos. A los applets que residen en ROM y a los preissuance applets, se les permite tener métodos nativos, ya que los distribuidores de tarjetas son los que controlan estos applets. La interfaz para acceder al código nativo desde tales applets es una tecnología propietaria de los vendedores de tarjetas.

5.2.2 MECANISMOS DE SEGURIDAD DE LA PLATAFORMA JAVA CARD

La seguridad es como una cadena, un solo eslabón débil puede hacer que se rompa la cadena. Las características de seguridad de Java Card están aseguradas a través de un número de mecanismos que se aplican en cada nivel del proceso de desarrollo del applet y en el procedimiento de instalación de la aplicación en tiempo de ejecución.

5.2.2.1 Comprobación en tiempo de compilación

Un applet se compone de uno o más ficheros Java. El código Java se compila usando cualquier entorno de desarrollo estándar de Java, tales como el JDK de Sun o el Café de Symantec. Los archivos binarios producidos por el compilador a partir del código fuente se llaman ficheros class.

El compilador de Java lleva a cabo comprobaciones frecuentes y rigurosas en tiempo de compilación, de modo que el compilador detecta tantos errores como le sea posible. El lenguaje Java está fuertemente tipado. A diferencia de C ó C++, el sistema de tipos no tiene fugas. Por ejemplo:

- A los objetos no se les puede hacer un cast del tipo de una subclase sin una comprobación explícita en tiempo de ejecución.
- Se comprueban todas las referencias a métodos y variables para asegurarse de que los objetos son del tipo apropiado.
- El compilador comprueba que no se violan los controles de acceso (tales como las referencias a variables privadas o métodos privados de otra clase).

- Los enteros no se pueden convertir en objetos. Los objetos no se pueden convertir en enteros.
- El compilador asegura estrictamente que un programa no accede al valor de una variable local no inicializada.

5.2.2.2 Verificación de los archivos class y comprobación del subconjunto del Lenguaje Java usado en Java Card

Aunque un compilador de confianza puede asegurar que el código fuente de Java no viola las reglas de seguridad, los ficheros class pueden venir de una red que no sea de confianza. En el entorno Java, un verificador (un componente de la máquina virtual de Java) comprueba todos los archivos class cargados, antes de que se ejecuten. El verificador de archivos class examina los archivos class en varias pasadas para asegurar que tienen el formato correcto y que sus bytecodes cumplen un conjunto de restricciones estructurales. En particular, los archivos class se comprueban para asegurar lo siguiente:

- No hay violaciones de gestión de memoria y tampoco underflows ni overflows de la pila.
- Se fuerzan las restricciones de acceso: por ejemplo, no se puede acceder a los métodos privados ni a los campos privados desde fuera de sus clases de definición.
- Se llaman a los métodos con los argumentos apropiados (en cuanto al número de argumentos del método y al tipo de cada uno).
- Se modifican los campos con valores del tipo apropiado.
- No se pueden hacer punteros.
- No se pueden hacer conversiones ilegales de datos, tales como convertir enteros a punteros.
- Se imponen las reglas de compatibilidad binaria.

A diferencia de la máquina virtual de Java, la máquina virtual de Java tiene una arquitectura dividida en dos partes: el convertidor que se ejecuta en un PC, fuera de la tarjeta y el intérprete que se ejecuta en el interior de la tarjeta. El convertidor es la parte inicial de la máquina virtual y toma los archivos class como entrada. Durante la conversión, los archivos class de un applet están sujetos al mismo nivel de verificación rigurosa. Es como si estuvieran en la máquina virtual de Java, con su verificador de archivos class en el momento de la carga de las clases. El convertidor puede incorporar el componente de verificación de la máquina virtual de Java, o puede implementar su propio verificador de archivos class.

Debido a que la tecnología Java Card define un subconjunto del lenguaje Java, el convertidor debe comprobar los archivos class mucho más, para asegurar que solamente se usan las características de ese subconjunto. Este paso se llama comprobación del subconjunto (subset checking). Durante este paso, el convertidor comprueba que el applet no viola las siguientes reglas del subconjunto:

- No se usan los tipos no soportados (por ejemplo, las variables del tipo `char`, `long`, `double` y `float`). Solamente se usa el tipo de dato `int` si el intérprete de Java Card lo soporta.
- No se usan las características no soportadas del lenguaje Java. Por ejemplo, un applet no debería crear hilos o arrays multidimensionales.
- La utilización de ciertas operaciones de Java están restringidas a cierto alcance. El alcance de cada una de esas operaciones en la plataforma Java Card, es menor que el de la plataforma Java. Por ejemplo, un paquete puede tener no más de 255 clases públicas e interfaces, y un array puede contener un máximo de 32.767 campos.
- No puede ocurrir un overflow o un underflow que quizás cause resultados aritméticos que se calculen de forma diferente a como se calcularían en la plataforma Java.

5.2.2.3 El fichero CAP y la verificación del fichero de exportación

Las clases de un applet se constituyen de uno o más paquetes. El convertidor toma todas las clases de un paquete y las convierte en un fichero CAP. Entonces el fichero CAP se carga en la smart card de Java y el intérprete lo ejecuta. Además de crear un fichero CAP, el convertidor genera un fichero de exportación que representa las API's públicas del paquete que se ha convertido. El intérprete no usa directamente el fichero de exportación. Se usa más tarde para convertir otro paquete que importe clases del paquete representado en el fichero de exportación. La información contenida en el fichero de exportación se usa para linkar y para comprobaciones de referencias externas.

El fichero CAP de la plataforma Java Card tiene la misma importancia que el archivo class para la plataforma Java. Es un formato binario para la carga de paquetes Java en una smart card de Java.

En la práctica, no hay garantías de que un fichero CAP, generado por un convertidor de confianza a partir de un verificador de archivos class, se cargue inmediatamente en una smart card de Java en un entorno seguro. De este modo, la corrección e integridad de un fichero CAP no es algo indudable. El trabajo del verificador de archivos CAP es asegurar que el fichero CAP cumple con las reglas. Debido a la limitación de memoria y potencia de cálculo de una smart card, el verificador de ficheros CAP se ejecuta fuera de la tarjeta. El verificador lleva a cabo comprobaciones estáticas en un fichero CAP antes de que se cargue en una smart card de Java.

Debido a la analogía de los papeles desempeñados por los archivos class y por los archivos CAP, el verificador de ficheros CAP tiene una función similar a la del verificador de ficheros class. Es decir, asegura que el fichero CAP tiene el formato correcto y que los bytecodes se atienen al conjunto de restricciones estructurales. En particular, comprueba el fichero CAP para asegurar lo siguiente:

- No hay violaciones de gestión de memoria y tampoco underflows ni overflows de la pila.

- Se fuerzan las restricciones de acceso: por ejemplo, no se puede acceder a los métodos privados ni a los campos privados desde fuera del objeto.
- Se llaman a los métodos con los argumentos apropiados (en cuanto a su número y al tipo de cada uno).
- Se modifican los campos con valores del tipo apropiado.
- No se pueden hacer punteros.
- No se pueden hacer conversiones ilegales de datos, tales como convertir enteros a punteros.
- Se imponen las reglas de compatibilidad binaria.

Advertir que estas comprobaciones son idénticas a las que realiza el verificador de archivos class. Además, el verificador de ficheros CAP impone reglas que son especiales debido a la estructura del fichero CAP y al entorno de Java Card:

- El paquete y cada applet definido en el paquete deben tener un AID válido con una longitud de 5 a 16 bytes. El AID del paquete y los AID's de los applets deben compartir el mismo número RID (los 5 primeros bytes del AID).
- Un applet debe definir un método `install` con la firma correcta, para que en la tarjeta se puedan crear las instancias del applet de forma apropiada.
- El orden de las definiciones de interfaces y clases en un fichero CAP deben seguir las reglas por las que las interfaces aparecen delante de las clases, y las superclases aparecen delante de las subclases. Esto asegura que la carga del archivo CAP y el proceso de linkado se puedan llevar de manera secuencial a la tarjeta.
- Se ajusta la bandera `int` si se usa el tipo `int` en el fichero CAP. Con esta comprobación, a una implementación de Java Card que no soporte el tipo `int` se le permite no aceptar el fichero CAP durante la carga, mediante una simple comprobación de la bandera `int`.

Durante la verificación, el verificador de ficheros CAP asegura que un archivo CAP es consistente internamente. Es consistente con los ficheros de exportación que él importa y además es consistente con el fichero de exportación que representa sus API's:

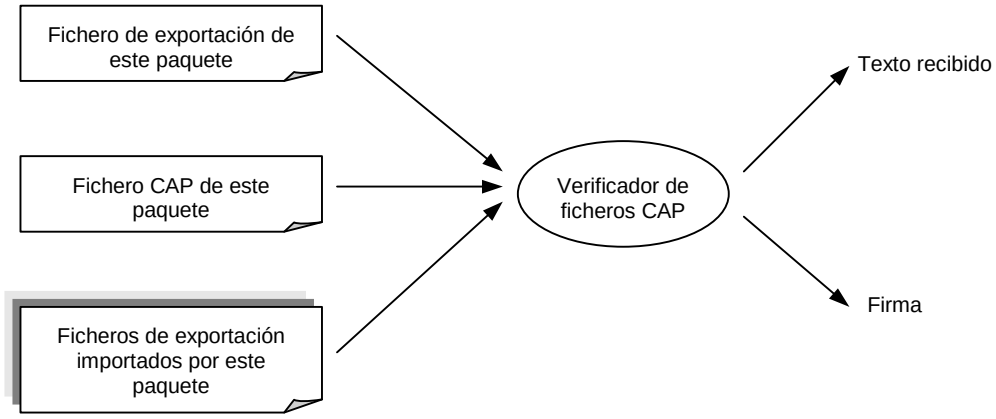


Figura 35: Proceso de verificación del fichero CAP

El verificador también examina si se han seguido las reglas de la versión de Java Card, incluyendo aquellas impuestas por la compatibilidad binaria (que se definen en las especificaciones de la máquina virtual de Java Card).

Si los archivos de exportación usados durante la verificación no son de confianza, también se deberían verificar. La verificación de ficheros de exportación comprueba un fichero de exportación internamente y lo contrasta con su correspondiente fichero CAP para asegurar que está bien formado y que satisface las restricciones requeridas por las especificaciones de la máquina virtual de Java Card.

5.2.2.4 Comprobaciones en la instalación

La instalación del fichero CAP se consigue a través de la cooperación del programa de instalación que se encuentra fuera de la tarjeta y el instalador de la tarjeta. Juntos, cargan un archivo CAP y si el fichero CAP define varios applets, crean una o más instancias de estos applets (dependiendo de la implementación del instalador, la creación de los applets se puede llevar a cabo en una etapa posterior).

La seguridad en la instalación consta de dos niveles. En el primer nivel están las protecciones de seguridad estándares impuestas por el instalador y el JCRE. En el segundo nivel están los criterios de seguridad dictados por los proveedores. Ambos niveles juntos, protegen de lo siguiente:

- Corrupción de los datos de instalación.
- Falsificación de los datos de instalación.
- Incompatibilidad entre el archivo CAP y los recursos de la tarjeta.
- Accesos ilegales desde fuera del fichero CAP.
- Insuficiencia de recursos y otros errores producidos durante la instalación y la inicialización.
- Inconsistencia en el estado debido a una rotura de la tarjeta o pérdida de energía, durante el proceso de instalación.

La corrección e integridad de un fichero CAP se verifican fuera de la tarjeta. El instalador de Java Card no lleva a cabo la mayoría de las verificaciones tradicionales de Java en el momento en que carga una clase. Antes de que se escriba cualquier dato en la tarjeta, el instalador realiza comprobaciones para ver si la tarjeta puede soportar el fichero CAP. Por ejemplo, el instalador comprueba si los recursos de memoria disponible en la tarjeta son suficientes para el fichero CAP. Y si la tarjeta no soporta el tipo `int`, el instalador comprueba si el fichero CAP contiene algún uso del tipo `int`, es decir, comprueba si la bandera `int` del fichero CAP está cambiada.

Cuando está leyendo el fichero CAP, se puede linkar al vuelo o después de que se cargue el fichero CAP entero. El proceso de linkado incluye la resolución de las referencias internas y externas. El instalador asegura que las referencias internas son en realidad, locales a la memoria del paquete, y que las referencias externas están linkadas con localizaciones accesibles para otros paquetes y el JCRE. Además, el instalador asegura que el fichero CAP es compatible de forma binaria con el software existente en la tarjeta.

A diferencia de la plataforma Java, la unidad de carga en la plataforma Java Card es un paquete (un fichero CAP). El instalador asegura que el fichero CAP hace referencia solamente a paquetes que ya están preparados en la tarjeta, porque no se soporta la carga de nuevas clases de forma incremental.

Si el fichero CAP define un número cualquiera de applets, el instalador puede crear instancias de applets llamando a sus métodos `install`. Cuando se crea la instancia de un applet, se le asigna un contexto a la instancia del applet. Múltiples instancias del mismo applet y múltiples instancias de múltiples applets que están definidos en el mismo paquete, comparten un solo contexto, referido como contexto de grupo.

El proceso de instalación es transaccional. Si ocurre un error, una rotura de la tarjeta o una pérdida de energía, durante la instalación, el instalador rechaza el fichero CAP y cualquier applet que haya sido creado durante la instalación, y recupera el espacio y el estado previo del JCRE.

Más allá de los requisitos mínimos de seguridad definidos en el instalador y en el JCRE, la tecnología Java Card no estandariza la política de instalación. Los proveedores tienen flexibilidad a la hora de configurar la tarjeta para permitir la instalación de applets desde varias fuentes, posiblemente acoplando diferentes niveles de protección para applets de diferentes fuentes.

La forma más simple de protección, es autentificar el programa de instalación (que se encuentra fuera de la tarjeta) usando un PIN (de este modo se provee una medida de confianza en el proveedor del fichero CAP y en el contenido del fichero CAP). Una estrategia más sofisticada puede usar firma digital y codificación de datos.

5.2.2.5 Cadena de confianza forzada criptográficamente

El desarrollo de applets de Java Card y el proceso de instalación, consisten en el diseño del código fuente y la implementación; la compilación del código, la conversión y la verificación; y la instalación del archivo CAP en la tarjeta.

Durante este proceso, están involucradas varias partes, incluyendo los desarrolladores, suministradores, vendedores de terminales, vendedores de tarjetas y otros. Si el proceso entero no se lleva a cabo en un entorno físicamente seguro, quizás se necesite transportar los ficheros en una red abierta. De este modo, la seguridad del sistema Java Card se puede reforzar a través de la cadena de confianza, en la cual, la identidad de cada parte involucrada se autentifica y se protegen la confidencialidad y la integridad de los datos.

La criptografía moderna ofrece herramientas poderosas para autenticar la identidad y para asegurar la confidencialidad y la integridad. Cualquier archivo de Java Card (código fuente, archivo class, fichero CAP, o archivo de exportación) se puede codificar para mantenerlo en secreto durante el paso entre el desarrollo y la instalación en la tarjeta. El fichero se puede firmar digitalmente para asegurar la integridad y para comprobar la identidad de su proveedor. Antes de que se instale el fichero CAP, el host (incluyendo el CAD) y la tarjeta deberían autenticarse mútuamente. Una vez que el host está autenticado, para la tarjeta debería ser segura la carga de datos en texto plano. De otra manera, las tareas de decodificación y verificación de datos que se requerirían en la tarjeta, podrían demandar demasiados recursos.

Para preparar la confianza en tales casos, el proveedor de la tarjeta necesita definir políticas en cuanto a la gestión de claves (generación de claves e intercambio) y la utilización de los mecanismos criptográficos.

5.2.2.6 Aplicación de la seguridad en tiempo de ejecución

La aplicación de la seguridad de Java Card en tiempo de ejecución cubre dos áreas: asegura de forma segura el tipo de lenguaje Java e impone el aislamiento de los applets gracias al applet firewall.

El fichero CAP contiene suficiente información de tipo como para permitir la comprobación meticulosa de tipos en tiempo de ejecución. Sin embargo, debido a la limitación de los recursos de computación, la mayoría de las implementaciones de JCRE optan por confiar en las medidas de seguridad impuestas por el convertidor y el verificador, y confiar en las comprobaciones de la instalación para detectar ciertas clases de referencias ilegales. Muchas comprobaciones que se realizan estáticamente fuera de la tarjeta, no se repiten en tiempo de ejecución. El intérprete de Java Card lleva a cabo comprobaciones que se deben tratar dinámicamente, como la comparación en tiempo de ejecución del tipo de un objeto con su tipo estático.

Para imponer el applet firewall, cuando se accede a un objeto, el intérprete de Java Card realiza comprobaciones para determinar si el acceso se puede conceder. Esto permite a los applets del mismo paquete acceder a los objetos de cada uno de los otros. En situaciones donde hay necesidad de soportar aplicaciones de applets que cooperan entre sí a través de los límites que imponen los paquetes, el intérprete de Java Card permite la compartición de objetos a través de los objetos de punto de entrada al JCRE, los arrays globales y los objetos de interfaz compartida.

Además, las especificaciones de Java Card en tiempo de ejecución, dictan una serie de requisitos que debe cumplir la implementación del JCRE para asegurar la seguridad en tiempo de ejecución. A continuación se presentan los requisitos más notables:

- El JCRE debe asignar apropiadamente e implementar los objetos de punto de entrada al JCRE y los arrays globales. Por ejemplo, un applet no puede guardar referencias a objetos temporales de punto de acceso al JCRE en variables de la clase, instancias de variables, o campos de un array (incluyendo los arrays transitorios).
- Para prevenir que los datos potencialmente delicados de un applet puedan filtrarse a otro applet a través del buffer APDU global, el buffer APDU se debe poner a cero siempre que se seleccione un applet, antes de que el JCRE acepte una nueva APDU de comando.
- La implementación de los mecanismos de atomicidad y de transacción del JCRE debe estar de acuerdo con la especificación, para asegurar la integridad de los datos. Por ejemplo, cuando un applet finaliza con una transacción todavía en progreso, el JCRE debería interrumpir la transacción automáticamente, provocando de este modo que el sistema de transacciones realice cualquier operación de limpieza.
- Los fallos en tiempo de ejecución se deben tratar apropiadamente. Por ejemplo, un error recuperable por falta de recursos debería desembocar en una `SystemException`. Cualquier error no recuperable, como el overflow de la pila, debería causar la parada de la máquina virtual. Cuando ocurre un error no recuperable, una implementación de JCRE puede requerir opcionalmente que se bloquee la tarjeta. De este modo se previene su utilización en el futuro.

5.2.2.7 Soporte criptográfico de Java Card

Todos los applets de Java Card y el software del sistema usan los mecanismos criptográficos. Las API´s de criptografía de Java Card permiten un uso flexible de estos mecanismos, basados en qué política de seguridad se desea y en qué mecanismos están disponibles en la tarjeta. Los proveedores de tarjetas pueden configurar la tarjeta para que sea posible la elección de algoritmos, del tamaño de la clave, etc. Igualmente, los desarrolladores de applets son capaces de definir sus propias políticas criptográficas, limitadas por el proveedor de la tarjeta.

5.2.3 LA SEGURIDAD DE LOS APPLETS

La seguridad de cualquier aplicación está determinada por la seguridad de la plataforma en la que se ejecuta y por las características de seguridad diseñadas en la aplicación. La plataforma Java Card está diseñada para asegurar que los applets se puedan construir, instalar y ejecutar de una manera muy segura. También se permite a los proveedores el poder definir requisitos y políticas de seguridad en un futuro, para cumplir las necesidades de su industria particular. Por ejemplo, la Open Platform impone reglas de seguridad para la instalación de los applets y la gestión de applets en una smart card de Java.

La seguridad en el nivel de aplicación se necesita programar en los applets. Debido a que las características de seguridad están construidas en la plataforma misma, los desarrolladores de applets pueden concentrar sus esfuerzos en definir una estrategia

de seguridad para los applets, en lugar de poner un esfuerzo extra en programar los applets para compensar las deficiencias de una plataforma insegura.

Los desarrolladores de applets deberían definir una estrategia de seguridad para prevenir las formas más probables de ataque contra el applet. Cuando está bien definida, una estrategia de seguridad provee los siguientes objetivos de seguridad en los applets:

- Autenticación: El acceso a las funciones y datos del applet está controlado. La identidad de la aplicación del host que da comandos al applet, se debería autenticar. También, durante la compartición de objetos, el applet servidor debería verificar la identidad del cliente antes de concederle el servicio.
- Confidencialidad: La privacidad de los datos del applet se debe proteger. No se debería acceder a los datos seguros, como los números de cuenta y los balances, sin la autenticación apropiada. Los datos secretos, como los PIN's y las claves criptográficas privadas, nunca deberían salir de la tarjeta.
- Integridad: La corrección de los datos del applet está defendida. Los datos del applet no se pueden cambiar sin la autenticación apropiada. La modificación de los datos se debería proteger mediante la comprobación de errores. Por ejemplo, el balance de la cartera no puede exceder un límite máximo ni caer por debajo de cero.

Es necesario decir, que applets diferentes necesitan diferentes niveles de seguridad. Por ejemplo, el applet wallet requiere un mayor grado de seguridad que un applet de juego. También, la seguridad está impuesta por el coste en recursos de computación (rendimiento y memoria). Por lo tanto, los desarrolladores de applets debería evaluar los requisitos de seguridad del applet y elegir el mecanismo apropiado para proteger sus valiosos servicios.

6 APLICACIONES DE LA TECNOLOGÍA JAVA CARD A LOS DISPOSITIVOS MÓVILES (SIM/USIM)

En la actualidad, se hace un uso intensivo de las tarjetas inteligentes en el sector de las telecomunicaciones móviles. En este capítulo, dedicado a estas tarjetas inteligentes, se verán:

- Las características, en cuanto a hardware y software, de las tarjetas (SIM/USIM) usadas en las redes de telefonía móvil GSM y UMTS
- El SIM Application Toolkit, que permite a un applet (applet toolkit), que se encuentra en la tarjeta inteligente, utilizar el terminal móvil como interfaz para que el usuario pueda interaccionar con dicho applet. El SIM Application es un mecanismo por el que las operadoras de telefonía móvil pueden ofrecer nuevos servicios de valor añadido a sus clientes. Este mecanismo es el utilizado en la aplicación realizada de este proyecto fin de carrera.
- La SIM API para Java Card, que establece clases para que los applets puedan: acceder al sistema de ficheros de la tarjeta SIM y hacer uso del SIM Application Toolkit.

6.1 LA TARJETA SIM/USIM

6.1.1 INTRODUCCIÓN

En los estándares de telefonía móvil GSM y UMTS, el enlace radio facilita la intrusión, de modo que usuarios no autorizados pueden hacer un uso fraudulento del mismo. Para evitarlo se adoptan varias medidas de seguridad, entre las que destacan el encriptado (codificación) digital del enlace radio (para asegurar la privacidad de las conversaciones) y la autentificación (que es una comprobación de validación y de uso no autorizado de un terminal). La consecución de estas dos medidas se basa en el empleo de una tarjeta inteligente de identificación de usuario SIM (Suscriber Identity Module en GSM) ó USIM (en el mundo UMTS).

Al comienzo del estándar GSM, la SIM, al igual que cualquier smart card, estaba dotada de un microprocesador y de 8 kbytes de memoria, que permitían el almacenamiento de hasta 12 mensajes. Como se puede ver en la Figura 36 a medida que ha ido pasando el tiempo, se ha incrementado la potencia del hardware (memorias RAM, ROM, Flash y EEPROM más rápidas, incremento de la capacidad de proceso con sistemas RISC de hasta 32 bits) como en su capacidad para soportar servicios y funcionalidades avanzadas, incorporando tecnologías antes reservadas a sistemas de mayor capacidad (sistemas operativos multitarea como MultOS, lenguajes de alto nivel como Java o MEL, interoperabilidad, acceso remoto, etc.).

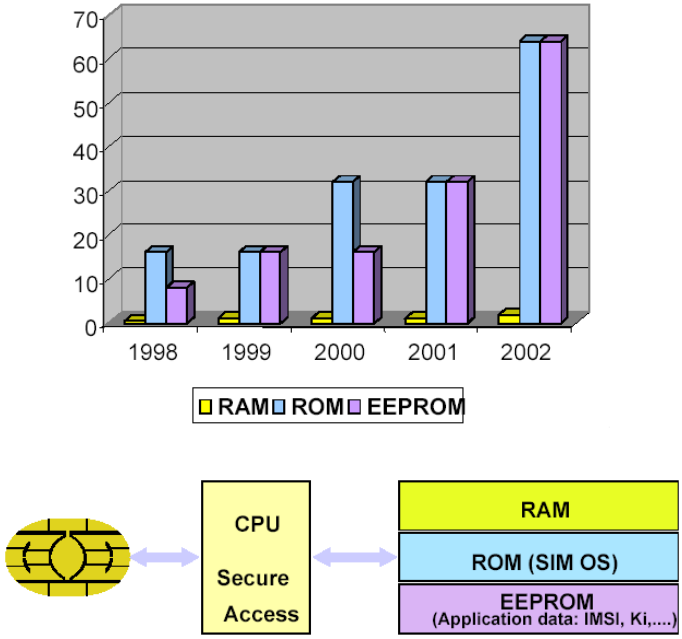


Figura 36: Evolución de la cantidad de memoria de las tarjetas SIM

Por ello, en las denominadas tarjetas fase 2+ aparece la noción de aplicación que se almacena y ejecuta dentro de la tarjeta y que utiliza el terminal como interfaz al mundo exterior, a través del estándar SIM Application Toolkit (del que hablará más adelante). UMTS reutiliza y potencia las ideas clave de la tarjeta SIM (soporte a la seguridad, portabilidad y plataforma de desarrollo de servicios) en lo que es una apuesta clara por esta tecnología.

6.1.2 ICC/UICC: (UNIVERSAL) INTEGRATED CIRCUIT CARD

Se define ICC/UICC como el soporte físico y lógico sobre el que se apoya un conjunto abierto de aplicaciones. La más importante es la aplicación GSM ó USIM (según la tarjeta sea ICC ó UICC), que ofrece un conjunto de servicios básicos tanto al terminal como a la red GSM ó UMTS. En las tarjetas UICC también se puede definir una aplicación GSM que proporcionará compatibilidad hacia atrás si la tarjeta se emplea como SIM en redes 2G. Las especificaciones de UICC son suficientemente abiertas como para poder alojar aplicaciones de propósito más general que las destinadas a las telecomunicaciones móviles.

Tanto la SIM como la USIM, vienen especificadas por: unas características físicas, una interfaz eléctrica con el terminal, mecanismos de establecimiento de la comunicación y transporte, el modelo lógico de la tarjeta, así como los comandos de ficheros y procedimientos independientes de la aplicación. Todas estas especificaciones se encuentran en la norma GSM 11.11 y en TS 31.102.

6.1.2.1 Características físicas y eléctricas

En cuanto a las dimensiones físicas de la ICC/SIM y la UICC/USIM, existen dos versiones:

- ID-1 : que posee las dimensiones de una tarjeta de crédito.
- Plug-in ó microtarjeta (con unas dimensiones de 25x15 milímetros). En la siguiente figura se muestran la forma y un corte transversal de este tipo de tarjetas:

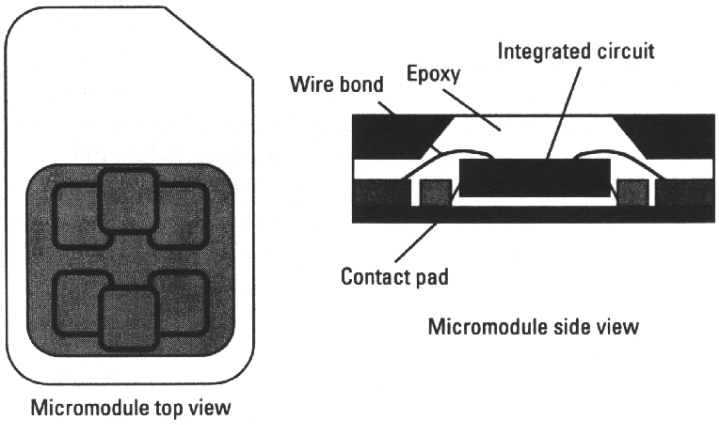


Figura 37: Forma y corte transversal de una tarjeta SIM plug-in

Ambas versiones están definidas en la ISO/IEC 7816-1,2,3. Algunos fabricantes han propuesto la estandarización de tarjetas de tamaño inferior al Plug-in, denominado "the third size".

Respecto a las características de las tarjetas, existen tres modos de operación: 5V, 3V y 1,8V, con unos consumos máximos de corrientes asociados. El terminal GSM ó UMTS intentará activar la tarjeta desde el voltaje más bajo hasta obtener una respuesta estable. En GSM se suelen utilizar las tarjetas que se alimentan con 5 ó 3 V, mientras que en UMTS se utilizan 3V ó 1,8V.

6.1.2.2 Protocolos de comunicación

Como ya se sabe por la sección 3.2.4.4 Protocolo TPDU de la página 33, hay dos protocolos de comunicación entre la tarjeta y el terminal: el protocolo T=0 y el protocolo T=1. El protocolo T=0 se utiliza tradicionalmente en GSM. Tanto para las tarjetas UICC como para los terminales UMTS, el protocolo T=0 es obligatorio, mientras que el protocolo T=1 es obligatorio para el terminal pero no para tarjeta.

6.1.2.3 Estructura lógica

El diálogo de alto nivel entre el terminal y la tarjeta hace referencia a una estructura lógica basada en ficheros, con condiciones de acceso asociadas. Los conceptos empleados en 2G como en 3G son los mismos. Sin embargo en 3G la estructura es genérica y abierta, soportando la noción de aplicación (que también se soporta en fases de desarrollo más tardías de la tarjeta SIM).

La estructura lógica se basa en un árbol de directorios y ficheros. Los directorios pueden contener otros directorios o ficheros, y hacen agrupaciones lógicas de estos elementos basándose en funcionalidades asociadas. Como se vió en la sección 3.2.5.1 Sistemas de ficheros de las smart card (página 34), existen dos tipos de directorios: el

directorio MF y el directorio DF. Además se define un directorio de aplicación o ADF, que es la raíz absoluta dentro del dominio de una aplicación. En los ficheros se almacena la información necesaria para dar soporte a los servicios y funcionalidades de la tarjeta.

Todos los ficheros de la tarjeta tienen un FID (File Identifier) asociado, que junto a su ruta de acceso lo definen unívocamente. Adicionalmente, los ficheros tienen unas condiciones de acceso respecto a cada una de las operaciones que se pueden realizar sobre ellos. Para cada fichero, se puede definir una matriz que asocia cada operación (lectura, escritura, borrado, etc.) con la condición necesaria para poder ejecutar la operación: Always, PIN, clave administrativa, Never, etc. Por ejemplo, para poder leer un registro de la agenda se necesita modificar el PIN. Para modificar el IMSI de la tarjeta, la condición es Never, por lo que nunca se podrá modificar utilizando mecanismos convencionales. En la Figura 38 se puede observar la organización jerárquica de los ficheros de los que consta la tarjeta SIM.

Como puede observarse, bajo el directorio raíz (MF) existen dos ficheros y dos directorios. El fichero EF ICCID almacena el identificador de la tarjeta de circuitos integrados (ICC: Integrated Circuits Card) que es único y no modificable. El fichero EF ELP define los lenguajes preferidos del usuario en orden de prioridad. El directorio DF GSM contiene la aplicación GSM y el directorio DF TELECOM almacena información relacionada con los servicios que ofrece la operadora de telecomunicación a las aplicaciones presentes en la tarjeta (DF ADN: agenda telefónica, DF SMS: almacén de SMS, DF LND: últimos números marcados, DF MSISDN (Mobile Station International ISDN Number): contiene el número asociado al abonado, etc.).

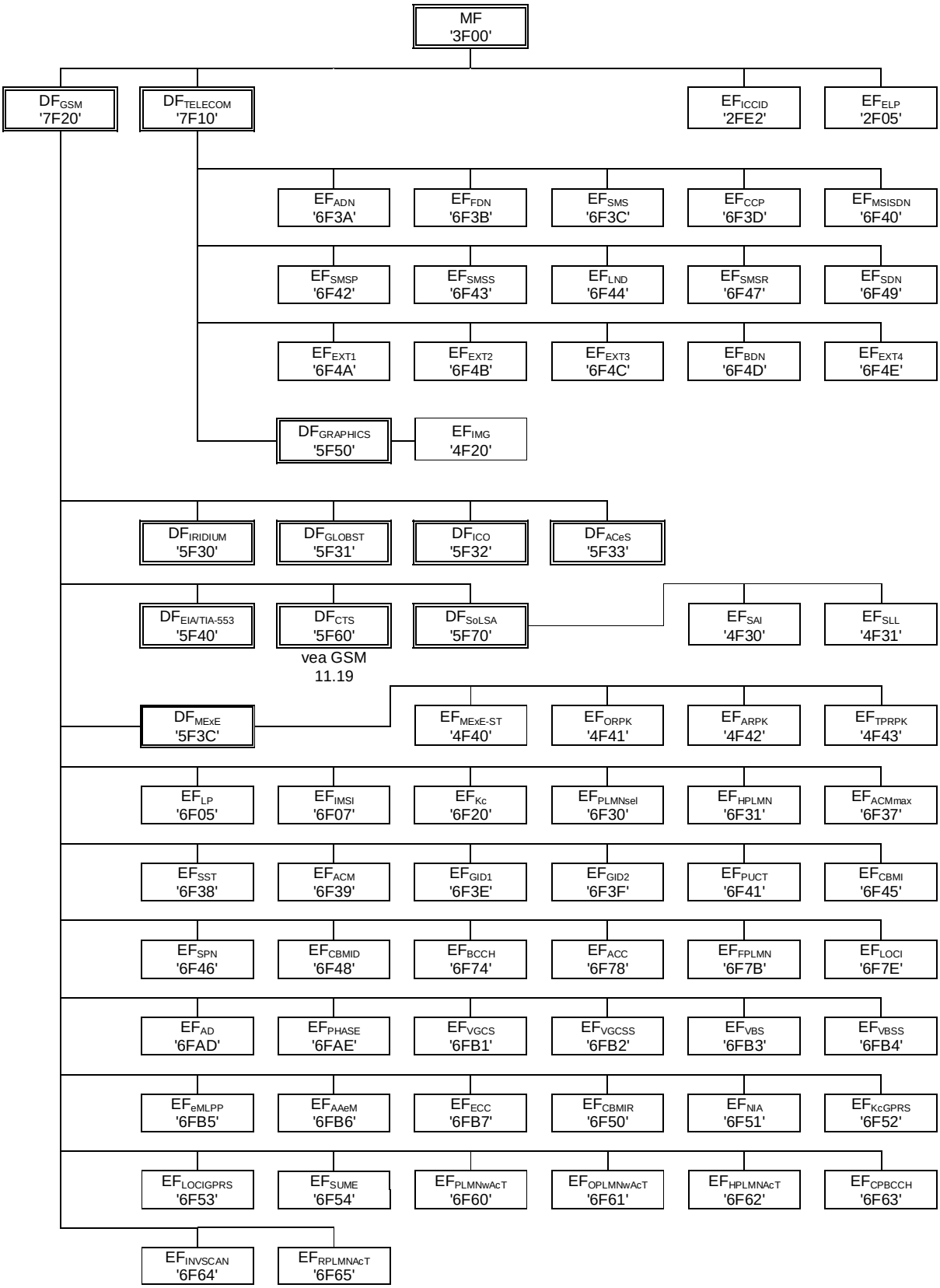


Figura 38: Estructura de ficheros de la tarjeta SIM

La estructura lógica genérica de la UICC, en el mundo UMTS, es la siguiente:

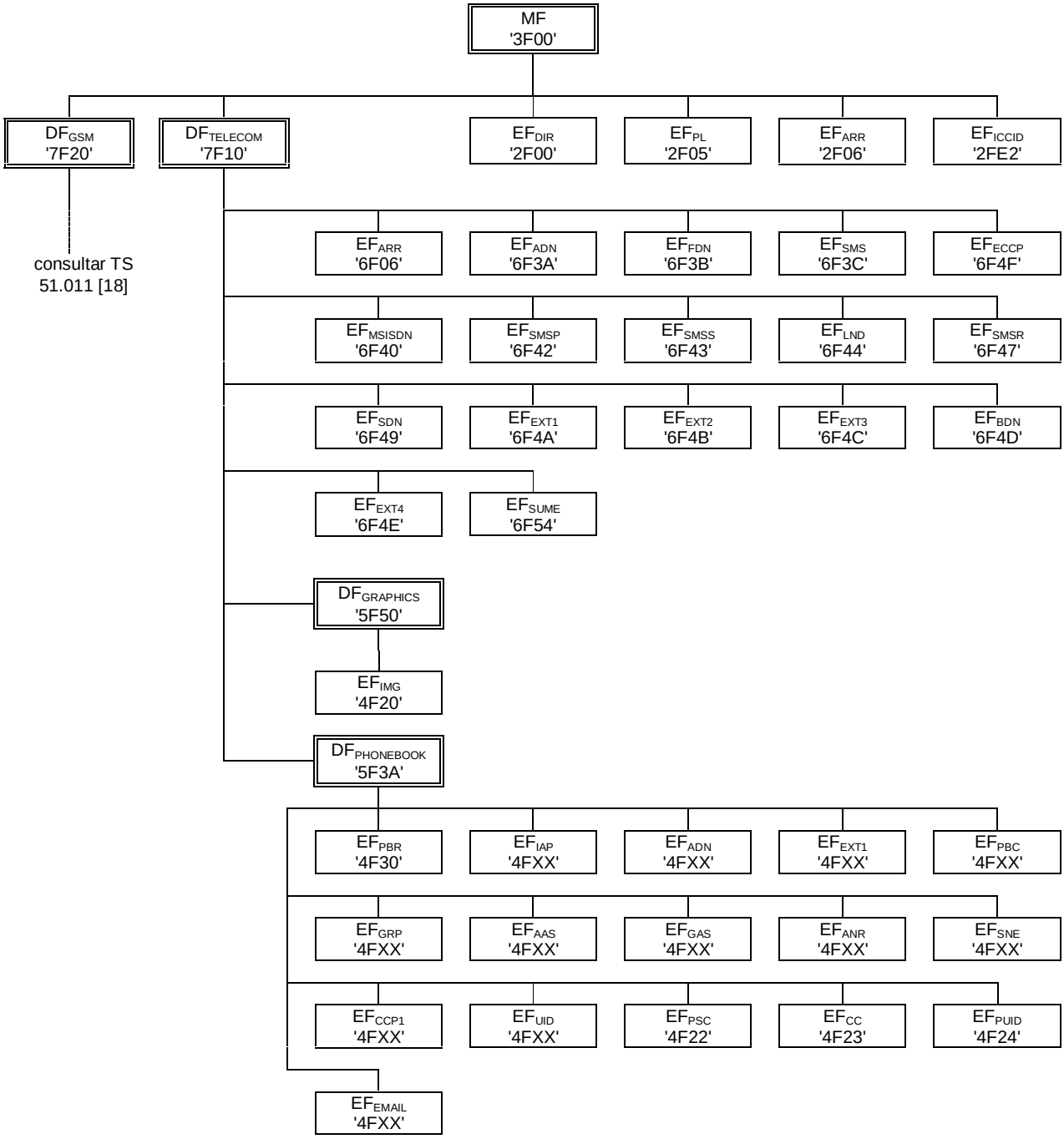


Figura 39: Estructura lógica de la UICC

Esta estructura no difiere mucho de la presentada para la tarjeta SIM. En la UICC existen tres ficheros: EF DIR, donde se almacenan las referencias (AID's) de las aplicaciones presentes en la tarjeta; EF ICCID donde se almacena el identificador único de la tarjeta y EF PL con la lista de lenguas predefinidas. También dispone de un directorio DF TELECOM, donde se almacena toda la información considerada común a todas las aplicaciones de la tarjeta (con una agenda telefónica mejorada) y tantos ADF's como aplicaciones estén presentes en la tarjeta.

6.1.2.4 Operaciones

Sobre las estructuras lógicas presentadas, se pueden realizar un conjunto de operaciones estándar. Estas operaciones, pueden ser invocadas por el terminal así como ejecutadas remotamente por parte del operador para poder realizar tareas administrativas sobre los clientes. Todas las operaciones se basan en el intercambio de APDU's, que serán transportadas convenientemente al protocolo de nivel físico que acuerden las dos entidades, tarjeta y terminal.

Adicionalmente, existen otras operaciones dependientes de la aplicación. Por ejemplo, la solicitud de mecanismos de autenticación que solicita la red al cliente depende de la aplicación que se esté ejecutando en la tarjeta (GSM, USIM u otras). Asimismo, cada proveedor de tarjetas puede proporcionar operaciones administrativas propietarias para gestionar funcionalidades o capacidades no estandarizadas.

Las operaciones típicas, independientes de la aplicación, especificadas por el 3GPP son las siguientes:

- SELECT; STATUS; GET RESPONSE: son operaciones lógicas que se utilizan para navegar por la estructura lógica de la tarjeta, obtener características de ficheros y realizar operaciones de intercambio de información.
- READ BINARY, READ RECORD, UPDATE BINARY, UPDATE RECORD, SEARCH RECORD, INCREASE: son operaciones de lectura y escritura, que dependen del tipo de fichero (lineal, cíclico o transparente).
- VERIFY PIN, CHANGE PIN, DISABLE PIN, ENABLE PIN, UNBLOCK PIN: son operaciones administrativas relacionadas con la gestión de la clave personal.
- TERMINAL PROFILE; ENVELOPE; FETCH; TERMINAL RESPONSE: son comandos necesarios para el soporte de los comandos proactivos de las tarjetas SIM de la fase 2+. A continuación se detallan estos comandos:
    - TERMINAL PROFILE: el ME usa esta función para transmitir a la (U)SIM sus capacidades (o perfil) relacionadas con la funcionalidad (U)SIM Application Toolkit. La (U)SIM no tiene que responder con ningún dato.
    - ENVELOPE: Esta función se utiliza para transferir datos a las aplicaciones (U)SIM Application Toolkit que se encuentran en la SIM/USIM. La (U)SIM recibe una cadena de datos y responde con una estructura de datos definida en la norma 11.14 (GSM) o en la 31.11 (UMTS).
    - FETCH: Esta función se usa para transferir un comando (U)SIM Application Toolkit desde la SIM al ME. En la C-APDU emitida por el ME no se transmite ningún dato. La respuesta dada por la (U)SIM contiene un comando SAT dirigido al ME.
    - TERMINAL RESPONSE: Esta función se usa para transferir desde el ME a la SIM, la respuesta a un comando (U)SIM Toolkit previo. Los datos de la APDU de comando transmitida por el ME contienen

una cadena de bytes que se trata de la respuesta al anterior comando SAT e indica si la ejecución del comando ha sido exitosa o ha fallado. La tarjeta SIM no responderá con ningún dato.

- AUTHENTICATE: con esta operación, definida en UMTS, se generan las claves de cifrado e integridad, así como la respuesta a la solicitud de la red. Sustituye al antiguo comando RUN GSM ALGORITHM, y se comporta como éste en la autenticación dentro de una red GSM.

Las operaciones que se han mostrado son las que se usan en UMTS. En GSM se usan las mismas operaciones (en algunos comandos, los nombres cambian). Cada uno de estos comandos tiene una APDU asociada. Por ejemplo para el comando SELECT, la APDU correspondiente es la siguiente:

CLA	INS	P1	P2	Lc	Campo de datos	Le
A0	A4	00	00	02	FID	depende

Tabla 12: APDU asociada al comando SELECT

A este comando, la SIM responde con una R-APDU, cuya estructura depende de si el FID corresponde a un directorio o a un fichero. Para más información vea la norma GSM 11.11 ó la TS 31.102.

6.1.3 APLICACIÓN SIM/USIM

En esta sección se va a describir la aplicación más importante que se va a poder ejecutar sobre la plataforma ICC (en GSM) y UICC (en UMTS): la aplicación SIM ó USIM (según estemos hablando del mundo GSM ó UMTS). La aplicación SIM se describe en la norma GSM 11.11, mientras que la aplicación USIM se especifica en la norma TS 31.102 del 3GPP. La aplicación USIM es un superconjunto de la tarjeta SIM, en contenidos, funcionalidades y mecanismos de seguridad. Todos los elementos presentes en la USIM se pueden particularizar convenientemente para transformarla en una tarjeta SIM válida en una red GSM/GPRS.

La estructura de directorios de la tarjeta SIM ya se mostró en la Figura 38 de la página 145. El directorio de la aplicación GSM (DF GSM) está contenido en el directorio principal o MF de la tarjeta. En DF GSM se encuentran todos los contenidos relacionados con la aplicación GSM. A continuación se explicará la función de algunos ficheros elementales de DF GSM:

- EF LP: contiene la lista de las lenguas preferidas (LP: Language Preference) por orden de prioridad, de forma que el usuario pueda escoger sus preferencias. El ME quizás use esta información para propósitos que tengan que ver con MMI (Man Machine Interface) o para presentar en pantalla los mensajes de difusión en la lengua preferida.
- EF LOCI: contiene información de localización (LOCI: Location Information). Los datos guardados son: el TMSI (Temporary Mobile Subscriber Identity), el temporizador que controla las actualizaciones de posición periódicas (indica cada cuanto debe realizarse una actualización de posición periódica), el estado de la actualización (si se ha actualizado o no), si la PLMN o el área de localización no son permitidas, etc.

- EF IMSI: contiene el IMSI (International Mobile Subscriber Identity) del usuario.
- EF PLMNsel: contiene una lista de PLMN's (Public Land Mobile Network) en orden de preferencia. Esta información la determina el usuario o el operador de red.
- EF FPLMN: contiene los códigos de redes móviles prohibidas (FPLMN: Forbidden PLMN).
- EF SST: contiene la tabla de servicios de la SIM, SST (SIM Service Table), es decir, indica los servicios que han sido definidos en la SIM y dentro de éstos, cuáles están activos. Los servicios a los que se refiere son por ejemplo, ADN, FDN, SMS, CCP, PLMN Selector, LND, MSISDN, etc.

En cuanto a la aplicación USIM, ésta también posee su propio DF (que se encuentra en el MF). En la Figura 40 se muestra el contenido del ADF (directorio, DF, de una aplicación en el mundo UMTS) de la aplicación USIM.

Igual que en la UICC existe la posibilidad de almacenar distintas aplicaciones en el ADF USIM. La USIM tiene un esquema parecido (aunque evolucionado) del existente en una tarjeta SIM convencional y el operador o terceros desarrolladores pueden crear aplicaciones y servicios sobre la USIM, apoyándose en una arquitectura abierta y en el estándar USAT.

La USIM incluye todas las funcionalidades presentes en una tarjeta SIM fase 2+ y se les da un soporte más adecuado, destacando la gestión y registro de las llamadas entrantes y salientes, el incremento de capacidades de la agenda (se ha creado un DF PHONEBOOK), el soporte a los servicios SOLSA y el almacenamiento de imágenes en la tarjeta. A continuación se describirá algunos directorios o archivos que se encuentran en la aplicación USIM. Hay que mencionar que las descripciones realizadas de algunos de los archivos de la aplicación GSM, también son válidas en la aplicación USIM:

- EF LI: contiene la lista de las lenguas preferidas (LI: Language Indication) por orden de prioridad, de forma que el usuario pueda escoger sus preferencias. El ME quizás use esta información para propositos que tengan que ver con MMI (Man Machine Interface) o para presentar en pantalla los mensajes de difusión en la lengua preferida.
- EF UST: este EF indica qué servicios están disponibles. Si un servicio no está inidicado como disponible en la USIM, el ME no seleccionará ese servicio.

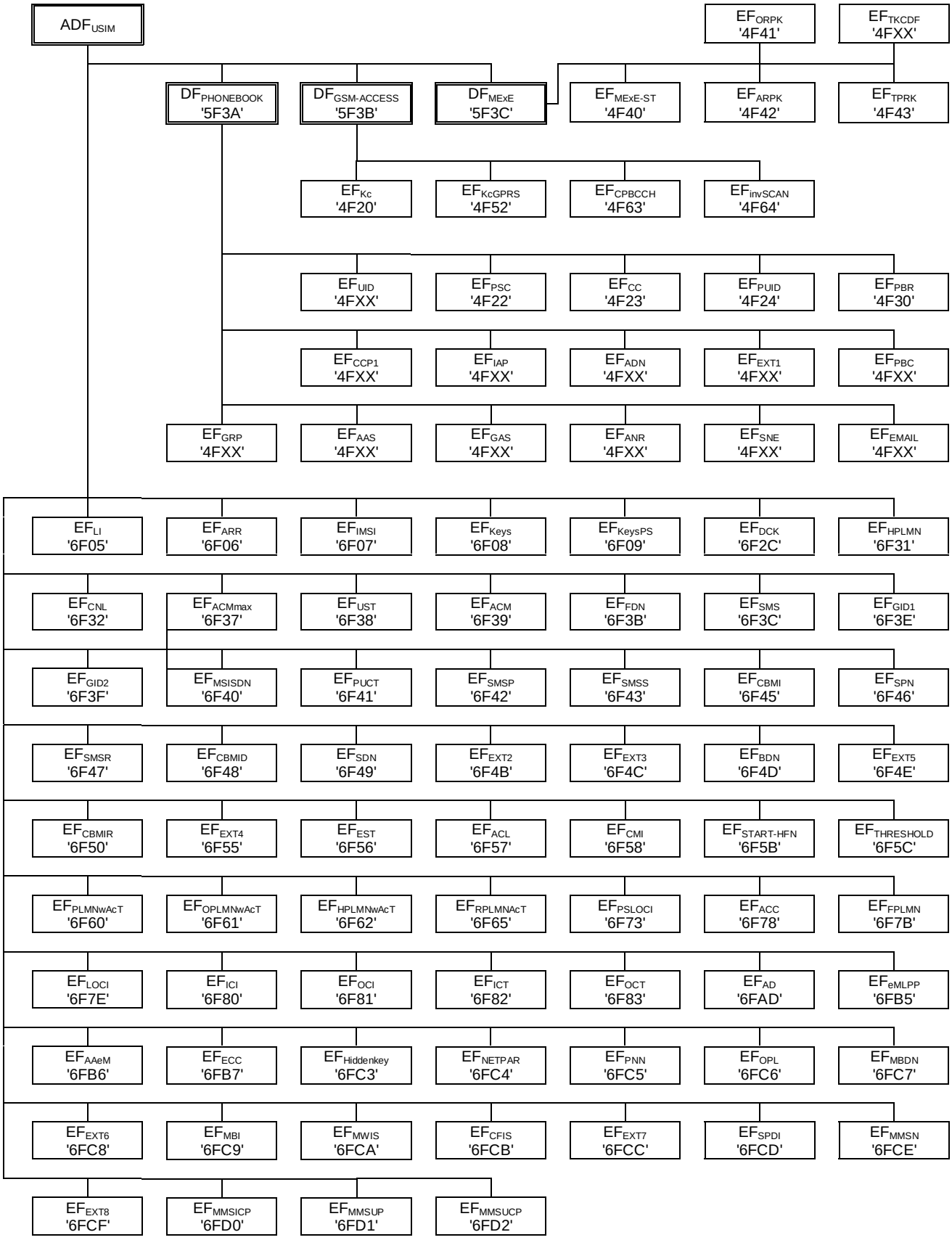


Figura 40: Estructura de ficheros de la aplicación USIM

- DF PHONEBOOK: la UICC quizás contenga una agenda telefónica global (bajo DF TELECOM) o una agenda específica (bajo cada ADF USIM que

exista en la tarjeta) por cada aplicación o ambas a la vez. Cuando ambos tipos de agendas telefónicas coexisten, son independientes y no comparten datos. En este caso, el usuario podrá seleccionar la agenda telefónica a la que quiere acceder. Cada agenda telefónica específica está protegida por el PIN de la aplicación.

- DF GSM-ACCESS: la aplicación SIM requiere los EF's que se encuentran bajo este directorio para que sea capaz de acceder al servicio a través de una red GSM.

6.1.4 SAT/USAT: (UNIVERSAL) SIM APPLICATION TOOLKIT

Como se ha mencionado anteriormente, las especificaciones iniciales de GSM dotaban a la tarjeta SIM de una funcionalidad limitada. Básicamente, ofrecían soporte a la seguridad y almacenamiento portable entre terminales de los datos y parámetros de configuración más importantes.

Debido al incremento de capacidad de la tarjeta inteligente, tanto en memoria como en capacidad de proceso, se decidió que podría ser interesante ejecutar aplicaciones en la tarjeta, con el propósito de dar cabida a servicios de valor añadido (VAS: Value-Added-Services). Estas aplicaciones contarían con el apoyo el terminal como interfaz con el usuario y con la red. Bajo este escenario apareció SIM Application Toolkit (SAT). Esta nueva especificación GSM 11.14 del ETSI definió, en primer lugar, un conjunto de APDU's o comandos a nivel de aplicación que permite que la tarjeta tenga un comportamiento proactivo, capaz de tomar la iniciativa frente al terminal ó ME (Mobile Equipment). Sobre estos comandos, se construye un API para el control de los recursos del terminal (teclado, auricular, pantalla, etc.), así como mecanismos de disparo de eventos que pudieran ser de interés para las aplicaciones.

En el lado de la tarjeta, se dispone de un sistema operativo, sobre el que es posible el desarrollo de aplicaciones que utilizarán esta interfaz. Asimismo, se ofrece la posibilidad de atender a eventos del tipo "nuevo mensaje corto recibido" o "establecimiento de una llamada". La lógica de la aplicación puede ser tan compleja como se desee. El aspecto de las aplicaciones es de tipo textual (eventualmente con iconos asociados) basado en menús y preguntas al usuario, siendo el formato de representación dependiente del terminal. Se permite la intervención en los procedimientos de establecimiento de llamada, la emisión de sonidos a través del auricular del terminal móvil y el envío de mensajes cortos, USSD's, SS y DTMF entre otras funcionalidades.

Debido a que los ME's tienen distintas características (según fabricantes y modelos), el SIM Application Toolkit ofrece la posibilidad de conocer las funcionalidades soportadas por el ME (PROFILE DOWNLOAD). Esto quiere decir que una vez que se conozcan las características del equipo móvil donde correrá la aplicación, se puede programar la captura de los eventos de interés y programar para ellos operaciones basadas o no en la interacción con el ME a través de los comandos proactivos.

Junto a estos comandos, se definen los protocolos necesarios para la administración remota, por parte del operador, de las aplicaciones instaladas en la tarjeta. El conjunto de tecnologías encargadas de la gestión remota de la tarjeta,

utilizando la insfraestructura de comunicaciones móviles, se denomina OTA (Over The Air). El mecanismo tradicional y más importante para la gestión de tarjetas es el mensaje corto, sobre el que se montan los protocolos adecuados que permiten garantizar que la comunicación que se establece entre el operador y la tarjeta es segura para ambas partes. El operador o el propio usuario puede cargar, borrar y actualizar las aplicaciones residentes en la tarjeta, gracias (en parte) al OTA.

El SIM Application Toolkit es actualmente uno de los mecanismos más potentes de los que dispone el operador de telefonía móvil para diferenciarse en sus productos y servicios. Permite desarrollar interfaces amigables con los servicios de información bajo demanda, y construir aplicaciones a la medida de nichos de mercado. Es un soporte muy adecuado para aplicaciones de comercio electrónico o banca móvil. Además, es la base de otras tecnologías como Dual Slot, que permite crear aplicaciones en la SIM capaces de controlar otras tarjetas de tipo EMV o monedero electrónico. Algunos ejemplos de aplicaciones pueden ser los siguientes:

- Activación y control de servicios de prepago.
- Servicios de información.
- Servicios de directorio y de guía telefónica.
- Servicios bancarios y de transacciones a través del móvil. En este tipo de aplicaciones se requieren ciertas medidas de seguridad. Para permitir el acceso a la aplicación bancaria se debe introducir un PIN. Si éste es válido la tarjeta SIM puede iniciar la aplicación bancaria. Además, los mensajes que se establecen entre la red y el terminal móvil se deben proteger mediante algoritmos de codificación como triple DES (muy usado en el mundo bancario).
- Correo electrónico.
- Servicios de localización. El funcionamiento de esta clase de servicios consiste en que después de que el usuario haya indicado el servicio requerido, la tarjeta SIM solicita al ME que le indique la información del área (localización). A continuación la tarjeta SIM envía un mensaje corto a la red con la información del área. La red responde con la situación (nombre de la calle) del servicio requerido, más cercana.

Es importante destacar que en la R99 (Release 99) de las especificaciones 3GPP se han reproducido las especificaciones del ETSI para tarjetas SIM de la fase 2+. En este sentido, se comprueba cómo no existe ninguna aportación nueva y no se puede hablar estrictamente de SIM Application Toolkit 3G.

6.1.4.1 Comandos SIM Application Toolkit

A continuación se describen, de forma breve, los grupos de comandos disponibles en el estándar para comprender el alcance de esta tecnología. Los nombres de cada uno de los comandos son suficientemente indicativos para comprender aproximadamente cual es su función:

- Interacción con el usuario: con este grupo de comandos es posible mostrar textos en pantalla, configurar menús y formular preguntas al usuario. El grupo de comandos es el siguiente:
  - DISPLAY TEXT: visualiza texto(de menos de 160 caracteres) en la pantalla.
  - GET INKEY: envía texto a la pantalla y solicita una respuesta de un solo carácter.
  - GET INPUT: envía texto a la pantalla y solicita una respuesta.
  - PLAY TONE: reproduce un tono.
  - SELECT ITEM: suministra una lista de opciones y recibe la elección del usuario.
  - SET UP MENU: suministra las opciones de un submenú nuevo para que se añadan al menú del terminal móvil.
  - LANGUAGE NOTIFICATION: la SIM usa este comando para notificarle al ME la lengua que se está usando actualmente en los comandos proactivos o en las respuestas a los comandos.
- Integración con la red: estos comandos permiten transmitir información encapsulada en mensajes cortos o USSD's (Unstructured Suplementary Service Data). Los procesos de establecimiento de llamada también están contemplados. El conjunto de comandos es:
  - OPEN CHANNEL: con este comando proactivo la SIM solicita al ME que abra un canal de datos con los parámetros indicados en el comando. Se pueden abrir diferentes tipos de canales según el portador usado para enviar los datos (también especificado en el comando proactivo):
    - CSD (Circuit Switched Data).
    - GPRS (General Packet Radio Service).
    - El portador por defecto de la red.
  - CLOSE CHANNEL: solicita al ME que cierre el canal de datos especificado en el comando proactivo por un identificador de canal.
  - SEND DATA: este comando proactivo, solicita al ME que envíe datos a través de un canal anteriormente establecido identificado por un identificador de canal.
  - RECEIVE DATA: la SIM solicita al ME que se le entregue un número especificado de bytes procedentes de un canal especificado por su identificador.
  - GET CHANNEL STATUS: este comando proactivo solicita al ME que devuelva un objeto de estado de los identificadores de canales especificados. El ME devolverá la información solicitada concerniente a los canales en un comando TERMINAL RESPONSE.

  - SEND DTMF: solicita al ME el envío de un tono DTMF (Dual Tone Multiple Frequency) durante el establecimiento de una llamada.
  - SEND SHORT MESSAGE: envía un mensaje corto a la red de telefonía móvil.
  - SEND SS: envía una SS (Supplementary String) a la red de telefonía móvil.
  - SEND USSD: envía una USSD (Unstructured Supplementary Services Data) a la red de telefonía móvil.
  - SET UP CALL: inicia una llamada de voz.
  - PROVIDE LOCAL INFORMATION: este comando solicita al ME que pase información local (como pueden ser el código de país o de red móvil) a la SIM.

- Interacción con el terminal: el resto de comandos disponibles sirven para configurar el comportamiento del terminal en relación con la tarjeta:
  - MORE TIME: solicita más tiempo para llevar a cabo una tarea.
  - POLL INTERVAL: con este comando la SIM le indica al ME con que frecuencia quiere que el ME envíe datos de estado durante el modo en espera. Este estado se desactiva con el comando POLLING OFF.
  - REFRESH: con este comando la SIM solicita al ME que lleve a cabo una inicialización de la SIM (según la norma GSM 11.11), y/o que avise al ME de que los contenidos o la estructuras de los EF's de la SIM han cambiado.
  - SET UP EVENT LIST: la SIM utiliza este comando para suministrar un conjunto de eventos. El ME será el encargado de monitorizar los eventos (informar de cuando se producen) que se encuentran en dicho conjunto de eventos.
  - TIMER MANAGEMENT: este comando solicita al ME que gestione un contador de la forma en la que se describe en el comando (empezar, desactivar y obtener el valor actual). En el caso de comenzar una cuenta, en el comando se indica la duración.
  - SET UP IDLE MODE TEXT: con este comando, la SIM suministra una cadena de texto que el ME mostrará en su pantalla como texto que indica que está en modo de espera (si el ME es capaz de hacerlo). El estilo de presentación se deja como decisión del fabricante del ME. Este testo se mostrará en pantalla de manera que no afecte al nombre de la red ni al nombre de los proveedores del servicio.
  - RUN AT COMMAND: envía comandos AT al módem del teléfono, si está disponible.

- Interacción con otras tarjetas: la SIM es capaz de intercambiar mensajes de nivel de aplicación con otras tarjetas, siempre y cuando el terminal

incorpore el hardware necesario para gestionar más de una tarjeta. Los desarrollos del Dual Slot se basan en estos comandos. La lista de comandos es:

- GET READER STATUS: este comando da información acerca de lectores de tarjetas y tarjetas insertadas adicionales (si el ME soporta la clase "a"). Esta información está restringida al estado y al identificador del lector de tarjetas. El ME devolverá la información solicitada con el comando TERMINAL RESPONSE (emitido por el ME).
- PERFORM CARD APDU: solicita al ME el envío de una APDU de comando a la tarjeta adicional (si soporta la clase "a"). Este comando es compatible con cualquier protocolo que se utilice entre el ME y la tarjeta adicional.
- POWER ON/OFF CARD: este comando solicita al ME que empiece/acabe una sesión con la tarjeta adicional.

6.1.4.2 Nuevos comandos y tendencias de SIM Application Toolkit

En la R99 de las especificaciones de SAT/USAT no se incluyeron algunos comandos presentes en las normas del ETSI, aunque se han incorporado en versiones de las especificaciones 3GPP. Estos nuevos comandos son parte de las dos tendencias hacia las que se dirige SIM Application Toolkit y el mundo de la tarjeta en general:

- Integración con la tecnología WAP: SIM Application Toolkit y Wireless Application Protocol son dos tecnologías complementarias, que pueden llevar al operador a crear servicios avanzados apoyándose en las ventajas de las dos. En este sentido aparece el nuevo comando LAUNCH BROWSER, que permite iniciar una sesión del navegador residente en el terminal con unos parámetros asociados (configuración, URL, ...). Aunque WAP y SAT son dos tecnologías complementarias, SAT se suele usar en aplicaciones que necesitan un alto grado de seguridad (como la banca móvil y servicios que hacen uso de información estática como guías telefónicas, directorios de compañías y páginas amarillas). WAP se suele utilizar en servicios más dinámicos, como la navegación por Internet y el acceso a información que cambia con frecuencia.
- Control de canales de datos desde la tarjeta SIM: a través del Bearer Independent Protocol (BIP), se pretende alcanzar un control total sobre los distintos portadores de datos accesibles desde el ME. Como ya se ha comentado anteriormente, una aplicación en tarjeta interacciona con la red a través del servicio de mensajes cortos (y en menor medida a través de USSD's). Con este nuevo conjunto de comandos (OPEN CHANNEL, CLOSE CHANNEL, SEND DATA, RECEIVE DATA y GET CHANNEL STATUS), la tarjeta SIM puede llegar a un protocolo de nivel de aplicación con un servidor, utilizando como portadores los servicios de conmutación de circuitos o paquetes (GSM CSD ó GPRS). Los detalles de establecimiento de llamada, gestión de movilidad, etc. y demás detalles de los niveles bajos de la comunicación son transparentes a la tarjeta y controlados por el terminal. Con esta tecnología se abre una puerta para el

desarrollo de servicios avanzados en la SIM, así como una optimización de los mecanismos de gestión remota de la tarjeta.

6.1.4.3 Procedimiento de envío de comandos

Esta sección explica cual es el mecanismo que hace que la tarjeta SIM pueda enviar comandos al ME. Esto le puede parecer contradictorio, ya que hasta lo visto, las smart cards son las que reciben los comandos procedentes del CAD.

El mecanismo de envío de comandos proactivos sigue usando el protocolo de transporte T=0, pero añade nuevas SW's. Estas SW's tienen el mismo significado que el fin normal '9000' y también se pueden usar con la mayoría de los comandos que responden con un final normal '9000', pero también permite que la tarjeta SIM le diga al ME que "tengo información que enviarte". Entonces, el ME usa la función FETCH para recoger y averiguar de qué información se trata (un comando proactivo). Este tipo de funcionalidades solo se usarán entre una SIM proactiva y un ME que soporte la característica proactiva de la SIM.

6.1.4.3.1 Identificación del servicio SIM Application Toolkit

Una tarjeta SIM proactiva se identifica por tener el servicio SIM proactiva activado en el EF SST (SIM Service Table). Un ME que soporte SIM's proactivas se identifica como tal, cuando envíe el comando TERMINAL PROFILE durante el procedimiento de inicialización de la SIM. Entonces el ME envía comandos STATUS a la SIM, en intervalos dados por el procedimiento de intervalo de polling. Así la SIM puede responder con la palabra de estado '91XX' y luego enviar un comando proactivo (como ya se indicará más adelante).

Una tarjeta SIM no enviará comandos proactivos (mediante la palabra de estado '91XX') a un móvil que no soporte la característica SIM proactiva. Un ME que soporte la característica SIM proactiva no enviará los comandos relacionados, a una SIM que no tenga este servicio activado.

6.1.4.3.2 Procedimiento general

Para todos los procedimientos que pueden terminar en '9000' (fin normal de un comando) y que no pueden terminar en '9FXX' (datos disponibles en la SIM), una SIM proactiva, funcionando con un ME que soporta SIM's proactivas, en su lugar usará la respuesta de estado '91XX'. El código de respuesta '91XX' indicará al ME que la SIM ha ejecutado con éxito el comando previo, de la misma manera que lo hace la palabra de estado '9000' (=> OK), pero además indicará que tiene datos de respuesta que contienen un comando proactivo de la SIM para un procedimiento particular del ME.

El valor XX indica la longitud de los datos de respuesta. El ME usará el comando FETCH para obtener estos datos. Es responsabilidad de la SIM recordarle al ME que tiene un comando proactivo pendiente, usando el código '91XX' hasta que el ME lo recupere con FETCH.

En GSM 11.11 se indica como la SIM puede iniciar un comando proactivo en cada uno de los cinco casos del protocolo APDU que expone dicha norma (casos

compuestos de los 4 vistos en el apartado 3.2.4.3 El protocolo APDU). Algunos comandos requieren que la SIM indique que tiene datos de respuesta para el ME (a través de la palabra de estado SW1/SW2 = '9FXX'), y que el ME los consiga usando el comando GET RESPONSE. Cuando el ME ha recibido un comando de la SIM, intentará procesar el comando inmediatamente. Cuando termine (con éxito o con fallo) informará a la SIM (con un código) tan pronto como pueda, usando TERMINAL RESPONSE.

La responsabilidad de reintento en caso de fallo del comando, recae en la aplicación SIM. La aplicación SIM puede juzgar si enviar el mismo comando otra vez, enviar uno diferente, o no intentarlo, según la información dada en el comando TERMINAL RESPONSE. En todo caso, solo se permite la ejecución de un comando proactivo a la vez.

6.1.4.3.3 Estructura de las comunicaciones del SIM Application Toolkit

Los comandos y respuestas del SIM Application Toolkit se envían a través de la interfaz como objetos de datos BER-TLV (objetos que cumplen las Basic Encoding Rules de ASN.1). Cada ADPU solo contiene un objeto BER-TLV.

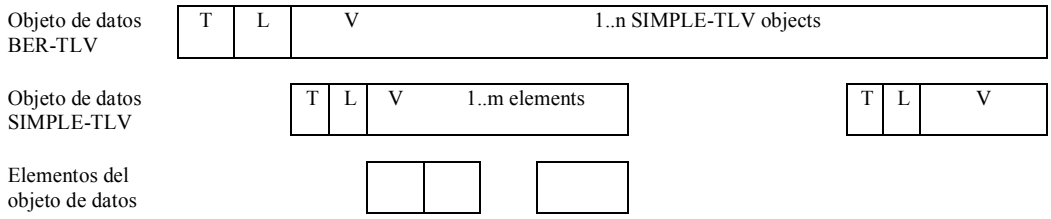


Figura 41: Estructura de los objetos de datos

La etiqueta (T=Tag) es un valor constante, de un byte de longitud, e indica que es un comando del SIM Application Toolkit. La longitud (L) se codifica en 1 ó 2 bytes según la norma ISO/IEC 7816-6. La siguiente tabla detalla la codificación:

Tabla 13: Longitudes de los objetos de datos BER-TLV

Length	Byte 1	Byte 2
0-127	longitud (de '00' a '7F')	no presente
128-255	'81'	length ('80' a 'FF')

De este modo se puede codificar en dos bytes cualquier longitud dentro de los límites de la APDU (por encima de los 255). Se elige esta codificación para mantener la compatibilidad con la norma ISO/IEC 7816-6.Cualquier valor para los bytes 1 y 2 que no sea el mostrado en la tabla de arriba, se tratará como un error y se rechazará el mensaje entero.

La parte de "valor" (V) del objeto de datos BER-TLV, se compone de objetos de datos SIMPLE-TLV, como se muestra en la descripción de los objetos de datos SIMPLE-TLV de los comandos individuales. Es obligatorio, para los objetos de datos SIMPLE-TLV, que sean suministrados en el orden dado por la descripción de cada comando. Los objetos de datos SIMPLE-TLV nuevos, se pueden añadir al final de un comando.

6.1.4.4 Transferencia de datos entre la red y la SIM

Para que la tarjeta SIM envíe datos a la red y reciba datos de la red, puede utilizar comandos dedicados o el BIP. La transferencia de información en la interfaz SIM-ME se lleva a cabo mediante el comando ENVELOPE.

Los comandos dedicados sirven para usar los mecanismos de transporte basados en SMS's punto a punto (SMS-PP), SMS Cell Broadcast (SMS-CB) y USSD (que no se verá). Los mecanismos más usados son los basados en el mensaje corto (SMS).

En cuanto al Bearer Independent Protocol (BIP), hay dos mecanismos de transporte: CSD (Circuit Switched Data) y GPRS (General Packet Radio Service). Los comandos proactivos OPEN CHANNEL, CLOSE CHANNEL, SEND DATA, RECEIVE DATA y GET CHANNEL STATUS y los eventos Data available y Channel status permiten a la SIM establecer un canal de datos con el ME, y a través del ME con un servidor remoto de la red. La SIM provee información al ME para que seleccione un BIP en el momento del establecimiento del canal. Entonces, el ME permite que la SIM y el servidor intercambien datos por ese canal de forma transparente. La SIM usa el servicio del nivel más bajo del ME para enviar unidades de datos del servicio (SDU) al ME. Este nivel más bajo es el nivel más alto del BIP seleccionado.

6.1.4.4.1 SMS

La SIM podrá enviar mensajes cortos a la red mediante la utilización del comando proactivo SEND SHORT MESSAGE.

En cuanto al envío de mensajes por parte de la red, ésta podrá utilizar los servicios SMS-PP ó SMS-CB. Para recibir los mensajes, los servicios deben estar activados en el EF SST (SIM Service Table). Cuando el ME recibe un mensaje SMS-PP, se lo pasa a la SIM de forma transparente usando el comando ENVELOPE. Si el móvil no soporta la descarga ó recepción de mensajes, el mensaje se guardará en EF SMS (como si fuera un SMS normal). Tras la recepción de un mensaje, la SIM responderá o no a la red. En el caso de que el ME reciba un mensaje SMS-CB, el ME comparará el identificador del mensaje Cell Broadcast con los identificadores de mensajes contenidos en EF CBMID. Si el identificador de mensaje se encuentra en EF CBMID, se pasa la página Cell Broadcast a la SIM, usando el comando ENVELOPE y el ME no mostrará el mensaje por pantalla. Si el identificador de mensaje no se encuentra en EF CBMID, el mensaje sólo se mostrará por pantalla.

6.1.4.4.2 CSD y GPRS

Como ya se ha comentado anteriormente, el control del flujo de datos en un diálogo CSD ó GPRS se lleva a cabo mediante los comandos OPEN CHANNEL, CLOSE CHANNEL, SEND DATA, RECEIVE DATA y GET CHANNEL STATUS; y los eventos Data available (notifica que hay datos disponibles) y Channel status (indica que se ha producido algún error en el canal). Con el comando OPEN CHANNEL se puede elegir el tipo de protocolo que se quiere utilizar: CSD ó GPRS.

En el caso utilizar CSD, hace falta que el control de llamadas (en particular las llamadas de datos) esté alojado y activado en el SST. Esto se debe a que CSD necesita que haya una llamada de datos. Esta llamada de datos se puede establecer cuando la

SIM (que proporciona los parámetros de la llamada) llame al comando OPEN CHANNEL (especificando la opción CSD) o cuando se llene el buffer de transmisión. A continuación se muestran ejemplos de establecimiento del canal CSD, transferencia de datos y cierre del canal:

Figura 42: OPEN CHANNEL→establecimiento inmediato de la llamada

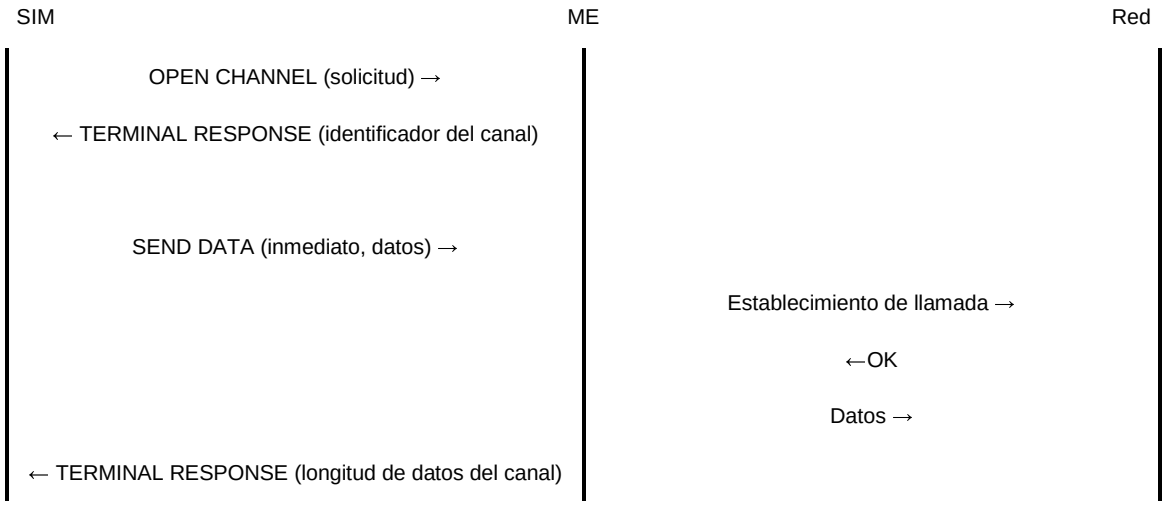
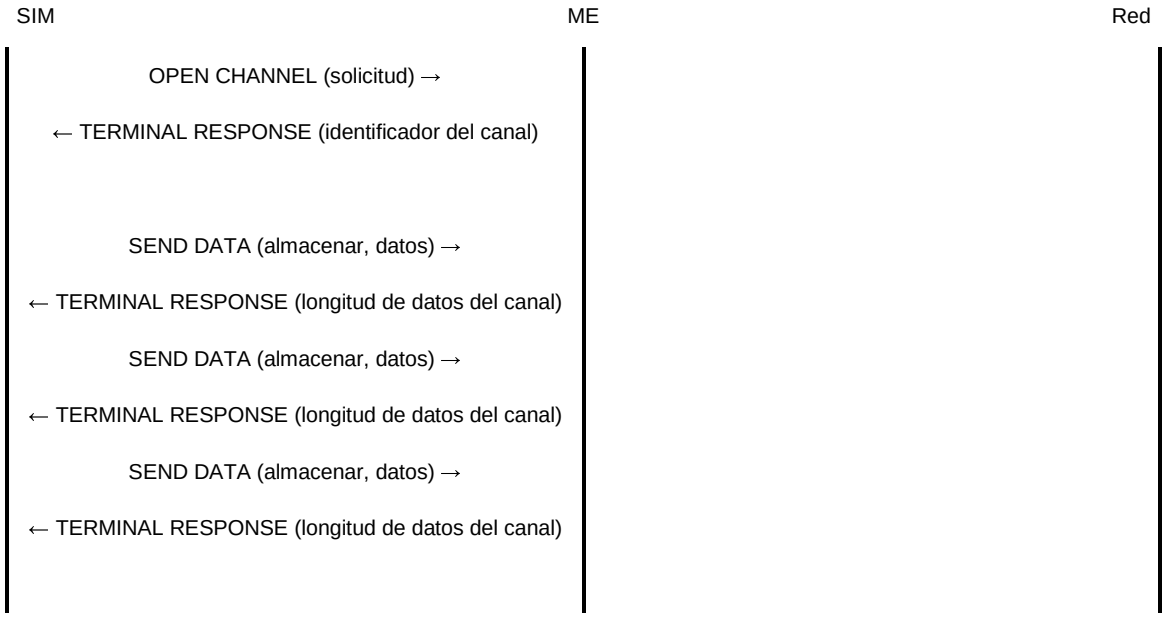


Figura 43: OPEN CHANNEL→establecimiento tardío/envío de datos inmediato



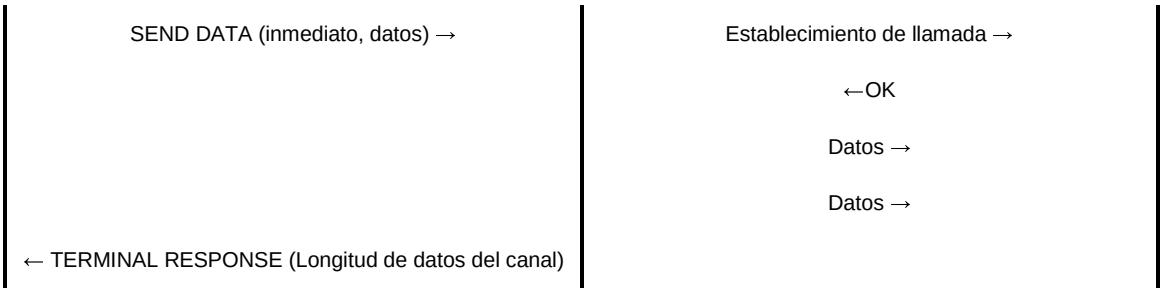


Figura 44: OPEN CHANNEL→establecimiento llamada con buffer Tx lleno

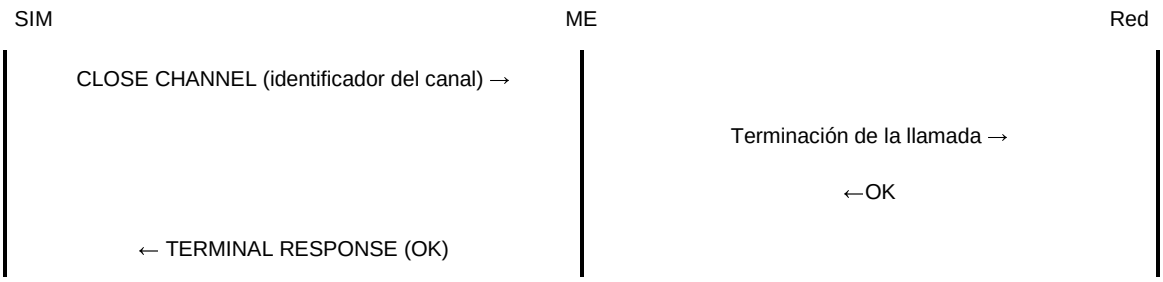


Figura 45: CLOSE CHANNEL

Hasta ahora se ha visto como establecer un canal y enviar datos a la red. La recepción de datos se basa en la captura del evento Data available. Este evento solo se dispara cuando el evento es parte de la lista actual de eventos (ajustada por el último comando SET UP EVENT LIST). Entonces, sólo si el buffer del canal elegido está vacío cuando los nuevos datos lleguen, el ME informará de esto a la SIM usando el comando ENVELOPE. Tras este aviso, la SIM recogerá los datos con el comando proactivo RECEIVE DATA. La siguiente figura completa lo expuesto:

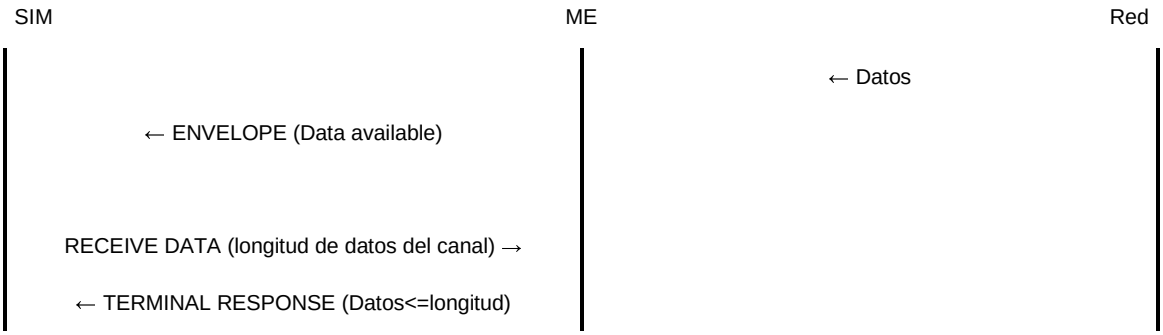


Figura 46: RECEIVE DATA

En el caso de GPRS, hace falta el establecimiento de un contexto PDP (Packet Data Protocol) con unos parámetros que la tarjeta SIM se encarga de proporcionar. El establecimiento de este contexto se puede realizar de forma inmediata o cuando haya suficientes datos para transmitir en el buffer de transmisión. El envío y recepción de datos se hace de forma similar al caso CSD. A continuación se muestran algunos diagramas de ejemplo:

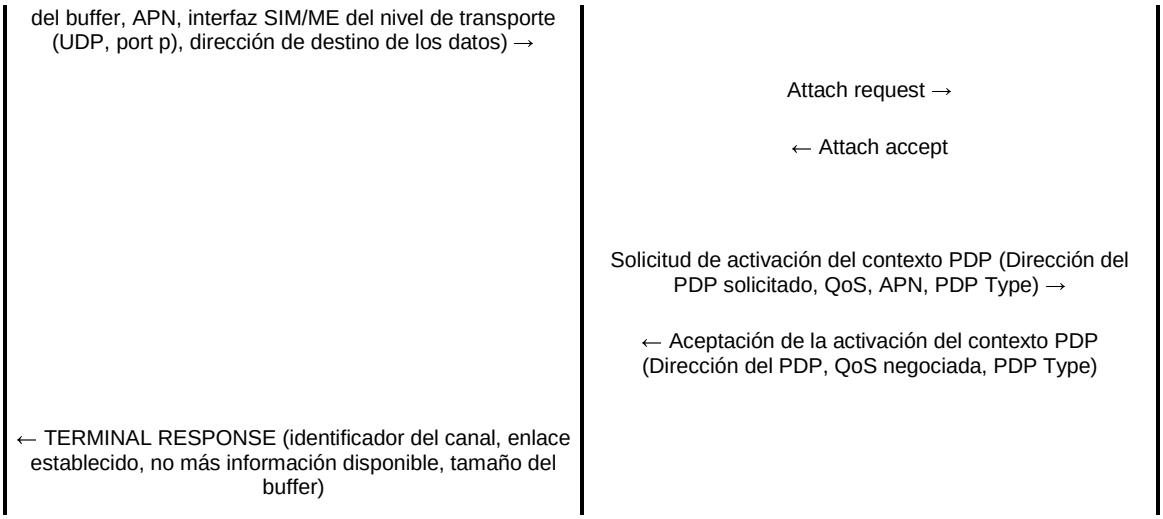


Figura 47: OPEN CHANNEL

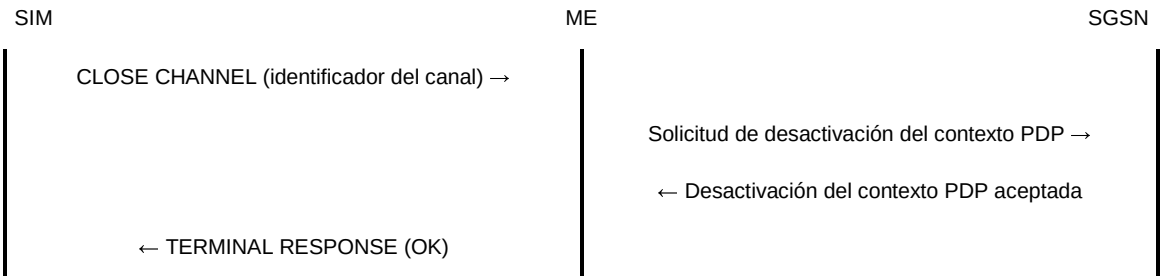


Figura 48: CLOSE CHANNEL

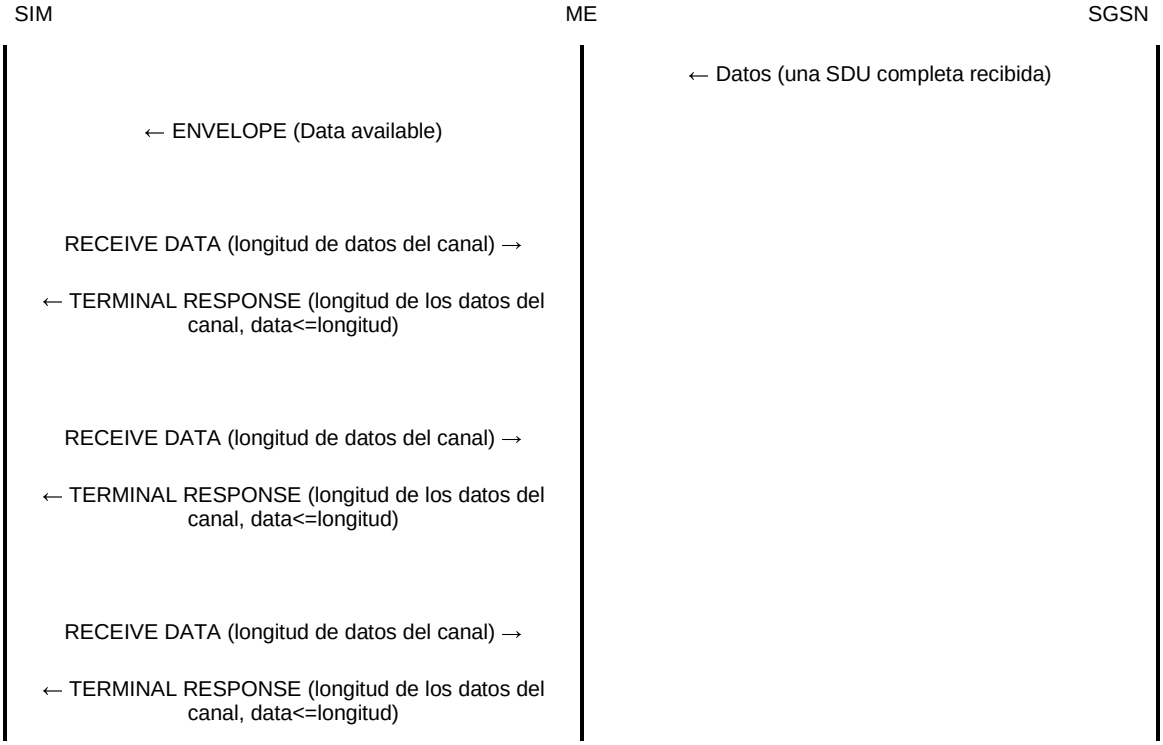


Tabla 14: RECEIVE DATA

Figura 49: SEND DATA usando el buffer de transmisión

6.1.5 CARACTERÍSTICAS DE LAS TARJETAS SIM Y USIM

En esta sección se resumen cuáles deben ser las características técnicas que debería tener una tarjeta para soportar el conjunto de funcionalidades requeridas.

6.1.5.1 Requisitos Hardware

La tecnología actual de tarjeta inteligente se basa en procesadores de 8 bits, con memorias EEPROM de 16 ó 32 kbytes, para almacenamiento de perfiles, datos y aplicaciones. Las cantidades habituales de ROM, donde generalmente se almacenan el sistema operativo y los estándares SIM Application Toolkit, funcionalidades OTA, y algoritmos de seguridad, no exceden de los 48 kbytes. La RAM, donde se gestiona la ejecución dinámica de las aplicaciones suele ser de 2 kbytes.

Las propuestas iniciales para tarjetas USIM, giran en torno a tarjetas de 64 kbytes de EEPROM, como paso inicial para migrar a la tecnología de 128 kbytes. La ROM es un elemento clave, ya que los sistemas operativos son cada vez más complejos y abiertos. Se habla de valores comprendidos entre 64 y 96 kbytes, con memorias RAM nunca inferiores a los 4 kbytes.

La aparición de servicios de comercio electrónico y soporte a las transacciones seguras WAP, hace necesario un soporte hardware para los algoritmos de firma digital. Todos los fabricantes de chips y proveedores de tarjetas tienen previsto incluir de serie criptoprocesadores y soporte software en todos sus desarrollos de 3G.

Respecto al protocolo de transporte, en un principio se soportará únicamente el T=0 (obligatorio en los terminales), aunque existen distintas alternativas, lideradas principalmente por los fabricantes de tarjetas. El Bearer Independent Protocol será quizá el principal impulsor de la incorporación de estándares de comunicación con el terminal más ágiles, así como del incremento de información útil al usuario en la tarjeta, cuyo acceso debe ser lo más inmediato posible.

6.1.5.2 Requisitos software

La tarjeta debe ser una plataforma lo más estándar posible, donde convivan las funcionalidades requeridas por las normas, y los desarrollos propios de cada operador. Dentro del escenario de una plataforma abierta, el terminal móvil debe cumplir el papel de elemento aglutinador de servicios y funcionalidades, donde pueden converger desarrollos en principio no pensados para las comunicaciones móviles.

Los fabricantes de tarjetas, que aportan valor añadido a un hardware tan desnudo, han optado desde hace tiempo por una arquitectura abierta, partiendo de sus desarrollos propietarios a nivel de sistema operativo. Para ocultar las diferencias entre los fabricantes (sistemas operativos) se ha optado por el modelo de máquina virtual en torno a la tecnología Java. Todas las aplicaciones se ejecutan sobre un sistema estándar y genérico, y es tarea del fabricante de tarjetas adaptar esa máquina virtual al sistema operativo y al hardware subyacente. Así el desarrollador no tiene que preocuparse por las peculiaridades del sistema operativo o del hardware del fabricante, a la hora de desarrollar una aplicación. Hace tiempo que el ETSI apostó por la tecnología Java, estandarizando el API para el desarrollo de aplicaciones, con este lenguaje, sobre la tarjeta SIM. El esquema de la estructura de la tarjeta SIM Java Card, se presenta en la siguiente figura:

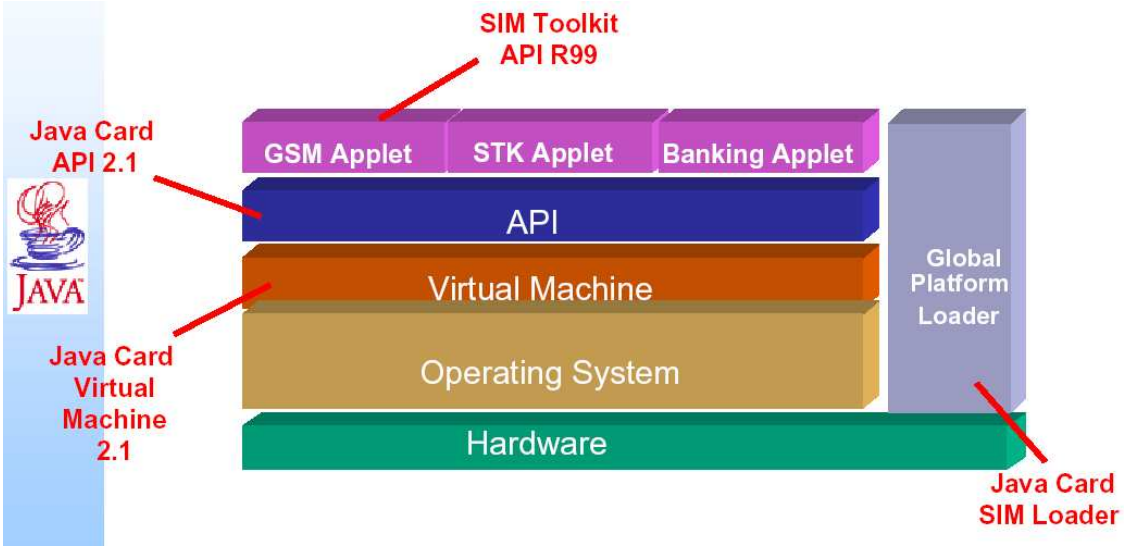


Figura 50: Esquema de la estructura de una tarjeta SIM Java Card

Hay que señalar que la acogida de la tecnología Java Card en el mercado de las tarjetas inteligentes en el sector de las telecomunicaciones, ha sido muy buena. El número de tarjetas Java Card en el mercado de las tarjetas inteligentes (TAM: Total Available Market) ha ido en aumento, en detrimento de otras tecnologías, como se puede ver en esta gráfica:

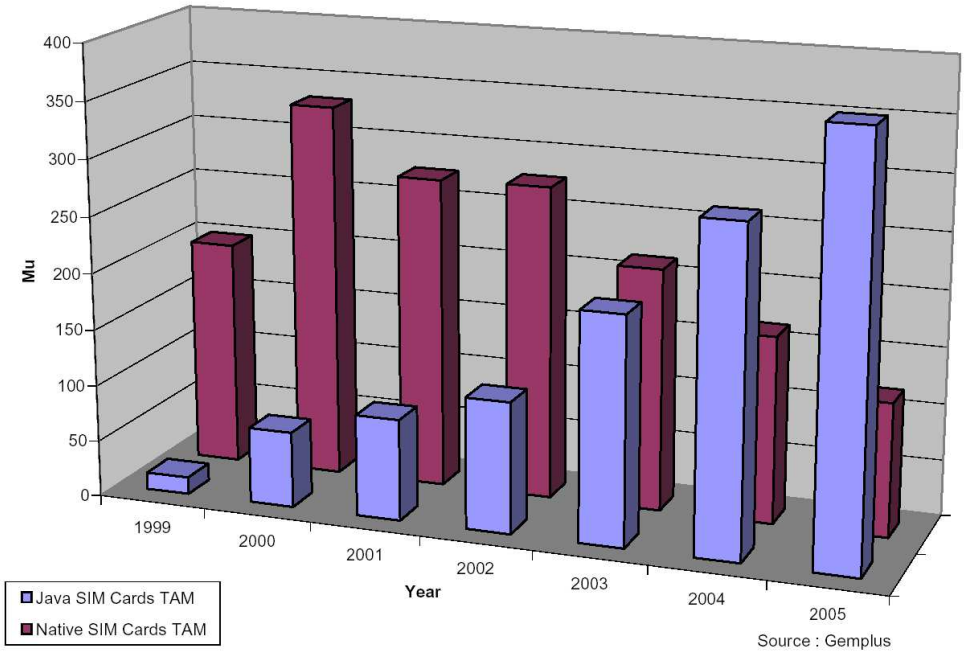


Figura 51: Java SIM Cards vs Native SIM Cards

En 2003 el volumen de tarjetas SIM con tecnología Java Card supone un 48% (un 65% de ellas son SIM's Java Card interoperables) del mercado total de tarjetas SIM. Para el año 2004, se espera que el 64% (del que el 90% será interoperable) del mercado total de tarjetas sean Java Card.

Esto se debe a los beneficios que encuentran las operadoras de telefonía móvil en el uso de la tecnología Java Card. Los beneficios más importantes son: la interoperabilidad, la proliferación de desarrolladores (con lo que estimula la aparición de nuevos servicios y en menos tiempo) y el no tener que desarrollar la aplicación más de una vez (Java Card segura la independencia de las características particulares de cada tarjeta). En la siguiente figura se muestran las características de Java Card que más interesan a las operadoras de telefonía móvil en porcentajes:

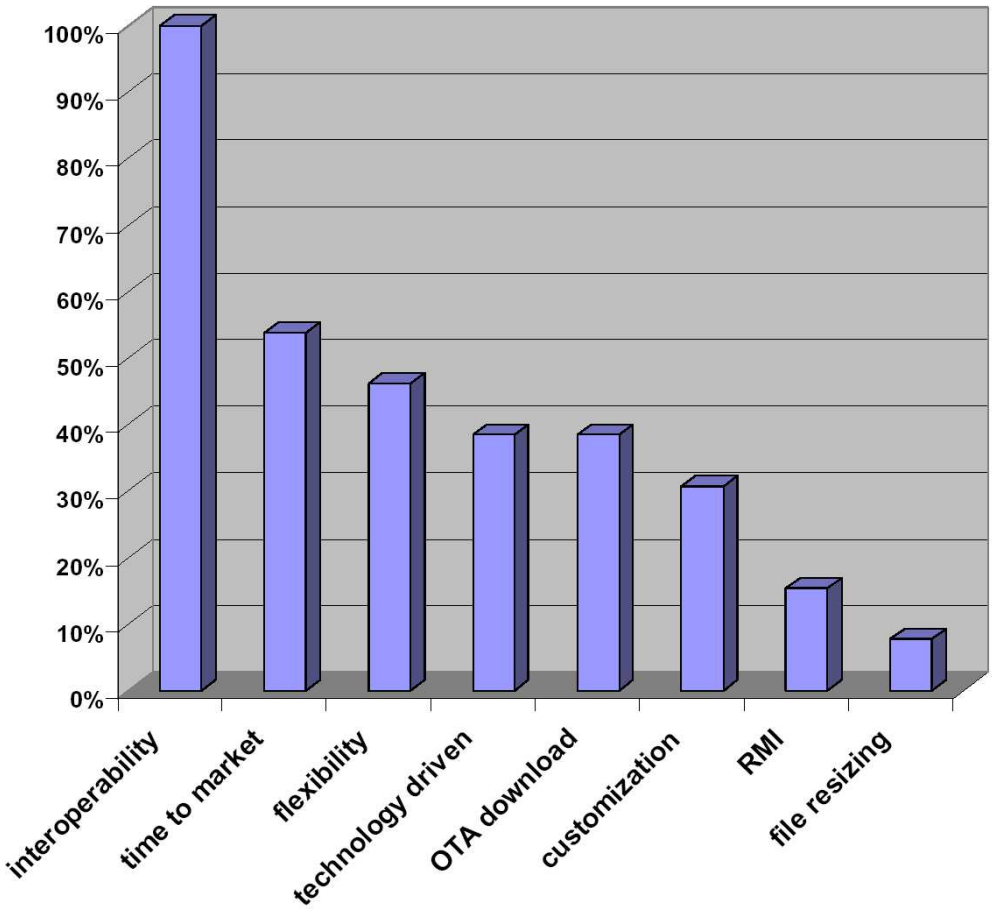


Figura 52: Interés de las operadoras por las características de Java Card

En el mercado han aparecido otras propuestas, dentro del contexto de tarjeta inteligente, lideradas por Microsoft y Maosco Ltd (Smart Card for Windows y MultOs) que son las principales alternativas a Java. Es importante destacar que estas alternativas no están diseñadas explícitamente para su explotación dentro del mundo de las comunicaciones móviles, aunque se está estudiando el uso de estas tarjetas como módulos SIM.

A nivel de usuario estas tarjetas son compatibles, con el mismo comportamiento y funcionalidades. Sin embargo, a nivel de desarrollo de aplicaciones, estas soluciones no son compatibles entre sí. Ambas alternativas ofrecen un estándar completo: sistema operativo y entorno de ejecución. Java únicamente estandariza el entorno de ejecución.

Sea cual sea la solución adoptada, se debe tener presente que la arquitectura debe estar preparada para crecer tanto en recursos disponibles como en funcionalidades requeridas y estándares incorporados.

6.1.6 COMPATIBILIDAD GSM/UMTS FASE 1 RELEASE 99

A continuación se resumen los criterios de compatibilidad entre las tarjetas USIM con las tarjetas SIM actuales. La tarjeta USIM no supone una revolución respecto de la tarjeta SIM tradicional. Debido a esto, el 3GPP recomienda a los fabricantes de tarjetas y operadoras, diseñar las primeras versiones de la tarjeta UMTS sobre la base de

una tarjeta SIM fase 2+ (funcionalidad SIM Application Toolkit Release 99). Los requisitos que se proponen para esta tarjeta básica son los siguientes:

- La tarjeta USIM debe implementar los mecanismos avanzados de seguridad definidos para UMTS.
- Un ME UMTS debe soportar tarjetas SIM tanto fase 2+, como módulos de acceso a una red UMTS. Se aconseja que no ofrezca soporte a la tecnología de 5V.
- El operador debe decidir si la seguridad que proporciona una tarjeta SIM es suficiente para aceptarla como usuario de la red UMTS. En todo caso, los servicios UMTS proporcionados a un cliente que solamente tenga una tarjeta SIM en su terminal 3G se deberían limitar a los servicios que obtendría de una red GSM.
- Es posible disponer de varias aplicaciones sobre una misma UICC, pero en una primera fase solamente se requerirán las funcionalidades SIM y USIM, pudiendo compartir los ficheros correspondientes si es necesario.
- El operador debería disponer de un mecanismo seguro y estandarizado para la transferencia de aplicaciones y datos hacia la UICC.

6.2 LA SIM API

6.2.1 INTRODUCCIÓN

En esta sección se describirá la SIM API. La especificación en la que se define la SIM API es la GSM 2.19. En las especificaciones GSM 3.19 (Release 1999) y TS 43.019 (Release 4) se encuentran la SIM API para la tecnología Java Card en su versión 2.1. Las especificaciones de la UICC API para UMTS se describen en la norma TS 31.111 y en la TS 102.241 se encuentra la UICC API para la versión 2.2 de Java Card. De aquí en adelante solo se hará referencia a las especificaciones de las normas GSM para el SIM Application Tololkit, ya que las de UMTS son muy parecidas a las de GSM.

El alcance de esta sección comprende temas como la arquitectura GSM Java Card en una tarjeta SIM, la descarga de los applets haciendo uso de los recursos OTA y la instalación de los applets en la SIM.

6.2.2 ARQUITECTURA GSM JAVA CARD

La arquitectura de la SIM API basada en Java Card 2.1 se presenta en la siguiente figura:

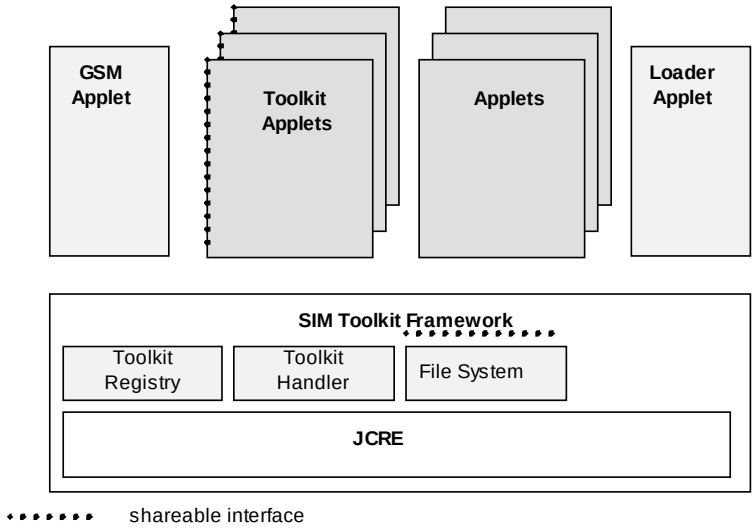


Figura 53: Arquitectura de GSM Java Card

Es este esquema se distinguen una serie de elementos que se explican a continuación:

- SIM Toolkit Framework: este es el entorno de ejecución de GSM Java Card y está compuesto por el JCRE, el Toolkit Registry, el Toolkit Handler and el File System.
- JCRE: es como el sistema operativo de la tarjeta SIM y es capaz de seleccionar cualquier applet.
- Toolkit Registry: este componente maneja toda la información de registro de los applets toolkit y sus vínculos con el registro del JCRE.
- Toolkit Handler: este componente controla la disponibilidad del manejador del sistema y el protoco toolkit (por ejemplo la suspensión de un applet toolkit).
- File System: este componente contiene el sistema de ficheros del proveedor de tarjetas y maneja el control de acceso de los ficheros y el contexto del applet de fichero. Es un objeto que pertenece al JCRE que implementa la interfaz compartida `sim.access.SIMView`.
- Applets: como ya se sabe los applets derivan de la clase `javacard.framework.applet` e implementan los métodos: `process`, `select`, `deselect e install` tal y como se define en la Java Card 2.1 Runtime Environment Specification.
- Applets Toolkit: estos applets extienden la clase `javacard.framework.applet` e implementan la interfaz compartida `sim.toolkit.ToolkitInterface` para que estos applets se puedan disparar mediante una invocación a su método `processToolkit`. Los AID's de estos applets se definen en la norma TS 101 220.

- Applet GSM: este es el applet por defecto, se comporta como un applet normal en la tecnología Java Card (como se vio en secciones anteriores). Su AID se define en la norma TS 101 220. Este applet maneja las APDU's que se envían a la SIM, CHV1/2 (Card Holder Verification information: condición de acceso usada por la SIM para la verificación de la identidad del usuario), el algoritmo de autenticación GSM y el control de acceso a los ficheros de abonado (todas estas cuestiones se especifican en la norma GSM 11.11).
- Applet Loader: este applet se encarga de los aspectos de instalación y desinstalación de los applets como se especifica en la norma GSM 03.48.
- Shareable interface: que se definen en las especificaciones Java Card 2.1.

6.2.3 GSM FRAMEWORK

El GSM Framework (soporte GSM) consiste en el applet GSM y el objeto de sistema de ficheros del JCRE. El GSM Framework se compone de dos paquetes:

- El paquete GSM de bajo nivel.
- El paquete `sim.access`, que permite que los applets puedan acceder a los ficheros GSM.

6.2.3.1 Acceso a los ficheros de datos GSM

La API ofrece los siguientes métodos a los applets de la tarjeta, para que puedan acceder a los datos GSM:

- `select`: selecciona un fichero sin cambiar el fichero actual o de cualquier otro applet o de la sesión de abonado. En la invocación del método `processToolkit` de un applet toolkit, el fichero actualmente seleccionado es el MF. El contexto del fichero del applet toolkit permanece sin cambios durante toda la ejecución del método `processToolkit`. El registro actual quizás se pueda alterar si el fichero actual es un fichero cíclico. Este método devuelve información acerca del fichero seleccionado.
- `status`: lee la información de estado del fichero del DF actual.
- `readBinary`: lee los bytes de datos del EF transparente actualmente seleccionado por el applet.
- `readRecord`: lee los bytes de datos del EF lineal fijo o cíclico actualmente seleccionado por el applet, sin cambiar el puntero de registro actual de cualquier otro applet del abonado. Este método permite leer una parte de un registro.
- `updateBinary`: modifica los bytes de datos del EF transparente actualmente seleccionado por el applet.
- `updateRecord`: modifica los bytes de datos del EF lineal fijo o cíclico actualmente seleccionado por el applet. El puntero de registro actual de otros applets o del abonado, no se cambiará en el caso del EF lineal fijo.

Sin embargo, el puntero de registro de un EF cíclico se cambiará para todos applets o para el abonado, al registro número 1. Este método permite actualizar parte de un registro.

- `seek`: busca un registro del fichero lineal fijo actualmente seleccionado por el applet, empezando por un patrón dado. El puntero de registro de cualquier otro applet o de la sesión de abonado, no será modificado.
- `increase`: incrementa el valor del último registro actualizado del EF cíclico actualmente seleccionado y guarda el resultado en el registro más antiguo. El puntero de registro se ajusta a este registro y éste se convierte en el registro número 1. Este método devuelve el valor incrementado.
- `rehabilitate`: rehabilita el EF actualmente seleccionado por el applet, con efecto en todos los applets y el abonado.
- `invalidate`: invalida el EF actualmente seleccionado por el applet, con efecto en todos los applets y el abonado.

6.2.3.2 Control de acceso

Los privilegios del control de acceso de un applet se conceden durante la instalación según el nivel de confianza. Cuando un applet solicita el acceso a los ficheros GSM o del operador, el SIM Toolkit Framework comprueba si permite el acceso, examinando la información del fichero de control almacenado en la tarjeta. Si concede el acceso, el SIM Toolkit Framework procesa el acceso solicitado. Si no concede el acceso, lanza una excepción. Los contenidos y codificación de los ficheros que contienen información de control de acceso se definen en la norma GSM 11.11.

6.2.4 SIM TOOLKIT FRAMEWORK

La SIM API se compone de API's para la norma GSM 11.11 (ICC y Aplicación SIM) y para la norma GSM 11.14 (SIM Application Toolkit). La relación entre los applets (que pueden no ser toolkit), el sistema de ficheros y las APDU's que vienen del ME se muestra en la Figura 54: Arquitectura del SIM Toolkit Framework

El trasiego de APDU's que se produce entre la SIM y el ME, lo controla el GSM Framework (que es un applet del JCRE) que además también controla el acceso al sistema de ficheros de la tarjeta. El SIM Toolkit Framework, se encarga de instalar, desinstalar, seleccionar (ejecutar) y deseleccionar los applets toolkit, y provee acceso a los ficheros.

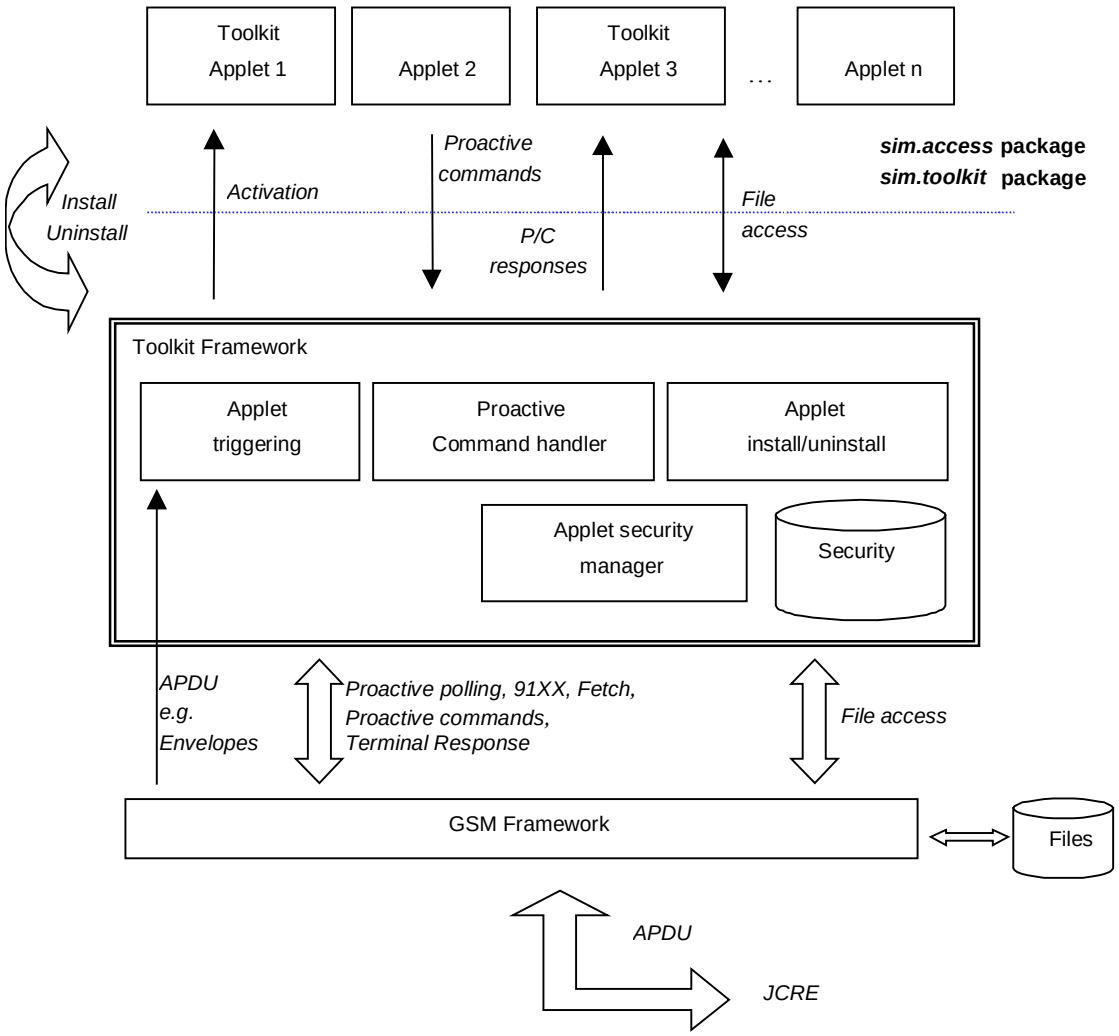


Figura 54: Arquitectura del SIM Toolkit Framework

En este modelo, la estructura de campos de datos de GSM se ve como una serie de objetos de datos a través de la API. Por supuesto, en el modelo físico, los datos quizás estén guardados en ficheros elementales, pero las clases acceden a estos datos como si fueran parte de los objetos de aquellas clases.

6.2.4.1 Disparo de un applet

La aplicación de disparo (Applet triggering), porción del SIM Toolkit Framework, es responsable de la activación de los applets toolkit, que se debe a las ADPU's recibidas por la aplicación GSM.

El ME no será perjudicado por la presencia de applets en la tarjeta SIM. Por ejemplo un comando ENVELOPE sintácticamente correcto no dará una palabra de estado (SW) indicando un error, en el caso de que se produzca un fallo en un applet toolkit. La única aplicación que ve el ME es la aplicación SIM. Como resultado, el applet toolkit quizás lance una excepción, pero este error no se enviará al ME.

La diferencia entre un applet Java Card y un applet toolkit, es que el último no maneja las APDU's directamente. Tratará con mensajes de más alto nivel. Además la

ejecución de un método podría abarcar múltiples APDU's, en particular, el protocolo de comandos proactivos (FETCH, TERMINAL RESPONSE).

Como se ha visto arriba, cuando el applet GSM es la aplicación seleccionada y cuando un applet toolkit se dispara, el método `select()` del applet toolkit no se lanzará hasta que el applet toolkit no sea realmente seleccionado por si mismo mediante una APDU SELECT.

Aquí se muestran los eventos que pueden disparar un applet toolkit:

Tabla 15: Eventos que disparan un applet toolkit

Eventos
EVENT_FORMATTED_SMS_PP_ENV
EVENT_FORMATTED_SMS_PP_UPD
EVENT_UNFORMATTED_SMS_PP_ENV
EVENT_UNFORMATTED_SMS_PP_UPD
EVENT_FORMATTED_SMS_CB
EVENT_UNFORMATTED_SMS_CB
EVENT_MENU_SELECTION
EVENT_MENU_SELECTION_HELP_REQUEST
EVENT_CALL_CONTROL
EVENT_SMS_MO_CONTROL
EVENT_TIMER_EXPIRATION
EVENT_EVENT_DOWNLOAD
EVENT_MT_CALL
EVENT_CALL_CONNECTED
EVENT_CALL_DISCONNECTED
EVENT_LOCATION_STATUS
EVENT_USER_ACTIVITY
EVENT_IDLE_SCREEN_AVAILABLE
EVENT_LANGUAGE_SELECTION
EVENT_BROWSER_TERMINATION
EVENT_CARD_READER_STATUS
EVENT_UNRECOGNISED_ENVELOPE
EVENT_STATUS_COMMAND
EVENT_PROFILE_DOWNLOAD

Hay que añadir que un rango de eventos está reservado para uso propietario (de –128 a –1). El uso de estos eventos puede hacer que el applet toolkit sea incompatible.

El applet toolkit será disparado por la recepción de eventos registrados y será capaz de acceder a los datos asociados al evento, usando los métodos provistos por la clase `sim.toolkit.ViewHandler.EnvelopeHandler`.

El orden de disparo de los applets toolkit se establecerá mediante el nivel de prioridad de cada applet toolkit y que se define en el proceso de descarga. Si varios applets toolkit tienen el mismo nivel de prioridad, el último applet toolkit descargado tiene preferencia.

6.2.4.2 Registro de un applet toolkit

Durante la instalación del applet toolkit, el applet se registra en el JCRE y en el SIM Toolkit Framework para que se pueda seleccionar mediante ambos mecanismos de selección (mediante un evento y mediante el lanzamiento de su método `select()`).

El applet toolkit tiene que llamar al método `getEntry()` para conseguir una referencia de su registro y entonces registrar explícitamente cada uno de los eventos requeridos. El applet toolkit puede cambiar los eventos en los que está registrado durante su tiempo de vida.

El applet toolkit se podrá registrar dinámicamente en algún evento, por ejemplo EVENT_MENU_SELECTION, llamando al método corrspondiente, por ejemplo `initMenuEntry()`.

6.2.4.3 Tratamiento de los comandos proactivos

El protocolo SIM Application Toolkit (basado en: la palabra de estado 91xx, FETCH, TERMINAL RESPONSE) es manejado por el applet GSM y el Toolkit Handler, el applet toolkit applet no maneja aquellos eventos.

El SIM Toolkit Framework provee una referencia del `sim.toolkit.ViewHandler.EditHandler.ProactiveHandler` al applet toolkit para que cuando se dispare al applet toolkit, éste pueda:

- inicializar el comando proactivo actual con el método `init()`.
- enviar varios TLV (Tag, Length, Value) simples como se define en GSM 11.14 al comando proactivo actual con los métodos `appendTLV()`.
- perdir al SIM Toolkit Framework que envíe este comando proactivo al ME y que espere la respuesta, mediante el método `send()`.

El applet GSM y el SIM Toolkit Framework controlarán la transmisión del comando proactivo al ME, y la recepción de la respuesta. Entonces, el SIM Toolkit Framework volverá al applet toolkit justo después del método `send()`. A continuación, al applet toolkit se le proveerá un `sim.toolkit.ViewHandler. ProactiveResponseHandler`, para que el applet toolkit pueda analizar la respuesta.

El comando proactivo se envía al ME tal y como está definido, y el applet toolkit los construye sin ninguna comprobación por parte del SIM Toolkit Framework.

El applet toolkit no proporcionará los siguientes comandos proactivos: SET UP MENU, SET UP EVENT LIST, POLL INTERVAL, POLLING OFF. Esto es debido a que son comandos proactivos del sistema que afectan a los servicios del SIM Toolkit Framework.

En el caso en que un applet toolkit utilice el comando proactivo SET UP IDLE MODE TEXT para imprimir un texto en la pantalla, el SIM Toolkit Framework no garantiza que otro applet no sobrescriba este texto en una etapa posterior.

6.2.4.4 Tratamiento de las respuestas

Para permitir que un applet toolkit responda a algunos eventos específicos (por ejemplo EVENT_CALL_CONTROL_BY_SIM), el SIM Toolkit Framework provee el `sim.toolkit.ViewHandler.EditHandler.EnvelopeResponseHandler`.

Entonces, el applet toolkit puede anunciar una respuesta a algunos eventos con los métodos `post()` o `postAsBERTLV()`. Así el applet toolkit puede continuar su proceso (por ejemplo preparar un comando proactivo). El SIM Toolkit Framework devolverá la APDU de respuesta definida por el applet toolkit (por ejemplo 9FXX ó 9EXX).

6.2.4.5 DisponiBilidad de los controladores

Los controladores del sistema: `ProactiveHandler`, `ProactiveResponseHandler`, `EnvelopeHandler` y `EnvelopeResponseHandler` son objetos temporales de punto de entrada al JCRE (ya vistos en la sección 4.8.3 Los objetos de punto de entrada al JCRE que se encuentra en la página 99).

Las siguientes reglas definen los requisitos mínimos para la disponibilidad de controladores del sistema y el tiempo de vida de sus contenidos.

6.2.4.5.1 Controlador `ProactiveHandler`

- El `ProactiveHandler` es válido desde la invocación hasta el fin del método `processToolkit`.
- Si un comando proactivo está pendiente, quizás el `ProactiveHandler` no esté disponible.
- En la invocación del método `processToolkit`, la lista TLV se borra.
- En la llamada de su método `init`, se borra su contenido y se inicializa.
- Después de una llamada al método `ProactiveHandler.send`, el controlador permanecerá sin cambios (es decir, después del envío de un comando proactivo) hasta que se llame al método `ProactiveHandler.init` ó `appendTLV`.

6.2.4.5.2 Controlador `ProactiveResponseHandler`

- El `ProactiveResponseHandler` quizás no esté disponible antes de la primera llamada al método `ProactiveHandler.send`.
- El `ProactiveResponseHandler` está disponible después de la primera llamada al método `ProactiveHandler.send` hasta la terminación del método `processToolkit`.
- Si un comando proactivo está pendiente, quizás el `ProactiveResponseHandler` no esté disponible.
- El contenido del `ProactiveResponseHandler` se cambia después de la llamada al método `ProactiveHandler.send` y permanece sin cambios hasta la próxima llamada al método`ProactiveHandler.send`.

6.2.4.5.3 Controlador `EnvelopeHandler`

- El `EnvelopeHandler` y su contenido están disponibles para todos los applets toolkit disparados, desde la invocación hasta la terminación de sus métodos `processToolkit`.
- El SIM Toolkit Framework garantiza que todos los applets toolkit registrados se pueden disparar y que reciben los datos.

6.2.4.5.4 Controlador EnvelopeResponseHandler:

- El `EnvelopeResponseHandler` está disponible para todos los applets toolkit, hasta que un applet toolkit anuncie una respuesta o envíe un comando proactivo. Después de una llamada al método `post`, el método no estará más tiempo disponible.
- El contenido de `EnvelopeResponseHandler` se debe anunciar antes de la primera invocación al método `ProactiveHandler.send` o antes de la terminación del método `processToolkit`, para que el applet GSM pueda ofrecer esos datos al ME (por ejemplo 9FXX/9EXX). Después de la primera invocación al método `ProactiveHandler.send`, el `EnvelopeResponseHandler` no estará disponible.

6.2.4.6 Comportamiento del SIM Toolkit Framework

Las siguientes reglas definen el comportamiento del SIM Toolkit Framework:

- Disparo de un applet toolkit, es decir, la invocación del método `processToolkit()` de la interfaz compartida `ToolkitInterface`:
  - El contexto actual se cambia al del applet toolkit.
  - Una transacción pendiente se interrumpe.
  - No se invoca a los métodos `select()` ó `deselect()`.
  - No se puede acceder y no se puede crear un objeto `CLEAR_ON_DESELECT` como se define en las especificaciones de la tecnología Java Card, ya que la aplicación actualmente seleccionada no se cambia (por ejemplo el applet GSM) y no corresponde con el contexto actual que es el del applet toolkit.
  - El contexto de fichero actual del applet toolkit es el MF.
  - El contexto de fichero actual del applet actualmente seleccionado no se altera.
  - El applet toolkit no puede acceder al objeto `APDU`.
- Terminación de un applet toolkit, es decir, la vuelta del método `processToolkit()`:
  - El JCRE cambia de nuevo al contexto del applet actualmente seleccionado, el applet GSM.

  - No hay invocaciones de los métodos `select()` ó `deselect()`.
  - Una transacción pendiente de una applet toolkit se interrumpe.
  - Los datos transitorios permanecen sin alteraciones.
  - El contexto de fichero actual del applet toolkit se pierde.
  - El contexto de fichero del applet actualmente seleccionado no se altera.
  - El applet GSM no confía en el contenido del objeto `APDU`. El contenido de la APDU quizás deba ser cambiado por el sistema.

- Invocación del método `ProactiveHandler.send()`:
  - Durante la ejecución puede haber cambios de contexto, pero a la vuelva del método `send()` se restaura el contexto del applet toolkit.
  - No hay invocación de los métodos `select()` o `deselect()`.
  - Una transacción pendiente de un applet toolkit se interrumpe en el instante en que se invoca al método.
  - El contexto de fichero actual del applet toolkit permanece sin cambios. Nunca se volverá del método `send()` si el applet GSM se deselecciona y otro applet se selecciona explícitamente.
- Emisión de comandos proactivos del sistema (comportamiento dinámico del SIM Toolkit Framework)
  - El SIM Toolkit Framework enviará sus comandos proactivos de sistema tan pronto como no haya pendiente ninguna sesión proactiva y todos los applets registrados a los eventos actuales hayan sido disparados y hayan vuelto de la invocación del método `processToolkit`.

6.2.5 GESTIÓN DE LA VIDA DEL APPLET TOOLKIT

La gestión de la vida del applet toolkit comprende la preparación del applet, la descarga, la instalación, el registro, la configuración, la ejecución y el borrado/desactivación. La Figura 55 muestra de forma esquemática cada una de las etapas que componen el ciclo de vida de un applet toolkit.

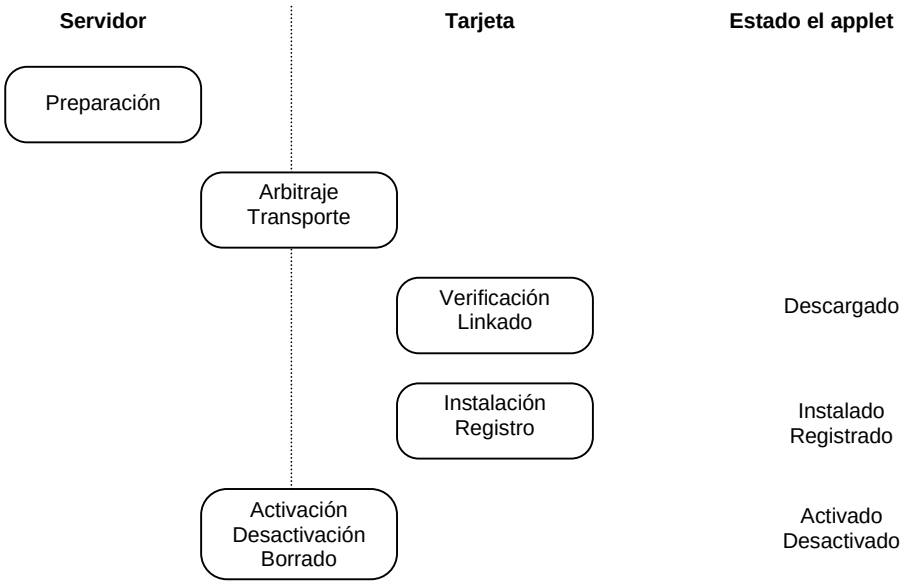


Figura 55: Ciclo de vida del applet toolkit

6.2.5.1 Preparación del applet toolkit

La preparación del applet se refiere a la fase opcional de verificar que el código del applet cumple con los estándares de proveedores de tarjetas.

El applet se identificará mediante un AID que se asigna siguiendo el procedimiento descrito en la norma ISO 7816-5 y un AVN (Applet Version Number). Tanto el AID como el AVN se asignan durante la fase de preparación del applet.

Además, un applet toolkit puede tener asignado un TAR (Toolkit Application Reference) que se usa para identificar a una aplicación de segundo nivel. El TAR del applet se registra durante la descarga del mismo. El TAR (compuesto por 3 bytes) del applet se toma a partir de los bytes 12, 13 y 14 del AID del applet.

Los requisitos mínimos del applet estarán especificados, tales como la versión del API, la capacidad de la SIM, requisitos de recursos disponibles, etc.

6.2.5.2 Descarga

La descarga se refiere al proceso de transportar el código del applet desde un servidor de descarga hasta la SIM y a la generación del código de descarga en la SIM.

El proceso funciona bajo un principio de control impuesto por el proveedor de la tarjeta. El proveedor quizás elija delegar esta responsabilidad a una o más partes de confianza. También debe tener en cuenta restricciones de recursos (por ejemplo la cantidad máxima de memoria disponible) o de acceso (por ejemplo limitar o reducir la funcionalidad).

El proceso de descarga implica cuatro fases bien diferenciadas: el arbitraje, el transporte, la verificación y el linkado. Si el servidor de descarga lo solicita, la tarjeta puede suministrarle un informe acerca del éxito o el fallo (incluyendo un código de identificación de errores) de la descarga.

6.2.5.2.1 Arbitraje

Esta fase se efectúa mediante la autenticación mutua que se da entre la SIM y el servidor de descarga, y el establecimiento de las claves de sesión apropiadas para asegurar la seguridad durante la transferencia de datos.

Los requisitos mínimos del applet se verifican con respecto al entorno presente en la SIM (por ejemplo la versión de la API, la capacidad de la SIM y la memoria disponible). Si esto fallase, se abortaría el proceso de descarga.

Los identificadores de los applets (AID's) y los números de versión (AVN's) de cualquier applet ya instalado en la SIM, se comparan con el AID y el AVN del applet que se va a descargar. Si un applet idéntico ya está instalado en la SIM (el AID y el AVN coinciden), se salta las fases de transporte, verificación y linkado. Si en la tarjeta hay disponible un applet con el mismo AID pero con un AVN diferente, se borra.

6.2.5.2.2 Transporte

Esta etapa efectúa el transporte de los paquetes de datos desde el servidor de descarga hasta la SIM. Quizás se haga de acuerdo con la norma GSM 03.48 (Mecanismos de Seguridad para el SIM Application Toolkit) y opcionalmente con el uso de codificación con claves generadas e intercambiadas durante la fase de arbitraje.

6.2.5.2.3 Verificación

Esta etapa efectúa la verificación de los datos recibidos. Si la etapa de verificación falla, el applet será descartado.

6.2.5.2.4 Linkado

Esta etapa lleva a cabo el linkado del código recibido con el entorno de ejecución presente en la tarjeta.

6.2.5.3 Instalación/Registro/Reactivación

Esta etapa se refiere a la ejecución del código de applet además de la instalación y registro del applet en el entorno de ejecución SIM/ME.

Si el applet ya existía en la SIM y es desactivado, la solicitud de instalación reactivará el applet. Hay otros métodos de reactivación posibles a través de comandos.

6.2.5.4 Configuración

Esta etapa supone cualquier proceso de configuración necesario del código del applet para un usuario/ajuste/entorno particular.

6.2.5.5 Ejecución

En esta etapa, siempre que el applet esté activado, el applet se encuentra en un estado donde su ejecución se puede disparar por la presencia de cualquier evento al que se haya registrado.

6.2.5.6 Desactivación

Esta etapa supone el anular la capacidad de poder ejecutar el código del applet en la SIM. Este proceso puede ser disparado por el usuario, la operadora de la red o cualquier tercera parte.

6.2.5.7 Borrado

Esta etapa sigue a la de desactivación del applet. Previene la reactivación del applet. También puede liberar la memoria ocupada por el applet. Por razones de seguridad, la zona de memoria ocupada por el applet se puede rellenar con `null`.

6.2.6 GESTIÓN REMOTA DE APPLETS

Para las tarjetas basadas en la norma TS 03.19, es decir, para las tarjetas que utilizan la tecnología Java Card, se definen en la norma TS 03.48 un conjunto de comandos usados en la gestión remota de applets (como el applet GSM o los applets toolkit). Estos comandos se basan en la especificación Open Platform 2.0.1 (se puede encontrar en `http://www.globalplatform.org/`) de gestión de applets. La diferencia entre 03.48 y Open Platform, es que Open Platform utiliza procedimientos de autenticación que ofrecen más seguridad que 03.48. Para las tarjetas SIM que utilizan otras tecnologías quizás se usen otros mecanismos de descarga.

Cuando se habla de gestión, se hace referencia a la carga, instalación y borrado de applets. Por lo tanto, la gestión remota de applets supone para las operadoras una herramienta muy poderosa para poder gestionar fácilmente las aplicaciones SIM Application Toolkit residentes en las tarjetas SIM de las operadoras. Así se pueden implantar servicios novedosos en poco tiempo y conseguir diferenciarse de la competencia para captar nuevos clientes atraídos por dichos servicios.

6.2.6.1 Comportamiento de la aplicación de gestión remota

En esta sección se describen los pasos y comandos necesarios para instalar un applet en la SIM.

6.2.6.1.1 Descarga de un paquete

Las descargas de paquetes permite a la operadora (propietaria de la red) o el proveedor de servicios, descargar nuevos paquetes en la tarjeta SIM. La operadora o el proveedor de servicios gestionan la descarga de un paquete a través de una sesión de descarga con la tarjeta. Una sesión de descarga consiste en una secuencia de comandos, tal y como se describe en siguiente figura:

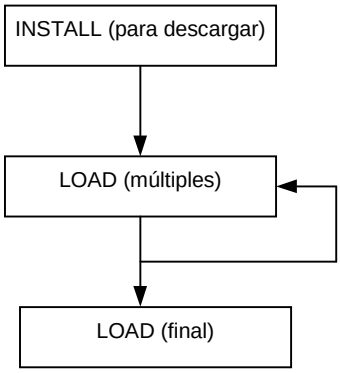


Figura 56: Secuencia de comandos de una sesión de descarga

La cantidad de SMS's utilizados para descargar el paquete, depende del tamaño del mismo. En estos SMS's se encuentran incrustados las APDU's, que contienen pequeños bloques de datos del fichero CAP que representa el paquete a descargar.

El tiempo de descarga de un applet toolkit a la tarjeta mediante el envío de mensajes cortos puede llegar a ser alto, como se puede comprobar en la siguiente figura. También se puede ver como con el uso de SMS vía GPRS (línea roja), el tiempo de descarga se reduce considerablemente.

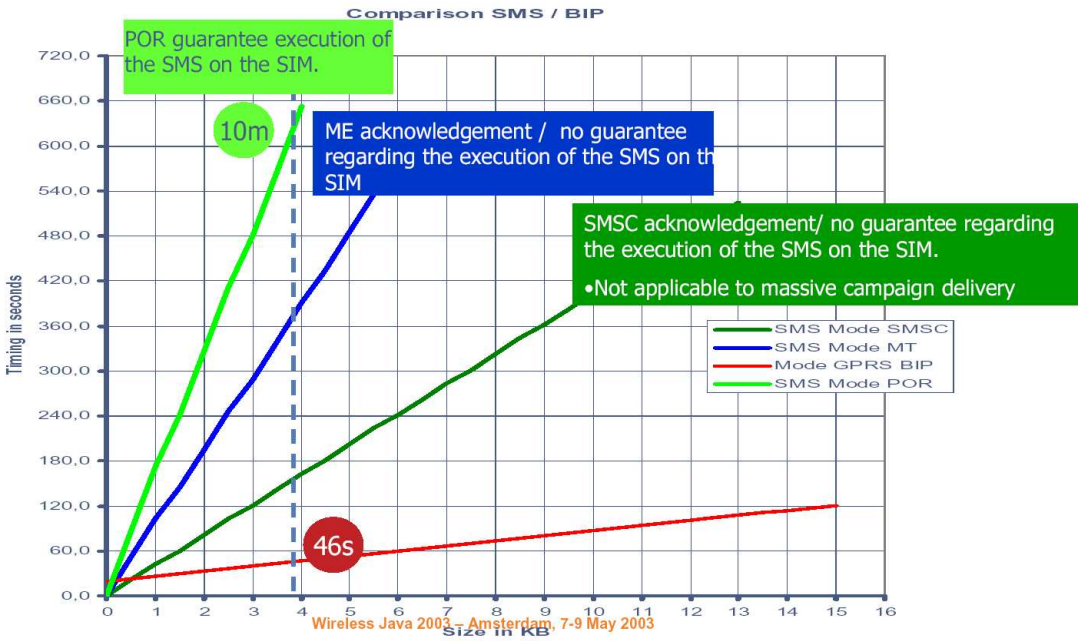


Figura 57: SMS vs SMS vía GPRS

En esta figura, las siglas POR significan acuse de recibo (Proof of Receipt) y debido a ello se ralentiza más el proceso de descarga (línea verde claro).

6.2.6.1.2 Instalación de applets

El proceso de instalación del applet permite a la operadora o al proveedor de servicios instalar una nueva aplicación en la tarjeta SIM. La instalación solo se llevará a cabo si el paquete correspondiente ya ha sido descargado en la tarjeta. La instalación del

applet se lleva a cabo usando el comando INSTALL. El proceso de descarga del paquete y la instalación del applet se resume en esta figura:

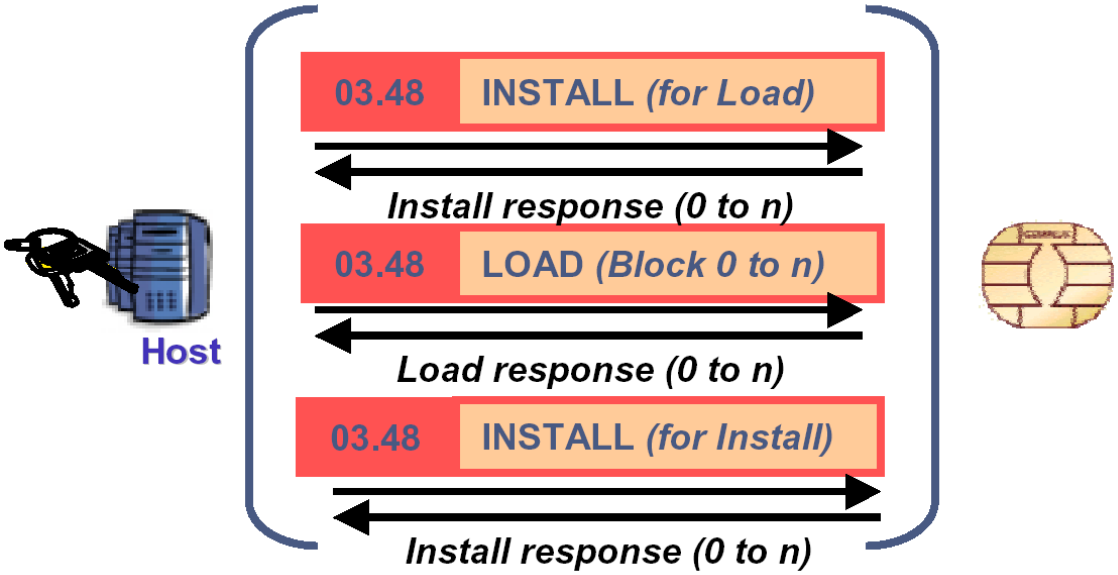


Figura 58: Proceso de descarga del paquete y la instalación del applet

6.2.6.1.3 Borrado de paquetes

El proceso de borrado de paquetes se realiza mediante el uso del comando DELETE. La SIM lleva a cabo el procedimiento de borrado de paquetes de la siguiente forma:

- Si quedan aplicaciones instaladas de este paquete, la tarjeta rechazará el borrado mediante el correspondiente código de error.
- Si otros paquetes hacen referencia al paquete que se quiere borrar, la tarjeta rechazará la solicitud de borrado con el correspondiente código de error.

6.2.6.1.4 Borrado de applets

El proceso de borrado de applets se lleva a cabo mediante el comando DELETE. La tarjeta SIM borrará todos los componentes que integran el applet.

6.2.6.1.5 Bloqueo y desbloqueo de applets

El procedimiento de bloqueo (y desbloqueo) de un applet, permite a la operadora y a los proveedores de servicios deshabilitar (y habilitar) un applet usando el comando SET STATUS. Cuando un applet se bloquea, no se podrá disparar o seleccionar, y todas las entradas a su menú se desactivarán (mediante el comando SET UP MENU).

6.2.6.1.6 Recuperación de los parámetros de un applet

La recuperación de los parámetros de un applet permite a la operadora o al proveedor de servicios solicitar remotamente los parámetros de un applet. Este procedimiento se realiza usando el comando GET DATA.

6.2.7 CONCLUSIÓN

El objetivo de esta sección ha sido el de describir todas las partes involucradas para hacer posible el funcionamiento de un applet (toolkit): la arquitectura GSM Java Card y la instalación del applet en la tarjeta SIM de forma remota a través de los recursos OTA. Gracias a la SIM API y a los mecanismos de gestión remota de applets, los applets toolkit pueden enviar comandos al terminal móvil (esto no ocurre con los applets normales de Java Card) y ser descargados a través de la red de telefonía móvil GSM. Esto supone que las operadoras pueden incorporar nuevos servicios basados en el SIM Application Toolkit de forma muy rápida (con la tecnología Java Card el tiempo que se tarda en desarrollar una aplicación es menor, y se puede instalar el applet independientemente de la localización en la que se encuentre el terminal móvil).

7 APLICACIÓN REALIZADA

Se ha realizado una aplicación que ofrece servicios a los usuarios de redes de telefonía móvil GSM. Para ello, se ha usado el SIM Application Toolkit. Este mecanismo, definido en el estándar GSM, permite que una aplicación (applet) que se encuentra en la tarjeta SIM, utilice el terminal móvil para que el usuario interaccione con dicha aplicación.

Las ventajas que presenta la utilización del SIM Application Toolkit, en los applets, se describen a continuación. Las aplicaciones (applets) residen y se ejecutan en la tarjeta SIM, por lo que el abonado podrá disponer de las mismas aplicaciones si cambia de terminal móvil. El terminal móvil solo deberá ser compatible con las tarjetas phase 2+ (casi todos los móviles fabricados a partir de 1999 soportan esta característica). Las operadoras de telefonía móvil pueden añadir, actualizar y eliminar aplicaciones a través del OTA. Otra ventaja, es la seguridad que aporta la tarjeta SIM, y los mecanismos de gestión remota y de seguridad para SAT definidos en la norma GSM 03.48.

7.1 OBJETIVOS DE LA APLICACIÓN REALIZADA

Esta aplicación posee unas funcionalidades que se pueden encontrar en las aplicaciones que traen las tarjetas SIM de las operadoras de telefonía móvil. Con los menús que presentan dichas aplicaciones, las operadoras de telefonía ofrecen multitud de servicios (suscripciones a la entrega de mensajes de noticias, servicio de localización de servicios públicos, juegos, ...) a los abonados.

La aplicación que se ha realizado para este proyecto fin de carrera, presenta una serie de funcionalidades que se detallan a continuación:

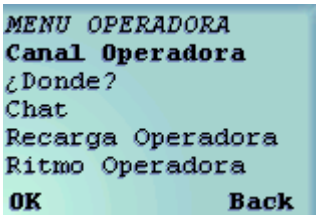
- Funcionalidad de presentación. La aplicación es capaz de desplegar menús en la pantalla del terminal móvil, que permiten seleccionar las distintas opciones que ofrece esta aplicación.
- Funcionalidad de cálculo. Esta funcionalidad permite pasar de euros a pesetas y de pesetas a euros. La complejidad de esta funcionalidad reside en que Java Card no permite el uso de números reales (por ejemplo, variables del tipo `float`). Para sobrepasar esta limitación de la tecnología Java Card, se ha tenido que diseñar un método que multiplique dos números representados por dos cadenas de caracteres.
- Funcionalidad de comunicación. Esta funcionalidad permite enviar correos electrónicos a través de SMS's (de forma transparente al usuario). El applet preguntará por la dirección de destino, el asunto del correo y por último se podrá escribir un texto de unos 80 caracteres. La aplicación del servidor, receptora de estos mensajes cortos, escapa de los objetivos de este proyecto fin de carrera.

7.2 SIMULACIÓN DE LA APLICACIÓN REALIZADA

En el momento en que se arranca el simulador (su aspecto se muestra en figura de abajo), aparecen dos mensajes en pantalla: SIMStudio, que es el nombre del simulador del teléfono y **Menu Operadora** (en negrita), que es el nombre de la aplicación. Si hubieran más applets toolkit instalados en la tarjeta, aparecerían en esta pantalla:

Figura 59: Teléfono móvil usado en la simulación de la aplicación realizada

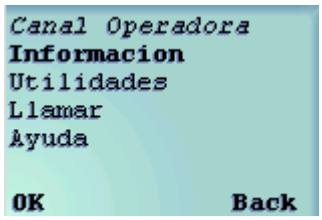
Para comenzar la simulación de la aplicación realizada hay que pulsar la opción OK (el botón superior izquierdo, debajo de la pantalla). A continuación aparecerá otra pantalla como esta:



En la parte superior de esta pantalla aparece el título del menú: MENU OPERADORA. Debajo del título, aparecen una serie de opciones que se pueden elegir pulsando en las teclas de arriba y abajo. La opción que se encuentre en negrita, es la que se puede seleccionar pulsando el botón OK. En esta pantalla deberían aparecer 10 opciones, pero el simulador solo puede mostrar 7 líneas. Se puede seguir pulsando la tecla de dirección hacia abajo para ver todas las opciones, hasta que se acaben. De todas

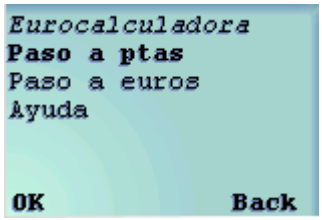
ellas, solo la primera, la novena y la décima tienen una funcionalidad asociada. Si se pulsa en alguna otra, se volverá a la pantalla de comienzo (SIMStudio).

Pulsando OK en la opción **Canal Operadora**, se puede ver otro menú más, igual al de la siguiente figura:



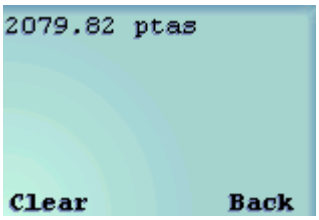
La primera línea de la pantalla muestra el título del menú (Canal Operadora). Las líneas de abajo son las distintas opciones que aparecen en este menú. La forma de seleccionarlas es la misma que la explicada anteriormente. Estas opciones no tienen ninguna funcionalidad asociada. Solo muestran cómo hacer un menú dentro de otro. Después de pulsar en alguna opción o tras cierto tiempo, el simulador volverá a la pantalla de comienzo.

A continuación, se mostrará la Eurocalculadora. Para acceder a esta funcionalidad se debe pulsar OK en la pantalla de inicio (SIMStudio). Luego, se debe buscar la opción **Eurocalculadora** mediante la tecla de dirección hacia abajo y pulsar OK. La pantalla que muestra esta opción, será como la de la figura:



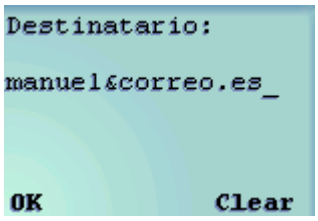
En la línea superior de la pantalla aparece el nombre del menú, que en este caso es Eurocalculadora. Además, aparecen tres opciones: Paso a ptas, Paso a euros y Ayuda. La opción de Ayuda no tiene funcionalidad asociada (podría mostrar un mensaje de ayuda). Las otras dos si tienen funcionalidad asociada. Para probarlas, se puede pulsar OK sobre la opción **Paso a ptas**. El resultado sería igual al presentado en la siguiente figura:

En esta figura se puede ver como la aplicación pregunta por la cantidad a convertir. Usando el teclado numérico del teléfono se puede teclear la cantidad a convertir, hasta un máximo de 20 dígitos incluyendo el punto (que se representa con el carácter `'*'`). Si se teclea `12*5` (12.5 €) y luego se pulsa la tecla OK, aparecerá la siguiente pantalla con la cantidad esperada de pesetas:

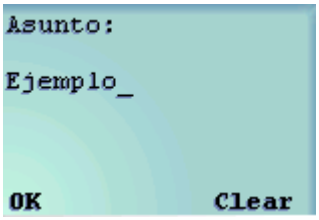


Tras un pequeño intervalo de tiempo, el simulador volverá a la pantalla de inicio (SIMStudio). La opción de Paso a euros de la Eurocalculadora no se explicará porque su funcionamiento es el mismo. La complejidad de la realización de la Eurocalculadora consiste en que para realizar la multiplicación por la constante de conversión de moneda, habría que utilizar variables que pudiesen contener números decimales. Pero como ya se sabe, la tecnología Java Card solo ofrece tipos de datos enteros. Haría falta un tipo de dato como float, que permita la multiplicación directa de números con decimales. Por ello, se ha implementado un método de multiplicación, que multiplica dos números contenidos en dos cadenas de caracteres (cada una representa a uno de los números). También se barajó la posibilidad de quitar los decimales, pasar los números a tipo short y realizar la multiplicación directa. Entonces, el resultado se podría pasar a una cadena de caracteres e insertar el punto. Pero esto supondría una limitación en la cantidad a convertir. Si se quisiera convertir 20 €, al multiplicar 20 · 166386 = 3327720, se sobrepasaría la cantidad que puede almacenar un tipo short. Por ello, se optó por multiplicar dos cadenas de caracteres, dando como resultado un número también contenido en una cadena de caracteres. En este caso, la única limitación es la cantidad de memoria a utilizar para almacenar los números implicados en la multiplicación.

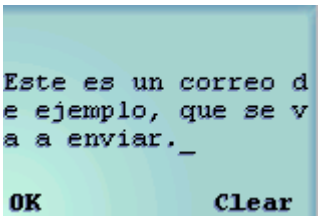
Por último, solo queda mostrar la funcionalidad de envío de correo electrónico. El proceso para acceder a esta funcionalidad, es el siguiente. En la pantalla de inicio se debe pulsar OK. A continuación se debe buscar la opción de **E-mail** y luego pulsar OK. La funcionalidad comienza preguntando la dirección del destinatario, como se muestra en la figura de abajo:



Se debe teclear usando el teclado alfa-numerico del terminal móvil. Pulsando de forma repetida en un mismo botón, se puede alcanzar el caracter deseado. En el caso de la figura se ha tecleado `manuel&correo.es`. Se ha comprobado que el símbolo @ no se puede teclear en el simulador. Una solución podría consistir en programar el servidor de correo para que cambiase el & por la @. Una vez terminada la introducción del destinatario se debe pulsar OK. Cuando esto ocurra, la aplicación solicitará la inserción del asunto (hasta 20 caracteres) del correo electrónico, como se muestra en la figura de abajo:



Como se puede comprobar en la figura, se ha insertado como Asunto la palabra `Ejemplo`. Pulsando OK, se podrá introducir el texto del correo. Un ejemplo de ello, se muestra a continuación:



Cuando se termine de teclear el correo (de tantos caracteres como la aplicación permita), hay que pulsar el botón OK, con lo que aparecerá un mensaje indicando que se está enviando el mensaje:

Después de esta pantalla, se vuelve a la pantalla de comienzo en la que se selecciona la aplicación realizada. Lo que no se muestra en el simulador son los datos enviados en el SMS, la parte del centro de conmutación de mensajes ni el procesado que tendría que realizar el servidor para enviar el correo electrónico.

7.3 ESTRUCTURA DE LA APLICACIÓN REALIZADA

La aplicación realizada se compone de un solo fichero (o clase) llamado `MenuOperadora.java`, que está definido dentro de un paquete denominado `MenuOperadoraEjemplo`.

A continuación se procederá al comentario detallado de cada una de las partes del código fuente de la aplicación realizada. Como en la sección anterior se han visto todas las funcionalidades y menús de la aplicación, no será difícil comprender la funcionalidad asociada a cada una de las partes del código fuente.

7.3.1 DEFINICIÓN DEL PAQUETE

Esta aplicación se incluye dentro de un paquete llamado `MenuOperadoraEjemplo`, es decir, el fichero `MenuOperadora.java`, se incluye dentro del directorio `MenuOperadoraEjemplo`. La necesidad de incluir la aplicación en un paquete, se debe a que la unidad mínima de carga de un applet en una tarjeta Java

Card es el fichero CAP, que representa un paquete (como se ha visto en un capítulo anterior).

```
package MenuOperadoraEjemplo;
```

7.3.2 DECLARACIÓN DE LOS PAQUETES IMPORTADOS

En esta aplicación, los paquetes necesarios a importar se incluyen en esta declaración:

```
import sim.toolkit.*;
import javacard.framework.*;
```

Aunque ya se han explicado las funcionalidades de cada uno de estos paquetes, a continuación se resumen aquí para mayor claridad:

- `sim.toolkit` : provee los medios para que los applets toolkit se puedan registrar a los eventos que proporciona el SIM Toolkit Framework. También sirve para manejar los objetos TLV y para enviar comandos proactivos que se ajustan a la especificación GSM 11.14.
- `javacard.framework` : provee clases soporte e interfaces para la funcionalidad principal de un applet Java Card. Lo más importante es que define una clase base `Applet`, que provee un soporte para la ejecución del applet y su interacción con el JCRE durante el tiempo de vida del applet.

7.3.3 CLASE `MENUOPERADORA`

Para ver esta clase, se dividirá la explicación de la misma en varios subapartados.

7.3.3.1 Definición de la clase

La clase de un applet Java Card normal, debe extender la clase `Applet`. Además, al tratarse de un applet toolkit, también debe implementar `ToolkitInterface` y `ToolkitConstants`. Los contenidos de estas clases se pueden consultar en los apéndices de este documento. El código asociado a esta definición de la clase, es el siguiente:

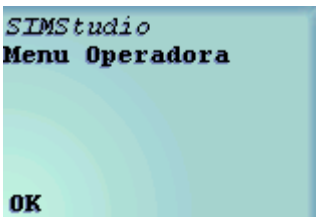
```
// La clase debe extender la clase Applet e implementar
// ToolkitInterface y ToolkitConstants
public class MenuOperadora extends javacard.framework.Applet
implements ToolkitInterface, ToolkitConstants{
```

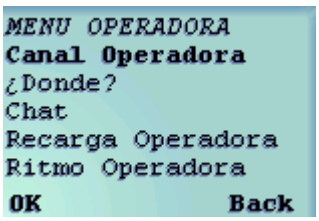
7.3.3.2 Definición de las variables de la clase

En esta sección se presentan la mayoría las variables que se van a usar en la clase. Se definen como objetos persistentes para asegurar que en tiempo de ejecución no va a faltar memoria RAM (del orden de 2 kbytes) para albergar por ejemplo, la pila de ejecución de Java Card. Todas estas variables se crearán durante el proceso de instalación del applet, con lo que se garantiza que siempre habrá espacio en memoria para ellas. Como se puede comprobar, la mayoría de las variables guardan cadenas de caracteres (un byte por carácter) que se corresponden con los textos que se van a mostrar en la pantalla del terminal móvil. También hay gran cantidad de constantes definidas que se distinguen por tener las palabras reservadas `static final`. El uso de estas palabras reservadas, supone un ahorro de espacio en memoria. A continuación se verá la funcionalidad de cada variable.

```
// Calificador de un comando, se usa en los objetos TLV
public static final byte CMD_QUALIFIER = (byte)0x80;

// Salida solicitada por el usuario cuando esta introduciendo caracteres
public static final byte EXIT_REQUESTED_BY_USER = (byte)0x10;

// menuEntry (entrada al menu del applet) es un array que contiene
// inicialmente la cadena: Menu Operadora
private byte[] menuEntry = {(byte)'M',(byte)'e',(byte)'n',(byte)'u',
                            (byte)' ',(byte)'O',(byte)'p',(byte)'e',
                            (byte)'r',(byte)'a',(byte)'d',(byte)'o',
                            (byte)'r',(byte)'a',};

// menuTitle (titulo del menu) es un array que contiene
// la cadena: MENU OPERADORA
private byte[] menuTitle = {(byte)'M',(byte)'E',(byte)'N',(byte)'U',
                            (byte)' ',(byte)'O',(byte)'P',(byte)'E',
                            (byte)'R',(byte)'A',(byte)'D',(byte)'O',
                            (byte)'R',(byte)'A',};
```

La variable `menuEntry` contiene el texto que presenta el simulador para poder elegir la aplicación realizada en la pantalla de inicio (SIMStudio). Cuando se pulsa OK, `menuTitle` da el nombre al menú desplegado (MENU OPERADORA) y debajo se presentan las distintas funcionalidades que ofrece la aplicación. Las pantallas a las que se hace referencia, son las siguientes:





Las variables (nombradas con `itemx`) que se muestran a continuación, se utilizan para dar nombre a las opciones del menú MENU OPERADORA (Canal Operadora, ¿Dónde?, Chat, ..., Eurocalculadora y E-mail).

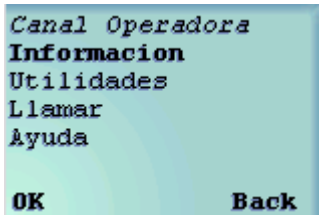
```
// item1 (elemento 1) es un array que contiene la cadena: Canal Operadora
```

```
private byte[] item1 = {(byte)'C',(byte)'a',(byte)'n',(byte)'a',(byte)'l',
                        (byte)' ',(byte)'O',(byte)'p',(byte)'e',(byte)'r',
                        (byte)'a',(byte)'d',(byte)'o',(byte)'r',(byte)'a',};
```

Cuando se elige la opción Canal Operadora, se presenta otro menú con más opciones. Las variables que representan los textos que dan nombre a esas opciones, se llaman `item1x`. El `1` viene de que Canal Operadora tiene el número 1 dentro de las primeras opciones a elegir. La `x` indica las opciones dentro del menú Canal Operadora. Simplemente se trata de una convención de nombres usada en este código fuente para conseguir mayor claridad.

```
//menu que está dentro de Canal Operadora
private byte[] item11 = {(byte)'I',(byte)'n',(byte)'f',(byte)'o',
                         (byte)'r',(byte)'m',(byte)'a',(byte)'c',
                         (byte)'i',(byte)'o',(byte)'n'};

private byte[] item12 = {(byte)'U',(byte)'t',(byte)'i',(byte)'l',
                         (byte)'i',(byte)'d',(byte)'a',(byte)'d',
                         (byte)'e',(byte)'s'};

private byte[] item13 = {(byte)'L',(byte)'l',(byte)'a',(byte)'m',
                         (byte)'a',(byte)'r'};

private byte[] item14 = {(byte)'A',(byte)'y',(byte)'u',(byte)'d',
                         (byte)'a'};

private Object[] ItemList1 = {item11, item12, item13, item14};
```

La variable `ItemList1`, contiene las referencias a las variables que contienen el texto de las opciones que se presentan dentro del menú Canal Operadora. La siguiente figura muestra el menú Canal Operadora:




```
// item2 (elemento 2) es un array que contiene la cadena: ¿Donde?
private byte[] item2 = {(byte)'¿',(byte)'D',(byte)'o',(byte)'n',(byte)'d',
                        (byte)'e',(byte)'?'};

// item3 (elemento 3) es un array que contiene la cadena: Chat
private byte[] item3 = {(byte)'C',(byte)'h',(byte)'a',(byte)'t'};

// item4 (elemento 4) es un array que contiene la cadena: Recarga Operadora
private byte[] item4 = {(byte)'R',(byte)'e',(byte)'c',(byte)'a',(byte)'r',
                        (byte)'g',(byte)'a',(byte)' ',(byte)'O',(byte)'p',
                        (byte)'e',(byte)'r',(byte)'a',(byte)'d',(byte)'o',
                        (byte)'r',(byte)'a',};

// item5 (elemento 5) es un array que contiene la cadena: Ritmo Operadora
private byte[] item5 = {(byte)'R',(byte)'i',(byte)'t',(byte)'m',(byte)'o',
```

```
                        (byte)' ',(byte)'O',(byte)'p',(byte)'e',(byte)'r',
                        (byte)'a',(byte)'d',(byte)'o',(byte)'r',(byte)'a',};

// item6 (elemento 6) es un array que contiene la cadena: Operadora Voz
private byte[] item6 = {(byte)'O',(byte)'p',(byte)'e',(byte)'r',(byte)'a',
                        (byte)'d',(byte)'o',(byte)'r',(byte)'a',(byte)' ',
                        (byte)'V',(byte)'o',(byte)'z'};

// item7 (elemento 7) es un array que contiene la cadena: Serv. Amigo
private byte[] item7 = {(byte)'S',(byte)'e', (byte)'r',(byte)'v',(byte)'.',
                        (byte)'A',(byte)'m', (byte)'i',(byte)'g',(byte)'o'};

// item8 (elemento 8) es un array que contiene la cadena: Juegos
private byte[] item8 = {(byte)'J',(byte)'u', (byte)'e',(byte)'g',(byte)'o',
                        (byte)'s'};
```

Las siguientes variables contienen todos los mensajes desplegados en la funcionalidad de la Eurocalculadora.

```
// item9 (elemento 9) es un array que contiene la cadena: Eurocalculadora
private byte[] item9 = {(byte)'E',(byte)'u', (byte)'r',(byte)'o',(byte)'c',
                        (byte)'a',(byte)'l', (byte)'c',(byte)'u',(byte)'l',
                        (byte)'a',(byte)'d', (byte)'o',(byte)'r',(byte)'a'};

    // Menu que esta dentro de Eurocalculadora
    private byte[] item91 = {(byte)'P',(byte)'a',(byte)'s',(byte)'o',
                             (byte)' ',(byte)'a',(byte)' ',(byte)'p',
                             (byte)'t',(byte)'a',(byte)'s'};

    private byte[] item92 = {(byte)'P',(byte)'a',(byte)'s',(byte)'o',
                             (byte)' ',(byte)'a',(byte)' ',(byte)'e',
                             (byte)'u',(byte)'r',(byte)'o',(byte)'s'};

    private byte[] item93 = {(byte)'A',(byte)'y',(byte)'u',(byte)'d',
                             (byte)'a'};

    private Object[] ItemList9 = {item91, item92, item93};

    private byte[] textCantidad = {(byte)'C',(byte)'a',(byte)'n',(byte)'t',
                                   (byte)'i',(byte)'d',(byte)'a',(byte)'d',
                                   (byte)':'};
```

Las variables `euros2ptas` y `ptas2euros`, contienen las constantes de conversión entre monedas. Los números enteros no se ponen como caracteres, para evitar la conversión de caracteres a números en el método `multiplica()`. El punto se representa con el carácter `'*'`.

```
    // Constante de conversion de euros a pesetas
    private byte[] euros2ptas = {(byte)1,(byte)6,(byte)6,(byte)'*',
                                 (byte)3,(byte)8,(byte)6,};

    // Constante de conversion de pesetas a euros
    private byte[] ptas2euros = {(byte)0,(byte)'*',(byte)0,(byte)0,
                                 (byte)6,(byte)0,(byte)1,(byte)0,(byte)1};
```

La variable `cadentrada`, es un buffer de 21 bytes (un byte por carácter) que contendrá el número introducido por el usuario mediante el teclado del terminal móvil. La variable `solucion` es otro buffer de 35 bytes que guarda la solución de la multiplicación realizada entre el número introducido por el usuario y una de las constantes de conversión definidas arriba.

```
// Buffer de entrada
private byte cadentrada[] = new byte[21];

// Buffer de solucion, se declara aqui por si no queda RAM en el
// momento de ejecutarse multiplica()
private byte solucion[]= new byte[35];
```

A continuación, se muestran las variables que contienen los textos a presentar por pantalla en la funcionalidad de E-mail. La distribución que realiza de los bytes de usuario que se pueden enviar en un SMS es la siguiente. Los primeros 20 bytes se reservan para la dirección de destino. Los siguientes 20 bytes se dedican para contener el Asunto. Y el resto de los bytes se destinan al texto del correo electrónico. Este reparto de bytes también se tendrá en cuenta en la aplicación servidora (que no se realiza en este proyecto fin de carrera), que se encargará de recibir los mensajes cortos y enviar los correos electrónicos.

```
// item10 (elemento 10) es un array que contiene la cadena: E-mail
private byte[] item10 = {(byte)'E',(byte)'-', (byte)'m',(byte)'a',(byte)'i',
                        (byte)'l'};

    private byte[] destinatario = {(byte)'D',(byte)'e',(byte)'s',(byte)'t',
                                   (byte)'i',(byte)'n',(byte)'a',(byte)'t',
                                   (byte)'a',(byte)'r',(byte)'i',(byte)'o',
                                   (byte)':'};

    private byte[] asunto = {(byte)'A',(byte)'s',(byte)'u',(byte)'n',
                                   (byte)'t',(byte)'o',(byte)':'};

    // Establezco una convencion en cuanto a la division de la informacion
    // que se puede enviar a traves de un SMS. 20 caracteres para la
    // direccion del destinatario y otros 20 para el asunto. El
    // resto del SMS se deja para texto. Esta convencion tambien
    // se usara en el servidor receptor del SMS.

    // Offset de direccion del destinatario
    public static final byte OFF_DIRECCION = (byte)0;
    // Offset de comienzo de asunto
    public static final byte  OFF_ASUNTO = (byte)19;
    // Offset de comienzo del texto
    public static final byte  OFF_TEXTO = (byte)39;
    // Para el texto quedan 128 - 40 = 88 caracteres

    // mail es un array que va a contener el SMS
    // (es decir, el correo electronico y la cabecera del SMS)
    private byte mail[] = new byte[255];
    private byte auxmail[] = new byte[255];

    // Numero del servidor al que dirigido el SMS. No hace falta el
    // numero del SMSC (centro de conmutacion de mensajes). Si el
    // ME no encuentra un numero valido de SMSC, inserta el numero del SMSC
    // que el ME tiene guardado
```

```
private byte[] nservidor = {(byte)'+',(byte)'3',(byte)'4',(byte)'6',
                            (byte)'0',(byte)'0',(byte)'0',(byte)'0',
                            (byte)'0',(byte)'0',(byte)'0',(byte)'0'};

// Tamaño maximo del número de telefono del servidor
private static final byte maxSizePhoneNum = (byte)12;
```

A partir de aquí se comienza la construcción de un SMS. Los campos presentes en un SMS se especifican en la siguiente tabla (para más información consultar la norma TS 23.040):

Tabla 16: Campos de un SMS

Campo		P1	P2	Descripción
TP-MTI	TP-Message-Type-Indicator	Ob	2b	Parámetro que describe el tipo del mensaje.
TP-RD	TP-Reject-Duplicates	Ob	b	Parámetro que indica si el centro de conmutación aceptará un SMS-SUBMIT (mensaje originado en el ME), que se encuentra en el centro de conmutación, y que tiene el mismo TP_MR y la misma TP-DA, que un mensaje anterior con la misma dirección de origen.
TP-VPF	TP-Validity-Period-Format	Ob	2b	Parámetro que indica si está presente o no el campo TP-VP.
TP-RP	TP-Reply-Path	Ob	b	Parámetro que indica la solicitud de un camino de respuesta.
TP-UDHI	TP-User-Data-Header-Indicator	Op	b	Parámetro que indica si el campo TP-UD contiene una cabecera.
TP-SRR	TP-Status-Report-Request	Op	b	Parámetro que indica si el terminal móvil solicita un informe acerca del estado del mensaje enviado.
TP-MR	TP-Message-Reference	Ob	E	Parámetro que indica el tipo de mensaje SMS-SUBMIT.
TP-DA	TP-Destination-Address	Ob	2-12o	Dirección de la entidad a la que va dirigido el mensaje.
TP-PID	TP-Protocol-Identifier	Ob	o	Parámetro que identifica al protocolo de nivel superior, si es que lo hay.
TP-DCS	TP-Data-Coding-Scheme	Ob	o	Parámetro que identifica el esquema de codificación usado en el campo TP-User-Data.
TP-VP	TP-Validity-Period	Op	o/7o	Parámetro que indica el instante a partir del cual el mensaje no será válido.
TP-UDL	TP-User-Data-Length	Ob	E	Parámetro que indica la longitud del campo TP-User-Data.
TP-UD	TP-User-Data	Op	*	Campo de datos.

P1: Obligatorio (Ob) u opcional (Op).
P2: Entero (E), bit (b), 2 bits (2b), octeto (o), 7 octetos (7o), 2-12 octetos (2-12o).
*: Depende del campo TP-DCS.

La siguiente figura muestra cual es la posición de cada uno de los campos que forman un SMS-SUBMIT (mensaje originado en el terminal móvil):

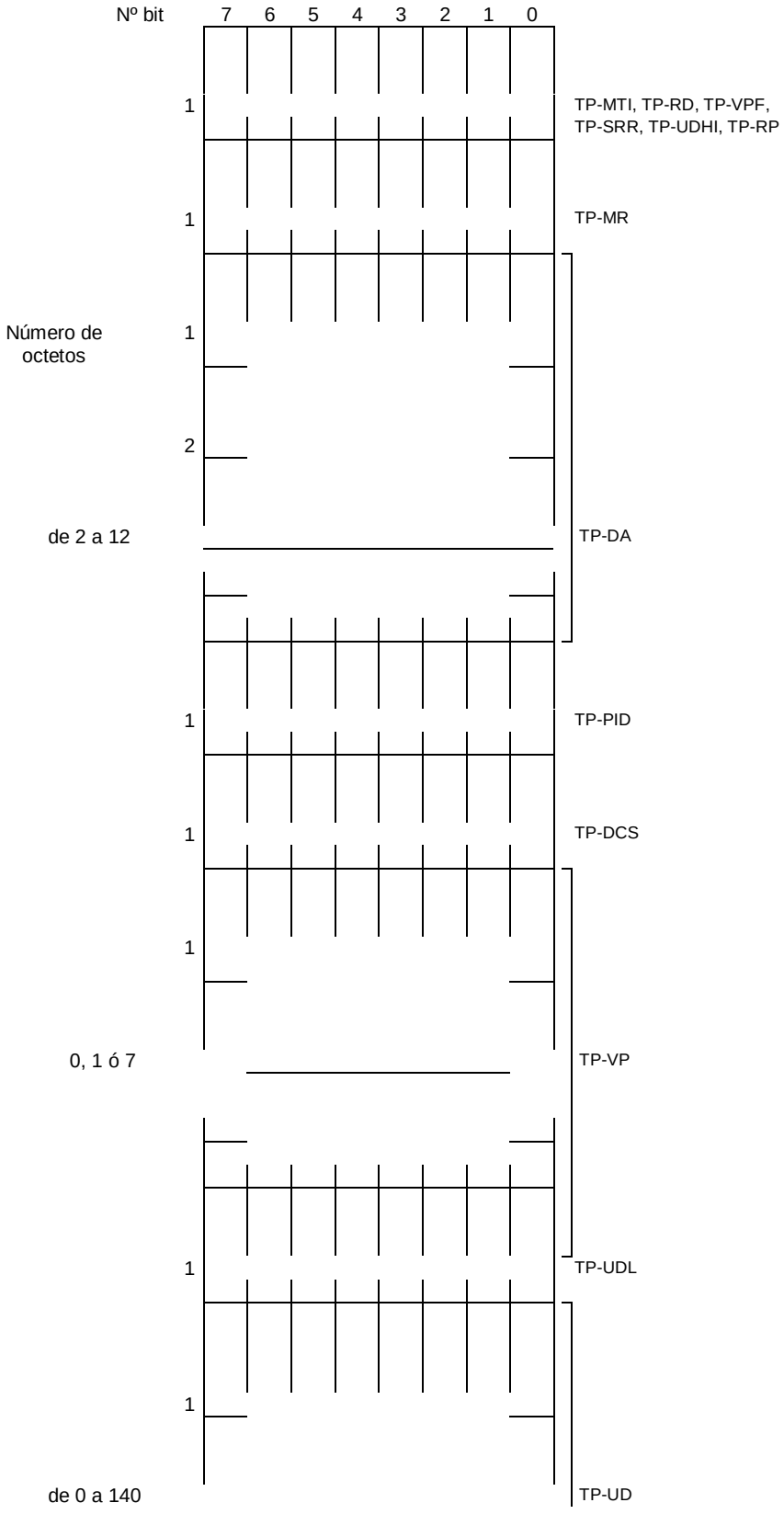


Figura 60: Campos de un SMS-SUBMIT

```
// Un SMS se compone de los siguientes campos TP.
```

```
// El orden es el de aparición.

// El campo TP-HEADER tiene los siguientes bits, con sus valores
// MTI = SMS-SUBMIT, RD = 0, VPF = 10, SRR=0, UDHI=1, RP=0
private static final byte TP_HEADER  =(byte)0x51;

// Campo referencia de mensaje, es un byte que identifica a
// un mensaje. Se puede empezar en 0
private static final byte TP_MR = (byte)0x00;

// Campo identificador de protocolo, el valor 0x41, indica codificacion
// de 8 bits en los datos
private static final byte TP_PID = (byte)0x41;

// TP_DCS = 0xF4 => Data Coding/message class 8-bit data Class 2
private static final byte TP_DCS = (byte)0xF6;

// Periodo de validez. El valor 0x00 significa 5 minutos si el bit
// VPF está puesto en modo relativo
private static final byte TP_VP = (byte)0x00;

// Para la cabecera 03.48 (que se encuentra dentro de los datos de
// usuario). Son valores necesarios para enviar el mensaje.
private static final byte TPudhl = (byte)2;
private static final byte Info0340Hi = (byte)0x70;
private static final byte Info0340Lo = (byte)0;
private static final byte CPLhi = (byte)0;
private static final byte CPLlo = (byte)0;
private static final byte CHL = (byte)13;
private static final byte SPI = (byte)0;
private static final byte KIC = (byte)0x10;
private static final byte KID = (byte)0x10;
private static final byte TAR = (byte)0x00; // Cambiado
private static final byte CNTR = (byte)00;
private static final byte Padding = (byte)00;

private static final byte Header0340Len = (byte)3;
private static final byte Header0348Len = (byte)16;

// Construccion de la cabecera
private static final byte[] Header0348 = {TPudhl,Info0340Hi,Info0340Lo,
                        CPLhi, CPLlo, CHL, SPI, SPI, KIC, KID,(byte)0,
                        (byte)0, (byte)3, CNTR, CNTR, CNTR, CNTR, CNTR,
                         Padding};

// Mensaje que se muestra por pantalla a la hora de enviar el SMS
private byte[] enviando = {(byte)'E',(byte)'n',(byte)'v',(byte)'i',
                          (byte)'a',(byte)'n',(byte)'d',(byte)'o'};
```

La variable `ItemList` contiene las referencias a las variables que contienen el texto de las opciones que se presentan dentro del menú MENU OPERADORA.

```
// ItemList es un array de clases Object (superclase de byte)
private Object[] ItemList = { item1, item2, item3, item4, item5, item6,
                              item7, item8, item9, item10};
```

El fin de las siguientes variables se verá a medida que se vayan usando en el código.

```
// La clase ToolkitRegistry permite al applet toolkit registrar
// su configuracion (eventos soportados)
private ToolkitRegistry reg;

// itemId guarda el identificador del item seleccionado
private byte itemId;

// result guarda el resultado de la ejecucion de un comando proactivo
private byte result;

// Variable auxiliar
private short aux;

// Variable auxiliar utilizada en un bucle while
private boolean repeat;
```

7.3.3.3 Constructor del applet `MenuOperadora()`

El método `install()` (que se verá a continuación), llama a este método para definir el texto (dado en la variable `menuEntry` = Menu Operadora) que aparecerá en el menú de selección de applets (ó aplicaciones) toolkit y registra al applet en los eventos de selección de menús (u opciones).

```
/**
* Constructor del applet
*/
public MenuOperadora() {
  // Consigue una referencia de su entrada en el registro toolkit del
  // SIM Toolkit Framework para que pueda controlar el registro
  // a ciertos eventos.
  reg = ToolkitRegistry.getEntry();

  // Define el nombre de la entrada al menu del applet y se registra en el
  // evento EVENT_MENU_SELECTION
  itemId = reg.initMenuEntry(menuEntry, (short)0x0000,
                             (short)menuEntry.length,
                             PRO_CMD_DISPLAY_TEXT, false, (byte) 0x00,
                             (short) 0x0000);
}
```

7.3.3.4 Método `install()`

El método `install()` crea la instancia del objeto `MenuOperadora` durante el proceso de instalación del applet en la tarjeta y luego registra el applet en el JCRE mediante la llamada al método `register()`.

```
/**
* El JCRE llama a este metodo para instalar el applet
*/
public static void install(byte bArray[], short bOffset, byte bLength) {
  MenuOperadora MyMenuOperadora = new MenuOperadora ();

  // Se llama a este metodo register para que el applet tome
```

```
    // el AID que contenga el fichero CAP
    MyMenuOperadora.register();
  }
```

7.3.3.5 Método `processToolkit()`

Este es el método estándar para manejar eventos toolkit en un applet toolkit. El Toolkit Handler lo llamará para que el applet procese el evento toolkit actual. En el caso de la aplicación realizada, se invoca a este método para notificar el evento de selección de la aplicación.

```
/**
* El SIM Toolkit Framework llama a este metodo, cuando se produce
* alguno de los eventos a los que esta registrado este applet
*/
public void processToolkit(byte evento) {
```

Las referencias a los manejadores se suelen obtener al comienzo de la llamada al método `processToolkit()`, menos el `ProactiveResponseHandler`, que es mejor obtenerlo después de una llamada al método `ProactiveHandler.send`. A continuación se realiza una descripción de cada manejador utilizado.

- `ProactiveHandler` : es un objeto temporal de punto de entrada al JCRE. Este manejador permite la construcción y envío de los comandos proactivos al ME. Para conseguir la referencia de este manejador hay que llamar al método estático `getTheHandler()`.
- `ProactiveResponseHandler` : es un objeto temporal de punto de entrada al JCRE. Un applet toolkit usará este manejador para conseguir la respuesta de los comandos proactivos. Para conseguir la referencia de este manejador hay que llamar al método estático `getTheHandler()`.

```
// Se obtienen las referencias de los manejadores
ProactiveHandler         proHdlr = ProactiveHandler.getTheHandler();
ProactiveResponseHandler rspHdlr;

// Segun el evento ...
switch(evento) {
  case EVENT_MENU_SELECTION:
```

Cuando se produce un evento del tipo `EVENT_MENU_SELECTION` (se ha seleccionado la aplicación realizada), hay que desplegar el menú con título MENU OPERADORA (variable `menuTitle`). Como ya se sabe, este menú se compone de 10 opciones a elegir (Eurocalculadora, E-mail, ...). El texto de cada una estas opciones se obtiene de la variable (`ItemList`) de lista de referencias a variables que contienen los textos.

Para desplegar el menú MENU OPERADORA hay que lanzar un comando proactivo `PRO_CMD_SELECT_ITEM`. El comando proactivo se acompañará del título del

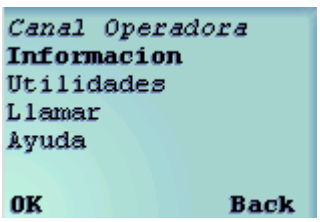
menú (MENU OPERADORA) mediante el método `ProactiveHandler.appendTLV()` (añade un objeto TLV al comando, que en este caso es el título del menú). Después se van añadiendo al comando, las 10 opciones de este menú, mediante el bucle `for`. Cada una de estas opciones se identifica con un número entero, dado por `(byte)(i+1)`. Es aquí donde se ve la utilidad de usar la variable `ItemList`, que permite acceder a los textos mediante el índice `i` del bucle `for`. Cuando el bucle `for` concluye, se envía el comando proactivo mediante el método `ProactiveHandler.send()`.

```
// Prepara el lanzamiento del comando SELECT ITEM
proHdlr.init(PRO_CMD_SELECT_ITEM,(byte)0,DEV_ID_ME);
// Envia el Titulo del menu
// TAG_ALPHA_IDENTIFIER indica que se van a enviar
// identificadores numericos
// TAG_SET_CR indica que el ME debe soportar el item por defecto
// (p. ej. el item anteriormente seleccionado)
proHdlr.appendTLV((byte) (TAG_ALPHA_IDENTIFIER | TAG_SET_CR),
                  menuTitle,(short)0,(short)menuTitle.length);
// Y añade los 10 items que se quieren representar en pantalla
// En el segundo parametro se pone el identificador del item
for (short i=(short)0; i<(short)10; i++) {
  proHdlr.appendTLV((byte) (TAG_ITEM | TAG_SET_CR),(byte) (i+1),
                    (byte[])ItemList[i],(short)0,
                    (short)((byte[])ItemList[i]).length);
}
// Se pide al SIM Toolkit Framework que envie el comando proactivo
// y se comprueba el resultado con RES_CMD_PERF (indica que el
// comando se ha llevado a cabo correctamente)
if ((result = proHdlr.send()) == RES_CMD_PERF) {
```

Una vez desplegado el menú anterior, la aplicación tiene que esperar a que el usuario escoja alguna de las opciones presentadas en el menú. Mediante la llamada al método `ProactiveResponseHandler.getItemIdentifier()`, la aplicación puede conocer el identificador de la opción que ha elegido el usuario. Con este identificador de opción y el bucle `switch`, se puede poner en marcha la funcionalidad de la aplicación elegida por el usuario.

```
// Se consigue una referencia al manejador ProactiveResponseHandler
rspHdlr = ProactiveResponseHandler.getTheHandler();
// Devuelve el identificador de la opcion escogida por el usuario
switch (rspHdlr.getItemIdentifier()) {
```

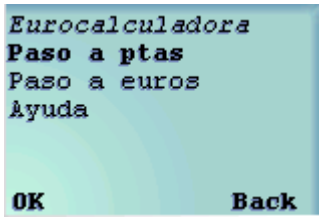
Si el usuario eligió Canal Operadora, la aplicación debe presentar el menú que se puede ver en la siguiente figura:



El procedimiento por el que se despliega el menú Canal Operadora es el mismo que el empleado en el menú anterior. Para no repetir los identificadores de opciones se utiliza la expresión `(byte)(10+i+1)` en el bucle `for`.

```
case 1: // Canal Operadora. Dentro, se presenta otro menu mas
        // Ninguna de estas entradas tiene funcionalidad
        // Prepara el lanzamiento del comando SELECT ITEM
        proHdlr.init(PRO_CMD_SELECT_ITEM,(byte)0,DEV_ID_ME);
        // Envia el titulo del menu: Canal Operadora = ItemList[0]
        proHdlr.appendTLV((byte)(TAG_ALPHA_IDENTIFIER|TAG_SET_CR),
                          (byte[])ItemList[0],(short)0,
                          (short)((byte[])ItemList[0]).length);
        // Añade cuatro opciones al menu
        for (short i=(short)0; i<(short)4; i++) {
          proHdlr.appendTLV((byte) (TAG_ITEM | TAG_SET_CR),
                    (byte)(10+i+1),(byte[])ItemList1[i],
                    (short)0,(short)((byte[])ItemList1[i]).length);
        }
        // Se envia el comando
        proHdlr.send();
        // Como ya se ha dicho, no hay funcionalidad asociada a
        // ninguno de los items u opciones
        break;
case 2:
case 3:
case 4:
case 5:
case 6:
case 7:
case 8: break;
case 9: // Eurocalculadora
```

Cuando se escoge la funcionalidad de Eurocalculadora, la aplicación presenta un menú mediante el mismo procedimiento descrito anteriormente. El menú es el siguiente:



A partir de aquí no se volverá a presentar la pantalla con el menú desplegado. En la sección 7.2 Simulación de la aplicación realizada de la página 183, se encuentran todos los menús desplegados por la aplicación realizada. Para no repetir los identificadores de opciones se utiliza la expresión `(byte)(90+i+1)` en el bucle `for`. Estos identificadores se usarán en el siguiente bucle `switch`.

```
        proHdlr.init(PRO_CMD_SELECT_ITEM,(byte)0,DEV_ID_ME);
        // Envia el titulo del menu: Eurocalculadora = ItemList[8]
        proHdlr.appendTLV((byte)(TAG_ALPHA_IDENTIFIER|TAG_SET_CR),
                          (byte[])ItemList[8],(short)0,
                          (short)((byte[])ItemList[8]).length);

        // Añade tres entradas al menu
        for (short i=(short)0; i<(short)3; i++) {
```

```
  proHdlr.appendTLV((byte) (TAG_ITEM | TAG_SET_CR),
           (byte)(90+i+1),(byte[])ItemList9[i],
           (short)0,(short)((byte[])ItemList9[i]).length);
}
// Se envia el comando
if ((result = proHdlr.send()) == RES_CMD_PERF) {
  rspHdlr = ProactiveResponseHandler.getTheHandler();
  switch (rspHdlr.getItemIdentifier()) {
```

El "case 91" representa el caso de paso de euros a pesetas. Esta funcionalidad comienza preguntando al usuario la cantidad en euros a convertir y luego presenta por pantalla la cantidad en pesetas.

Mediante el método `ProactiveHandler.initGetInput()`, se inicia el comando proactivo GET INPUT. Este método presenta en la pantalla del ME el texto "Cantidad?" y le ordena al ME que capture los caracteres tecleados por el usuario. El método `ProactiveHandler.send()` envía este comando proactivo. Como se puede observar, el envío del comando se realiza en un bucle `try – catch` para capturar y tratar las excepciones.

```
case 91: // Paso de euros a ptas
        do {
          repeat = false;
          try {
            // Al usuario se le pide que introduzca
            // la cantidad a convertir. Podra teclear
            // hasta 20 caracteres.
            // El primer parametro puesto a 0
            // indica que solo se podran teclear
            // numeros.
            proHdlr.initGetInput((byte)0,
                          DCS_8_BIT_DATA,
                          textCantidad,
                          (byte)0,
                          (short)textCantidad.length,
                          (short)0,(short)20);
            // Envio del comando proactivo
            proHdlr.send();
          }
          catch (ToolkitException MyException) {
            // Si no hay un elemento disponible
            if (MyException.getReason() ==
              ToolkitException.UNAVAILABLE_ELEMENT) {
                if (rspHdlr.getGeneralResult() !=
                  EXIT_REQUESTED_BY_USER)
            //Si el usuario no ha solicitado salir del
            //formulario vuelve a entrar en el bucle
            repeat = true;
            }
          }
        }
        while (repeat);

        // Recepcion de los datos de respuesta
        // Se inicializa cadentrada
        Util.arrayFillNonAtomic(cadentrada,
             (short)0,(short)cadentrada.length,
             (byte)' ');
```

Los caracteres tecleados por el usuario (caracteres numéricos ó el carácter `'*'`) se copian a la cadena de bytes `cadentrada`, anteriormente inicializada con el carácter de espacio en blanco. Después se llama al método `multiplica()` para multiplicar los números contenidos en `cadentrada` y `euros2ptas`. Este método devuelve la longitud de la solución en número de bytes (`aux`). Así se puede poner la palabra "ptas" después de la cantidad obtenida de la multiplicación. Después, se manda la solución a la pantalla del terminal móvil gracias a los métodos `ProactiveHandler.initDisplayText()` (que representa al comando proactivo DISPLAY TEXT) y al método `ProactiveHandler.send()`.

```
// Se copia lo tecleado por el usuario
// al array cadentrada
rspHdlr.copyTextString(cadentrada,(short)0);

// Multiplicacion
aux=(short)multiplica(cadentrada, euros2ptas);

// Se mandan los datos recibidos a la
// pantalla del ME
solucion[(short)aux] = (byte)' ';
solucion[(short)(aux+1)] = (byte)'p';
solucion[(short)(aux+2)] = (byte)'t';
solucion[(short)(aux+3)] = (byte)'a';
solucion[(short)(aux+4)] = (byte)'s';

proHdlr.initDisplayText((byte)0,
          DCS_8_BIT_DATA, solucion,
          (short)cadentrada[(short)0],
          (short)(20));

// Envio del comando proactivo
proHdlr.send();

break;
```

La estructura de la funcionalidad de paso de euros a pesetas es la misma que la anterior.

```
case 92:  //Paso de euros a pesetas
        do {
          repeat = false;
          try {
            // Al usuario se le pide que introduzca
            // la cantidad a convertir. Podra teclear
            // hasta 20 caracteres
            // El primer parametro puesto a 0
            // indica que solo se podran teclear
            // numeros.
            proHdlr.initGetInput((byte)0x00,
                          DCS_8_BIT_DATA,
                          textCantidad,
                          (byte)0,
                          (short)textCantidad.length,
                          (short)0,(short)20);
            // Envio del comando proactivo
            proHdlr.send();
          }
```

```
                        catch (ToolkitException MyException) {
                          // Si no hay un elemento disponible
                          if (MyException.getReason() ==
                            ToolkitException.UNAVAILABLE_ELEMENT) {
                              if (rspHdlr.getGeneralResult() !=
                                EXIT_REQUESTED_BY_USER)
                          //Si el usuario no ha solicitado salir del
                          //formulario vuelve a entrar en el bucle
                          repeat = true;
                          }
                        }
                      }
                      while (repeat);

                      // Recepcion de los datos de respuesta
                      // Se inicializa cadentrada
                      Util.arrayFillNonAtomic(cadentrada,
                           (short)0,(short)cadentrada.length,
                           (byte)' ');

                      // Se copia lo tecleado por el usuario
                      // al array cadentrada
                      rspHdlr.copyTextString(cadentrada,(short)0);

                      // Multiplicacion
                      aux=(short)multiplica(cadentrada, ptas2euros);

                      // Se mandan los datos recibidos a la
                      // pantalla del ME
                      solucion[(short)aux] = (byte)' ';
                      solucion[(short)(aux+1)] = (byte)'e';
                      solucion[(short)(aux+2)] = (byte)'u';
                      solucion[(short)(aux+3)] = (byte)'r';
                      solucion[(short)(aux+4)] = (byte)'o';
                      solucion[(short)(aux+5)] = (byte)'s';

                     proHdlr.initDisplayText((byte)0,DCS_8_BIT_DATA,
                               solucion,(short)cadentrada[(short)0],
                               (short)(20));

                      // Envio del comando proactivo
                      proHdlr.send();

                      break;

                  case 93: break; // No hay ayuda asociada
                }
              }
              break;
      case 10: // Aplicacion de correo electronico
```

Una vez escogida la funcionalidad de E-mail, el usuario ve como la aplicación le pide que teclee la dirección de destino del correo electrónico. Pero antes se realizan una serie de operaciones. La primera de estas operaciones es convertir el número del servidor, que se encuentra en una cadena de caracteres en formato ASCII (variable `nservidor`), mediante el método `asciiToAdn()`. Este método, que se define más adelante, convierte el número del servidor en un formato de numeración llamadoADN (Abbreviated Dialling Numbers, los dígitos del número se pasan a codificación BCD y también se añaden caracteres de señalización), necesario para que la red GSM entienda el número de destino del mensaje (norma GSM 04.08). Después de realizar la conversión del número telefónico del servidor, se va rellenando la cabecera del mensaje SMS, campo a campo.

```
Util.arrayFillNonAtomic(mail, (short)0,
                        (short)255,(byte)' ');
Util.arrayFillNonAtomic(auxmail, (short)0,
                        (short)255,(byte)' ');

 // Aqui comienza la preparacion del SMS que contiene
 // el E-mail

 // Se copia en auxmail el número de telefono
 //en formato ADN
 asciiToAdn(nservidor, maxSizePhoneNum, auxmail);

 // mail contendra el SMS
 short indice = (short)0;
 mail[(short)indice++] = TP_HEADER;
 mail[(short)indice++] = TP_MR;

 // Calculo del tamaño de la direccion TPDA (tamaño del
 // numero de telefono del servidor en formato ADN)
 byte len = CalculateADNLength(auxmail,(short)0);

 // Se copia el numero del servidor al SMS
 indice = (short)Util.arrayCopyNonAtomic(auxmail,(short)0,
                                         mail,indice,len);

 // Se sigue completando campos de la cabecera de un SMS
 mail[indice++] = TP_PID; // Identificador de protocolo
 mail[indice++] = TP_DCS; // Esquema de codificacion
 mail[indice++] = TP_VP;  // Periodo de validez del mensaje

 // Ahora viene el campo de longitud de datos de usuario
 // ó TP-UDL, todavía no se sabe la longitud, con lo que se
 // deja indicada la posicion del campo
 short saveTPUDLpos = (byte)indice++;
```

Ya se han completado los campos de la cabecera del SMS (a excepción de la longitud de datos de usuario del SMS, que todavía no se sabe). A continuación se rellena la cabecera 03.48, que se incluyen en los datos de usuario del SMS. La misión de esta cabecera es la de indicar el nivel de seguridad aplicado al mensaje que se va a enviar. En esta aplicación no se hará uso de medidas de seguridad. La especificación que define esta cabecera es la GSM 03.48.

```
 // Insercion de la cabecera 03.48 en los datos de usuario
 indice=(short)Util.arrayCopyNonAtomic(Header0348,(short)0,
         mail,indice,(short)(Header0348Len+Header0340Len));

 // Ya se tiene el SMS, ahora solo hace falta insertar el
 // correo en el campo de datos
 // indice contiene la posicion a partir de cual se pueden
 // ir introduciendo los datos
```

El SMS ya está casi formado, tan solo queda adquirir los datos tecleados por el usuario y calcular la longitud total del campo de datos de usuario del SMS. El procedimiento para adquirir los datos tecleados por el usuario, es el mismo que el utilizado para adquirir la cantidad a convertir en la Eurocalculadora. Como se puede observar, existen tres bucles `do – while` con la misma estructura que la presentada

para la Eurocalculadora. La misión de estos bucles es la de preguntar al usuario la dirección del destinatario, el asunto y el texto del correo electrónico. A medida que se van obteniendo los datos, se van introduciendo en la variable `mail`, que es el buffer que contiene el SMS a enviar, gracias al método `ProactiveResponseHandler.copyTextString()`. La codificación utilizada para los caracteres, es de 8 bits. Por lo tanto, en el campo de usuario podrán viajar hasta 140 bytes a compartir con la cabecera 03.48.

```
 // Primero se pregunta por la direccion de correo
 do {
   repeat = false;
   try {
     // Al usuario se le pide que introduzca
     // la direccion de correo. Podra teclear
     // hasta 20 caracteres
     // Con el primer parametro puesto a 1, se indica
     // que se pueden teclear caracteres y numeros
     proHdlr.initGetInput((byte)1, DCS_8_BIT_DATA,
                          destinatario,(byte)0,(short)
                          destinatario.length,(short)0,
                          (short)20);
     // Envio del comando proactivo
     proHdlr.send();
   }
   catch (ToolkitException MyException) {
     // Si no hay un elemento disponible
     if (MyException.getReason() ==
                    ToolkitException.UNAVAILABLE_ELEMENT) {
       if (rspHdlr.getGeneralResult() !=
                                       EXIT_REQUESTED_BY_USER)
         // Si el usuario no ha solicitado salir del
         // formulario vuelve a entrar en el bucle
         repeat = true;
     }
   }
 }
 while (repeat);

 // Recepcion de los datos de respuesta
 // Se copia lo tecleado por el usuario al array mail
rspHdlr.copyTextString(mail,(short)(indice+OFF_DIRECCION));

// Luego se pregunta por el asunto
 do {
   repeat = false;
   try {
     // Al usuario se le pide que introduzca
     // el asunto. Podra teclear
     // hasta 20 caracteres
     proHdlr.initGetInput((byte)1, DCS_8_BIT_DATA,
                          asunto,(byte)0,(short)asunto.length,
                          (short)0,(short)20);
     // Envio del comando proactivo
     proHdlr.send();
   }
   catch (ToolkitException MyException) {
     // Si no hay un elemento disponible
     if (MyException.getReason() ==
                    ToolkitException.UNAVAILABLE_ELEMENT) {
       if (rspHdlr.getGeneralResult() !=
                                       EXIT_REQUESTED_BY_USER)
         // Si el usuario no ha solicitado salir del
         // formulario vuelve a entrar en el bucle
```

```
            repeat = true;
          }
        }
      }
      while (repeat);

      // Recepcion de los datos de respuesta
      // Se copia lo tecleado por el usuario al array mail
      rspHdlr.copyTextString(mail,(short)(indice+OFF_ASUNTO));

     // Por ultimo el usuario debe escribir el texto
      do {
        repeat = false;
        try {
          // Al usuario se le pide que introduzca
          // el texto. Podra teclear hasta 140 menos
          // lo que ocupe la cabecera 0348
          proHdlr.initGetInput((byte)1,DCS_8_BIT_DATA,
                               null,(byte)0,(short)0,
                               (short)0,(short)(140-(Header0348Len
                               +Header0340Len+OFF_TEXTO+2)));
          // Envio del comando proactivo
          proHdlr.send();
        }
        catch (ToolkitException MyException) {
          // Si no hay un elemento disponible
          if (MyException.getReason() ==
                          ToolkitException.UNAVAILABLE_ELEMENT) {
            if (rspHdlr.getGeneralResult() !=
                                           EXIT_REQUESTED_BY_USER)
              // Si el usuario no ha solicitado salir del
              // formulario vuelve a entrar en el bucle
              repeat = true;
          }
        }
      }
      while (repeat);

      // Recepcion de los datos de respuesta
      // Se copia lo tecleado por el usuario al array mail
      short dataLength;
      // En principio dataLength guarda indice
      dataLength = (short)indice;
      indice=(short)rspHdlr.copyTextString(mail,(short)
                                             (indice+OFF_TEXTO));
      // Longitud del correo
      dataLength = (short)(indice - dataLength - 1);
```

A continuación se rellena el campo de longitud (que se dejó para después) del SMS y que indica la longitud total del campo de datos de usuario del SMS. También se calcula otro campo de longitud (CPL) de la cabecera 03.48

```
      // El campo TP-UDL es la longitud de los datos + su propia
      // longitud(1) + cabecera 03.48 + cabecera 03.40
      mail[saveTPUDLpos] = (byte)((byte)dataLength+
                    (byte)(Header0348Len + Header0340Len + 1));
      // Inicializa el valor de los bits LSB del CPL (Command
      // Packet Length)= Longitud de los datos + su propia
      // longitud + cabecera 03.48 exceptuando los dos bytes del
      // CPL y la cabecera 03.40
      mail[(byte)(saveTPUDLpos+2+Header0340Len)] =
                          (byte)(dataLength + (byte)(2 + CHL));
```

Con todos los campos del SMS ajustados, solo queda enviar el SMS. El procedimiento es parecido al del envío de cualquier comando proactivo al ME. Primero se indica que se trata de un comando proactivo SEND SHORT MESSAGE mediante el método `ProactiveHandler.init()`. A continuación se añade, al comando proactivo, un objeto TLV que contiene el texto ("Enviando") que se va a mostrar por pantalla cuando se envíe el mensaje. También se añade otro objeto TLV con el SMS completo. Por último se llama al método `ProactiveHandler.send()` para enviar este comando al terminal móvil.

```
                    // Comando de envio del SMS a la red
                    proHdlr.init(PRO_CMD_SEND_SHORT_MESSAGE,(byte)0,
                                                          DEV_ID_NETWORK );

                    //Primero se presenta por pantalla un mensaje diciendo que
                    //se esta enviando el mensaje
                    proHdlr.appendTLV(TAG_ALPHA_IDENTIFIER,enviando, (short)0,
                                                 (short)enviando.length);

                    // Objeto TLV que contiene el correo
                    proHdlr.appendTLV(TAG_SMS_TPDU,mail,(short)0,indice);

                    // Envio del comando al ME
                    proHdlr.send();

                    break;
              }
            }
            break;
      }
    }
```

7.3.3.6 Método `CalculateADNLength()`

Este método calcula la longitud del número telefónico del servidor en formato ADN. Este método es necesario para introducir la longitud de dicho número en uno de los campos de la cabecera del SMS. Se ha obtenido del juego `GuessNumber.java` que se puede obtener de la página web de la Sim Alliance (`www.simalliance.org`). Para más información consultar la norma GSM 04.08.

```
private static byte CalculateADNLength(byte[] ADN, short off){
  // Metodo obtenido del juego GuessNumber.java
  // Calcula la longitud del numero ADN
  return (byte)((byte)(ADN[off]/(byte)2)+(byte)(ADN[off]%(byte)2)+ (byte)2);
}
```

7.3.3.7 Método `shiftLeft()`

Este método se usa en el método `asciiToAdn()` para obtener el número telefónico del servidor en formato ADN. Se ha obtenido del juego

`GuessNumber.java` que se puede obtener de la página web de la Sim Alliance (`www.simalliance.org`). Para más información consultar la norma GSM 04.08.

```
private final static void shiftLeft(byte[] Data, short size, short times){
  // Metodo obtenido del juego GuessNumber.java
  // Este metodo se utiliza en el metodo asciiToAdn
  short ind;
  if (size == (short) 0 || size > (short) Data.length)
                                                  size = (short) Data.length;

  for(ind = 0; ind < (short)(size - times); ind++)
                                    Data[ind] = Data[(short)(ind + times)];
}
```

7.3.3.8 Método `asciiToAdn()`

Este método se usa para pasar el número telefónico del servidor de formato ASCII a formato ADN. A este método se le pasan tres parámetros. Dos de ellos son el array de entrada (número en formato ASCII) y el array de salida (número en formato ADN). El otro parámetro indica la longitud (en bytes) del número en formato ADN. Este método se ha obtenido del juego `GuessNumber.java` que se puede obtener de la página web de la Sim Alliance (`www.simalliance.org`). Para más información consultar la norma GSM 04.08.

```
private final static void asciiToAdn(byte[] asciiBuff,
                                     byte asciiLen, byte[] adnBuff){
  // Metodo obtenido del juego GuessNumber.java
  // Pasa un numero de telefono en codigo ascii, al formato ADN
  byte i, digit;
  byte bcdOff = 2;
  byte start = bcdOff;

 Util.arrayFillNonAtomic(adnBuff,(short)0,(short)adnBuff.length,(byte)0xFF);
  // Choose TON/NPI
  if(asciiBuff[0] == (byte) '+'){
    shiftLeft(asciiBuff, (short) (asciiLen & 0x00FF), (short) 1);
    asciiLen--;
    adnBuff[1] = (byte) 0x91;
  } else adnBuff[1] = (byte) 0x81; // by default

  adnBuff[0] = 0;

  for(i = 0; i < asciiLen; i++){
    switch(asciiBuff[i]){
      case (byte)'*' : digit = (byte) 0xA;
                       break;

      case (byte) '#' : digit = (byte) 0xB;
                        break;

      case (byte) 'p' : // DTMF Control digit separator (GSM 02.07 [3])
      case (byte) 'P' : digit = (byte) 0xC;
                        break;

     case (byte) '0' : case (byte) '1' :
      case (byte) '2' : case (byte) '3' :
      case (byte) '4' : case (byte) '5' :
```

```
      case (byte) '6' : case (byte) '7' :
      case (byte) '8' : case (byte) '9' :
        digit = (byte) (asciiBuff[i] - (byte) '0');
        break;

      default : continue;
    }

    adnBuff[0]++;

    if((adnBuff[0] % 2) != 0) adnBuff[start] = (byte) (digit << 4);
    else{
      adnBuff[start] = (byte) ((adnBuff[start] & 0xF0) | (digit & 0x0F));
      start++;
    }
  }

  // If not odd complete last byte with 0xF
  if((adnBuff[0] % 2) != 0){
   adnBuff[start] = (byte) ((adnBuff[start] & 0xF0) | 0x0F);
    start++;
  }

  // Swap
  for(i = bcdOff; i < start; i++)
    adnBuff[i] = (byte) ((byte)((byte)(adnBuff[i] << 4)
                 & 0xF0) | (byte)((byte)(adnBuff[i] >> 4) & 0x0F));
     }
```

7.3.3.9 Método `multiplica()`

Este método se utiliza para multiplicar dos números contenidos en dos cadenas de caracteres. A este método se le pasan dos parámetros. El primero de ellos (`a`) contiene el número tecleado por el usuario. El segundo (`b`) contiene la constante de conversión (`euros2ptas` ó `ptas2euros`). El método escribe, en la cadena de bytes `solucion`, el resultado de la multiplicación.

El método comienza convirtiendo la cadena de caracteres `a`. Cada byte de la cadena, se pasa de formato ASCII a formato numérico. A medida que se va realizando la conversión, se anotan en las variables `punto_a` y `tam_a`, la posición del punto y la longitud (en bytes) del número tecleado por el usuario. Si el número tecleado no tiene punto decimal, se añade un punto a la cadena de caracteres y luego un `0`.

En cuanto a la cadena de caracteres `b`, no hace falta pasarla de formato ASCII a formato numérico (ya está pasada a este último formato). De esta cadena tan sólo hace falta conocer la posición del punto (`punto_b`). La longitud es conocida (`tam_b`), ya que la constante está perfectamente contenida en una cadena de bytes (no sobra ni falta ningún byte).

```
public short multiplica (byte[] a, byte[] b) {
  //Multiplica dos numeros
  // a es una cadena de caracteres.
  // b es una cadena de bytes perfectamente ajustada
  // el punto se representa por '*'

  short punto_a = (short)-1;
```

```
short tam_a = (short)0;
short punto_b = (short)-1;
short tam_b = (short)0;
short acarreo = (short)0;
short i = (short)0;
short j = (short)0;
short h = (short)0;
short pos = (short)0;

// Se inicializa solucion, aqui se guardara el resultado
Util.arrayFillNonAtomic(solucion, (short)0,(short)solucion.length,
                        (byte)0);

//Se pone a en formato byte numerico y no de caracter
aux = (short)0;
for( ; aux < (short)a.length; aux++) {
  switch ((byte)a[(short)aux]) {
    case (byte)' ': if (punto_a == (short)-1) { // Si no hay punto
                      a[(short)aux]=(byte)'*'; // Se pone el punto
                      punto_a = (short)aux; //Se guarda posicion del punto
                      a[(short)(aux+1)]=(byte)0; // Seguido de un '0'
                      tam_a = (short)(aux+1); // Se guarda el tamaño
                    } else {
                      tam_a = (short)(aux - (short)1); // Se guarda
                    }                                  // el tamaño
                    aux = (short)a.length; // Se sale del bucle
                    break;
    case (byte)'*': punto_a = (short)aux;//Se guarda la posicion del punto
                    break;
    default :       a[(short)aux] = (byte)(a[(short)aux] - (byte)'0');
                    break;  // Se van convirtiendo a enteros
  }
}

aux = (short)0;
tam_b = (short)((short)b.length - (short)1); // Tamaño del array b

// Se busca la posicion del punto
for( ; aux < (short)tam_b; aux++) {
  if (b[(short)aux] == (byte)'*') {
    punto_b = (short)aux;
    aux = (short)b.length; // Se sale del bucle
  }
}
```

A partir de esta posición del código fuente, comienza el proceso de multiplicación. El método de multiplicación es el mismo que el utilizado para multiplicar dos números de forma manual (en un papel). La forma de disponer los dos números para multiplicar de forma manual, es la de poner uno arriba (variable `a`) y el otro abajo (variable `b`). El número de arriba se va multiplicando por cada dígito del número de abajo. Para hacerlo, se utilizan las variables `i` (para recorrer `a`) y `j` (para recorrer `b`). . `acarreo` irá guardando las decenas si, en el proceso de multiplicación de un dígito por otro, se produce un resultado mayor que 9. En la próxima multiplicación de un dígito por otro, se sumarán estas decenas a cantidad obtenida de la multiplicación. La variable `pos` irá recorriendo la variable `solucion` de derecha a izquierda. Para un `pos` fijo (que irá disminuyendo a la vez que lo hace `j`), `h` irá recorriendo `solucion`, a la vez que `i` va recorriendo la cadena `a`. En los dos bucles `for` que llevan a cabo la multiplicación, se realiza la suma de las multiplicaciones realizadas, pero no se tiene en cuenta el acarreo de las sumas (habrá bytes de `solucion` con valores mayores de 9). Tampoco se pone el punto al resultado guardado en `solucion`. Todo esto se soluciona

con otro bucle `for` que se encuentra a continuación, que también convertirá los bytes a formato ASCII.

```
acarreo= (short)0;
// pos indica la posicion en el array solucion, que se ira recorriendo
// de derecha a izquierda
pos = (short)(tam_a + tam_b + (short)1);

// En este bucle se realiza la multiplicacion, en el resultado
// no se pone el punto. Esto se deja para despues
// j recorre el array b (de derecha a izquierda)
// i recorre el array a (de derecha a izquierda)
for(j=(short)tam_b; j>=(short)0; j--){ // Se va recorriendo el array b
  if ((short)(j) == (short)(punto_b)) {} // Si se encuentra el punto se
  else{                                  // salta a la siguiente posicion
    pos--;
    h=(short)0; // h recorre solucion, a la vez que se recorre el array a
    acarreo=0;
    for(i=(short)tam_a; i>=(short)0; i--){ // Se va recorriendo el array a
      if ((short)(i) == (short)(punto_a)) {} // Si se encuentra el punto
      else{                                  // salta a la siguiente posicion
        // Multiplicacion
        aux=(short)((short)(a[(short)i]*b[(short)j]) + acarreo);
        // Se calcula el acarreo o parte decimal
        acarreo=(short)(aux/(short)10);
        // Se queda con las unidades
        aux=(short)(aux-(acarreo*(short)10));
        solucion[(short)(pos+h)]=(byte)
                                    (solucion[(short)(pos+h)]+(byte)aux);
        if (i==(short)0){ // Si se encuentra en la parte final de a, se
                          // guarda el acarreo en la siguiente cifra
                          // de la solucion
          solucion[(short)(pos+h-1)]=(byte)
                               (solucion[(short)(pos+h-1)]+(byte)acarreo);
        }
        h--;
      }
    }
  }
}

aux = (short)0;
acarreo = (short)0;
pos = (short)(-1); // Ahora, pos es una bandera que indica si se ha puesto
                   // el punto o no, en la solucion

// En este bucle se ira recorriendo solucion de derecha a izquierda.
// En el proceso de multiplicacion (arriba), no se ha tenido en cuenta
// el acarreo producido por la suma (solo el de la multiplicacion). Aqui
// se corrige esto.
// Cuando se llegue a la posicion del punto en la solucion, se desplaza
// la parte entera de la solucion a la izquierda para dejar paso al punto.
// Ademas, se ira pasando de valores enteros a caracteres, sumando el
// caracter '0'.
for (h=(short)(tam_a + tam_b + (short)0); h>=(short)0; h--){
  if ( h ==(short)(punto_a + punto_b + 0)) { // Es la posicion del punto?
    pos = (short)(solucion[(short)h]); // Se guarda el valor de ese campo
                                       // de solucion, para ponerlo en
                                       // el anterior, en la proxima
                                       // iteracion del bucle
    solucion[(short)h]=(byte)'.'; // Y se pone el punto
  }else {
    if(pos == (short)(-1)){ // Si todavía no se ha puesto el punto
      aux=(short)(solucion[(short)h]+(short)acarreo);
    }else{ // Si ya se ha puesto
```

```
          aux=(short)(pos+acarreo);
          pos=(short)solucion[(short)h];
        }
        acarreo=(short)(aux/(short)10); // Se calculan las decenas
        aux=(short)((short)aux-(short)(acarreo*(short)10)); // Unidades
        // Se ponen las unidades en solucion y ademas como caracter
        solucion[(short)h]= (byte)((byte)aux + (byte)'0');
      }
    }
```

El proceso de multiplicación ya está realizado. Solo queda quitar los ceros que sobren en la parte entera del número. Para ello, en el primer byte de `cadentrada` se guarda la posición a partir de la cual, se va a presentar `solucion` por pantalla. En la última parte de este método se devuelve la longitud (en bytes) de la solución, teniendo en cuenta que solo se desea un resultado con dos decimales.

```
    // Con este bucle, se eliminan los ceros que queden por delante
    // de la solucion
    for(aux = (short)0; solucion[(short)aux] == (byte)'0'; aux++);
    // cadentrada guarda el offset de la solucion en el array solucion
    cadentrada[(short)0] = (byte)aux;
    // En el caso de que se produzca una solucion del tipo 0.40404...
    // no interesa dejarlo como .40404...
    if (solucion[(short)aux] == (byte)'.') cadentrada[(short)0]--;

    // Se devuelve la longitud de la solucion, teniendo en cuenta que
    // solo se representara con dos decimales
    return (short)(punto_a + punto_b+ (short)3);
  }
```

7.3.3.10 Método `process()`

Esta aplicación no implementa ninguna funcionalidad si se selecciona como un applet de Java Card normal. Por ello, este método lanza una `ISOException` con el código de causa `ISO7816.SW_INS_NOT_SUPPORTED`. Así, se le indica al ME que esta aplicación no se comporta como un applet normal de Java Card, que implementa parte de su funcionalidad en el método `process()`.

```
  /**
  * El JCRE llama a este metodo una vez que se selecciona el applet
  * como un applet normal de Java Card
  */
  public void process(APDU apdu) {
    // Si una APDU selecciona el applet, lanza una excepcion ya que
    // el applet no ejecuta ninguna funcionalidad si se selecciona
    // como un applet Java Card normal
    ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
  }
}
```

Y aquí finaliza el código fuente de la aplicación realizada y que se encuentra en el fichero `MenuOperadora.java`.

8 CONCLUSIÓN Y LÍNEAS FUTURAS

Como se ha podido comprobar, este proyecto fin de carrera se ha enfocado hacia la implementación de aplicaciones con la tecnología Java Card en las tarjetas inteligentes usadas en dispositivos móviles (SIM y USIM). La prueba de ello se encuentra en la aplicación realizada. Con esta aplicación y los estudios que han conducido a su desarrollo, se ha pretendido dejar las bases de conocimiento necesarias para el desarrollo de nuevas aplicaciones.

Debido a la falta de entornos de desarrollo gratuitos que simulen una red de telecomunicaciones GSM, no se ha podido: implementar la parte del servidor que necesita la funcionalidad de envío de correo electrónico de la aplicación realizada, ni simular la descarga de la aplicación en una tarjeta SIM. De lo contrario, se hubiese podido desarrollar un servicio de correo totalmente operativo en una red de telecomunicaciones móviles GSM.

Además de implementar una aplicación como la realizada, este proyecto fin de carrera intenta presentar la importancia de las smart cards en aplicaciones de uso cotidiano y no solo en el sector de las comunicaciones móviles. También se ha prestado especial atención en el apartado práctico, presentando las herramientas y los ejemplos que se pueden utilizar para poder desarrollar nuevas aplicaciones de Java Card.

El enfoque dado a este proyecto fin de carrera, ha permitido que no se hayan tocado ciertos temas con la suficiente profundidad. Estos temas se pueden convertir en líneas futuras de investigación relacionadas con las tarjetas inteligentes. En los siguientes puntos se detallan algunas posibles líneas de investigación:

- Estudio de las tarjetas que no usen contactos para comunicarse con el CAD.
- Realización de una aplicación que use las posibilidades de criptografía de la tecnología Java Card.
- Implementación de una aplicación en una tarjeta inteligente real.
- Implementación de la aplicación que se ejecuta en el host y que interacciona con la aplicación de la tarjeta.
- Estudio más completo de los mecanismos de comunicación (SMS, CSD, GPRS ...) de las que dispone la tarjeta SIM/USIM para comunicarse con una aplicación remota.
- Estudio de la modalidad de envío de correo electrónico a través de SMS, establecido por la ETSI y el 3GPP. En la aplicación realizada no se usa esta modalidad.
- Implementación de una aplicación SIM Application Toolkit en una tarjeta SIM real.
- Integración entre el SIM Application Toolkit y WAP.
- Estudio de la descarga de aplicaciones SIM Applicaction Toolkit y el sistema de seguridad asociado, a través de OTA.

8.1 PLANIFICACIÓN TEMPORAL DEL PROYECTO

El proyecto se ha divido en cuatro tareas principales:

- Tarea de documentación. En ella, se han ido adquiriendo los documentos, tanto en formato papel como electrónico, y las herramientas necesarias, para realizar la aplicación objetivo de este proyecto fin de carrera.
- Tarea de realización de la aplicación. En esta tarea, se ha llevado a cabo el diseño, implementación y prueba de la aplicación realizada.
- Tarea de redacción del proyecto fin de carrera.
- Tarea de seguimiento del proyecto fin de carrera. Cada semana, se ha informado al profesor tutor, Don Antonio Jesús Sierra Collado, del estado de ejecución de las otras tareas. Así, ha podido seguir la evolución del proyecto y establecer las medidas correctivas que ha estimado oportunas. La información necesaria para el seguimiento del proyecto, se ha hecho llegar mediante correos electrónicos y reuniones en su despacho.

La siguiente figura muestra la evolución en el tiempo de las distintas tareas:

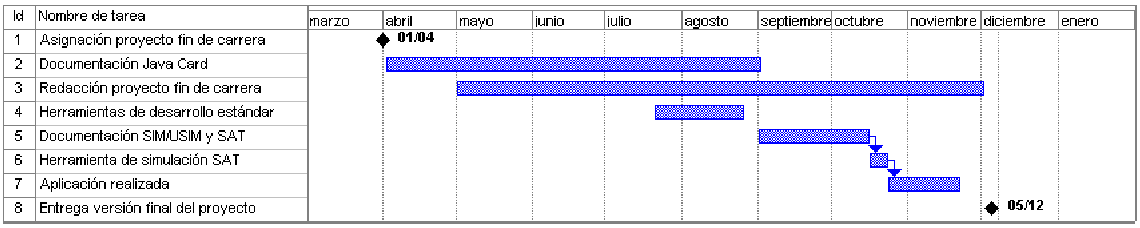


Figura 61: Diagrama temporal del proyecto fin de carrera

Las tareas que se incluyen en la figura, pertenecen a alguno de los tipos de tareas principales presentadas anteriormente.

8.2 PRESUPUESTO

En la realización del presupuesto se han tenido en cuenta dos aspectos que se detallan a continuación:

- El proyecto fin de carrera ha sido desarrollado por una sola persona cuyo nivel profesional podría corresponder al de un ingeniero junior.
- La ejecución del proyecto fin de carrera ha comprendido el periodo de tiempo entre abril y noviembre (ambos incluidos), del año 2003.

A continuación, se realiza un desglose del presupuesto y un resumen de dicho desglose.

8.2.1 DESGLOSE DEL PRESUPUESTO

El presupuesto del proyecto se puede desglosar en: coste de recursos humanos y coste de los recursos materiales empleados.

8.2.1.1 Coste de recursos humanos

El coste presupuestario de personal se ha calculado de la siguiente forma. Si se supone que se ha desempeñado una jornada de trabajo básica de 40 horas semanales y que se puede establecer un sueldo base de unos 1.500 € mensuales para un ingeniero junior, el coste de recursos humanos es:

Periodo de abril a noviembre (1.500 € x 8 meses)..... 12.000 €

Subtotal: 12.000 €

8.2.1.2 Coste de recursos materiales

Este apartado incluye los costes del hardware, software y consumibles utilizados.

8.2.1.2.1 Coste del software

El software utilizado ha sido software de libre de distribución o de versiones de evaluación. Por lo tanto el coste del software empleado es nulo.

8.2.1.2.2 Coste del hardware

El hardware empleado para desarrollar todo el proyecto ha sido un ordenador personal, una impresora y un pen drive (para transporte de datos voluminosos). Por lo tanto, el coste de hardware es el siguiente:

PC ..... 900 €
Impresora ..... 150 €
Pen drive ..... 75 €

Subtotal: 1125 €

8.2.1.2.3 Costes de los consumibles

Los costes asociados a los consumibles son los siguientes:

Electricidad consumida..... 90 €
Conexión básica a Internet..... 60 €
Material de oficina (papel, CD's...) ..... 50 €

Subtotal: 200 €

8.2.2 RESUMEN DEL PRESUPUESTO

Todo el coste anteriormente desglosado puede resumirse en la siguiente tabla:

Tabla 17: Coste total del proyecto fin de carrera

Coste de recursos humanos	
Periodo de abril a noviembre (1500 € x 8 meses)	12.000 €
Subtotal:	**12.000 €**

Coste de recursos materiales	
Coste del software	0 €
Coste del hardware	
PC	900 €
Impresora	150 €
Pen drive	75 €
Coste de los consumibles	
Electricidad consumida	90 €
Conexión Básica a Internet	60 €
Material de oficina	50 €
Subtotal:	**1.325 €**
TOTAL:	**13.325 €**

Por lo tanto, el coste total de proyecto fin de carrera es de 13.325 €.

9 GUÍA DE INSTALACIÓN Y PLANOS DEL CÓDIGO DE LA APLICACIÓN REALIZADA

Este capítulo se compone de una serie de secciones en las que se explican los procesos que hay que seguir para conseguir instalar y configurar el simulador que ha permitido desarrollar y probar la aplicación realizada. También se presenta el código fuente íntegro de la aplicación realizada. La información acerca de cómo está estructurado el código y la funcionalidad de cada una de sus partes, se encuentra en la sección 7.3 Estructura de la aplicación realizada en la página 186.

Además, se incluye otra sección sobre las herramientas de desarrollo estándar que se pueden utilizar para desarrollar y simular applets de Java Card. Para probar estas herramientas se incluye un ejemplo de applet Java Card, que además servirá para tener una idea más clara acerca de la estructura que deben tener estos applets. Y por último también se presenta un ejemplo de applet toolkit con funcionalidades que no se encuentran en el applet toolkit realizado.

9.1 HERRAMIENTA DE DESARROLLO UTILIZADA

Para desarrollar y probar el applet toolkit realizado, se ha elegido el simulador Aspects Developer de la compañía Aspects Software. Permite simular una tarjeta SIM, descargar aplicaciones en la SIM y probarlas en un interfaz gráfico amigable. En los siguientes apartados se detalla como instalar, configurar y hacer funcionar la herramienta.

9.1.1 PROCESO DE INSTALACIÓN DE LA HERRAMIENTA

El simulador elegido es una versión de evaluación y se puede conseguir de forma gratuita en la dirección de Internet: `http://www.aspectssoftware.com/devtools/index.html`. La compañía ofrece dos meses para probar esta versión de evaluación.

Antes de proceder al proceso de descarga hay que registrarse en la página web de esta compañía. Finalizado el proceso de registro se puede descargar la aplicación. Durante la descarga de la aplicación (en un fichero `.zip`) se recibe un correo electrónico parecido a este:

```
Thank you for your interest in Aspects developer. This email
contains the license key that you will need in order to install
the evaluation software. Please enter the information exactly as
shown below when prompted during the installation process.
Please note that this evaluation license expires on the 28 Oct
2003.

Username = Manuel Valenzuela
Expiry Date = 20031028
Key = 615423a60b6292653cda63a47b7b450ac77b9f7e

Thank you,
Aspects Software
```

Esta información será necesaria durante el proceso de instalación del simulador.

Para comenzar la instalación de la herramienta Aspects Developer, primero se debe tener una copia del Java Card Kit 2.1.2 en algún directorio. Este kit de desarrollo de Sun, se puede obtener de la página `http://java.sun.com/products/javacard/`. Se debe elegir la versión 2.1.2 para el sistema operativo Windows. Este entorno de desarrollo viene comprimido en un fichero `.zip` y se debe descomprimir en el directorio de trabajo, utilizando alguna herramienta de descompresión como WinZip. No hace falta realizar ninguna operación más con este entorno de desarrollo de Java Card.

También es necesario tener instalado el Java Development Kit (JDK) en el ordenador. La guía completa para conseguir e instalar este software para el sistema operativo Windows, se encuentra en la sección 9.3.1.1 Instalación del software de la página 228 .

Tras estos pasos, se puede instalar el simulador Aspects Developer. El siguiente paso es descomprimir el fichero `developer.zip` obtenido en el proceso de descarga realizado desde la página web de Aspects Software. El contenido de este fichero se puede dejar en cualquier directorio. Para comenzar la instalación se debe ejecutar el archivo `Setup.exe`, que se encontrará en el directorio anteriormente mencionado.

Después de aceptar los términos de la licencia, se deben rellenar los datos solicitados a partir del correo envíado por Aspects Software durante el proceso de descarga del simulador. En el caso del correo mostrado anteriormente, los valores de los campos a rellenar son los siguientes:

```
Username = Manuel Valenzuela
Expiry Date = 20031028
Key = 615423a60b6292653cda63a47b7b450ac77b9f7e
```

Después de introducir los datos, el simulador preguntará por el directorio de instalación y por la localización del acceso directo en la carpeta de `Programas` en el menú de `Inicio`. Cuando se haya acabado la etapa anterior, se debe escoger el tipo de instalación completa y luego asignar los ficheros `.hzp` a esta aplicación. Luego se debe presionar `Install` y cuando se termine la copia de los archivos, pulsar el botón `Finish`.

Desafortunadamente el programa no crea un acceso directo en la carpeta de `Programas` del menú de `Inicio`. Por lo tanto, se aconseja buscar el programa `AspectsDeveloper.exe` en el directorio `bin` que se creó durante la etapa de instalación. Por ejemplo, el programa podría encontrarse en: `C:\Archivos de programa\Aspects Software Limited\Aspects developer 2.0\bin`. Pinchando encima de `AspectsDeveloper.exe`, se puede crear un acceso directo en el Escritorio.

9.1.2 CONFIGURACIÓN DEL SIMULADOR

En esta sección se presentan los pasos a seguir para comenzar a trabajar con esta herramienta. Después de crear el acceso directo en el escritorio, se debe ejecutar el programa (pinchando en el acceso directo) para hacer funcionar esta herramienta de

simulación. Cuando haya finalizado la carga del programa se debe crear una "solución". Para hacerlo hay que desplegar los siguientes menús: `File ► New ► New Blank Solution`.

El programa pedirá un nombre (puede ser cualquiera) para la solución y que se le indique el directorio donde quiere almacenar la "solución" que se está definiendo. Después de crear la "solución", se deben definir los distintos paquetes o "proyectos" que la componen. Para esto, se debe ir a la opción `File ► New ► New Project` y proporcionar un nombre al proyecto. El nombre del proyecto debe coincidir con el nombre del paquete (el proyecto representa un paquete y puede haber varios en una solución) que se desea incluir en la solución. La aplicación realizada se encuentra en el paquete `MenuOperadoraEjemplo`, en el archivo `MenuOperadora.java`. Por lo tanto, para el caso de la aplicación realizada, se debe poner el nombre de proyecto `MenuOperadoraEjemplo`. También se deberá hacer un clic encima del icono `Sim Toolkit Applet` (se puede comprobar por los otros dos iconos, que se pueden simular otros tipos de applets de Java Card). En el apartado `Location` hay que indicar el directorio donde se encuentra el paquete. Solo se debe poner el directorio que contiene al directorio del paquete. Por ejemplo, si el fichero `MenuOperadora.java` se encontrara en la ruta `C:\java\MenuOperadoraEjemplo\MenuOperadora.java`, en el apartado de `Location` solo debería poner `C:\java`. Después hay que pulsar el botón para aceptar todos estos datos.

A continuación aparecerá un formulario. Se debe rellenar con los siguientes valores:

```
Class Name: MenuOperadora
Package Name: MenuOperadoraEjemplo
Package AID: AA AA AA AA AA A1
Applet AID: AA AA AA AA AA A2
Package Location C:\Java
```

Los AID's expuestos arriba, se ajustan a las reglas descritas para la asignación de AID's y que se encuentran en la página 51. También se debe tener en cuenta lo que se expuso acerca del TAR en la página 176.

Todo este proceso sirve para empezar una aplicación desde cero, sin los ficheros `.java`. Por ello el programa pregunta si quiere reemplazar `MenuOperadora.java`. Se debe contestar que `No`. Después aparecerá una ventana a la izquierda que contiene el código fuente de este fichero.

En la ventana `Proyect Explorer` (si no está disponible, se puede activar en `View`) hay que pinchar con el botón derecho del ratón en `MenuOperadora.java` y hacer clic en `Properties`. A continuación, el programa pedirá que se le indique el directorio donde se encuentra el Java Card Kit 2.1.2, descomprimido anteriormente. Este paso solo hay que hacerlo la primera vez que se usa el simulador. Después aparecerá un menú, en el que hay que rellenar los siguientes campos de SIM Options (con los otros no hay que hacer nada):

```
Maximum Menu Entries: 1
Maximum Menu Text Length: 100
Menu Entry Positions: 0000
```

Cuando se haya terminado, hay que pulsar el botón `Close`. Estos campos sirven para establecer características del ME que una aplicación SIM Application Toolkit debe conocer.

9.1.3 COMPILACIÓN DE LA APLICACIÓN REALIZADA

El apartado anterior ha dejado las condiciones necesarias poder compilar el ejemplo y generar el archivo CAP que se descargará a la tarjeta. Para ello, se debe desplegar el menú: `Project ► Rebuild Solution`. Se puede comprobar como el programa compila la aplicación realizada y, además, en la carpeta de `Generated Files` de la ventana de `Proyect Explorer`, presenta todos los ficheros generados (class, CAP, exportación,...).

Lo siguiente es cargar e instalar el fichero CAP en la tarjeta SIM. Esto se consigue deslizando el ratón hasta el apartado `Proyect ► Reload and Install Solution`. En la ventana `Card Explorer` se pueden observar los paquetes instalados en la tarjeta (los de la aplicación realizada, los paquetes de la SIM API y de Java Card, y otros).

Con la aplicación realizada instalada en una tarjeta SIM, solo quedaría probarla en un teléfono móvil. Para hacerlo, se deben desplegar los menús: `Tools ► Mobile Simulator ► Start Mobile Simulator`. El resultado es la aparición un teléfono móvil de color gris, pantalla verde y con un teclado numérico. El aspecto del simulador del teléfono móvil se presenta en la Figura 59 de la página 183.

Como se puede comprobar, el simulador tiene un teclado que se puede pulsar haciendo un clic con el ratón sobre cada tecla. Al teclado del teléfono también se puede acceder mediante el teclado numérico del PC. Además del teclado alfa-numérico, hay dos botones de dirección (arriba y abajo), un botón de colgar, otro de descolgar, y otros dos botones que están justo debajo de la pantalla y que sirven para seleccionar las opciones que se muestren en la parte inferior de la pantalla.

9.2 PLANOS DEL CÓDIGO DE LA APLICACIÓN REALIZADA

```
package MenuOperadoraEjemplo;
import sim.toolkit.*;
import javacard.framework.*;

public class MenuOperadora extends javacard.framework.Applet
implements ToolkitInterface, ToolkitConstants{

  public static final byte CMD_QUALIFIER = (byte)0x80;

  public static final byte EXIT_REQUESTED_BY_USER = (byte)0x10;

  private byte[] menuEntry = {(byte)'M',(byte)'e',(byte)'n',(byte)'u',
                              (byte)' ',(byte)'O',(byte)'p',(byte)'e',
                              (byte)'r',(byte)'a',(byte)'d',(byte)'o',
                              (byte)'r',(byte)'a',};

  private byte[] menuTitle = {(byte)'M',(byte)'E',(byte)'N',(byte)'U',
                              (byte)' ',(byte)'O',(byte)'P',(byte)'E',
                              (byte)'R',(byte)'A',(byte)'D',(byte)'O',
```

```
                            (byte)'R',(byte)'A',};

  private byte[] item1 = {(byte)'C',(byte)'a',(byte)'n',(byte)'a',(byte)'l',
                          (byte)' ',(byte)'O',(byte)'p',(byte)'e',(byte)'r',
                          (byte)'a',(byte)'d',(byte)'o',(byte)'r',(byte)'a',};

      private byte[] item11 = {(byte)'I',(byte)'n',(byte)'f',(byte)'o',
                               (byte)'r',(byte)'m',(byte)'a',(byte)'c',
                               (byte)'i',(byte)'o',(byte)'n'};

      private byte[] item12 = {(byte)'U',(byte)'t',(byte)'i',(byte)'l',
                               (byte)'i',(byte)'d',(byte)'a',(byte)'d',
                               (byte)'e',(byte)'s'};

      private byte[] item13 = {(byte)'L',(byte)'l',(byte)'a',(byte)'m',
                               (byte)'a',(byte)'r'};

      private byte[] item14 = {(byte)'A',(byte)'y',(byte)'u',(byte)'d',
                               (byte)'a'};

      private Object[] ItemList1 = {item11, item12, item13, item14};

  private byte[] item2 = {(byte)'¿',(byte)'D',(byte)'o',(byte)'n',(byte)'d',
                          (byte)'e',(byte)'?'};

  private byte[] item3 = {(byte)'C',(byte)'h',(byte)'a',(byte)'t'};

  private byte[] item4 = {(byte)'R',(byte)'e',(byte)'c',(byte)'a',(byte)'r',
                          (byte)'g',(byte)'a',(byte)' ',(byte)'O',(byte)'p',
                          (byte)'e',(byte)'r',(byte)'a',(byte)'d',(byte)'o',
                          (byte)'r',(byte)'a',};

  private byte[] item5 = {(byte)'R',(byte)'i',(byte)'t',(byte)'m',(byte)'o',
                          (byte)' ',(byte)'O',(byte)'p',(byte)'e',(byte)'r',
                          (byte)'a',(byte)'d',(byte)'o',(byte)'r',(byte)'a',};

  private byte[] item6 = {(byte)'O',(byte)'p',(byte)'e',(byte)'r',(byte)'a',
                          (byte)'d',(byte)'o',(byte)'r',(byte)'a',(byte)' ',
                          (byte)'V',(byte)'o',(byte)'z'};

  private byte[] item7 = {(byte)'S',(byte)'e', (byte)'r',(byte)'v',(byte)'.',
                          (byte)'A',(byte)'m', (byte)'i',(byte)'g',(byte)'o'};

  private byte[] item8 = {(byte)'J',(byte)'u', (byte)'e',(byte)'g',(byte)'o',
                          (byte)'s'};

  private byte[] item9 = {(byte)'E',(byte)'u', (byte)'r',(byte)'o',(byte)'c',
                          (byte)'a',(byte)'l', (byte)'c',(byte)'u',(byte)'l',
                          (byte)'a',(byte)'d', (byte)'o',(byte)'r',(byte)'a'};

      private byte[] item91 = {(byte)'P',(byte)'a',(byte)'s',(byte)'o',
                               (byte)' ',(byte)'a',(byte)' ',(byte)'p',
                               (byte)'t',(byte)'a',(byte)'s'};

      private byte[] item92 = {(byte)'P',(byte)'a',(byte)'s',(byte)'o',
                               (byte)' ',(byte)'a',(byte)' ',(byte)'e',
                               (byte)'u',(byte)'r',(byte)'o',(byte)'s'};

      private byte[] item93 = {(byte)'A',(byte)'y',(byte)'u',(byte)'d',
                               (byte)'a'};

      private Object[] ItemList9 = {item91, item92, item93};

      private byte[] textCantidad = {(byte)'C',(byte)'a',(byte)'n',(byte)'t',
                                     (byte)'i',(byte)'d',(byte)'a',(byte)'d',
                                     (byte)':'};

      private byte[] euros2ptas = {(byte)1,(byte)6,(byte)6,(byte)'*',
```

```
                                (byte)3,(byte)8,(byte)6,};

    private byte[] ptas2euros = {(byte)0,(byte)'*',(byte)0,(byte)0,
                                 (byte)6,(byte)0,(byte)1,(byte)0,(byte)1};

    private byte cadentrada[] = new byte[21];

    private byte solucion[]= new byte[35];

private byte[] item10 = {(byte)'E',(byte)'-', (byte)'m',(byte)'a',(byte)'i',
                         (byte)'l'};

    private byte[] destinatario = {(byte)'D',(byte)'e',(byte)'s',(byte)'t',
                                   (byte)'i',(byte)'n',(byte)'a',(byte)'t',
                                   (byte)'a',(byte)'r',(byte)'i',(byte)'o',
                                   (byte)':'};

    private byte[] asunto = {(byte)'A',(byte)'s',(byte)'u',(byte)'n',
                                   (byte)'t',(byte)'o',(byte)':'};

    public static final byte OFF_DIRECCION = (byte)0;
    public static final byte  OFF_ASUNTO = (byte)19;
    public static final byte  OFF_TEXTO = (byte)39;

    private byte mail[] = new byte[255];
    private byte auxmail[] = new byte[255];

    private byte[] nservidor = {(byte)'+',(byte)'3',(byte)'4',(byte)'6',
                                (byte)'0',(byte)'0',(byte)'0',(byte)'0',
                                (byte)'0',(byte)'0',(byte)'0',(byte)'0'};

    private static final byte maxSizePhoneNum = (byte)12;

    private static final byte TP_HEADER  =(byte)0x51;
    private static final byte TP_MR = (byte)0x00;
    private static final byte TP_PID = (byte)0x41;
    private static final byte TP_DCS = (byte)0xF6;
    private static final byte TP_VP = (byte)0x00;

    private static final byte TPudhl = (byte)2;
    private static final byte Info0340Hi = (byte)0x70;
    private static final byte Info0340Lo = (byte)0;
    private static final byte CPLhi = (byte)0;
    private static final byte CPLlo = (byte)0;
    private static final byte CHL = (byte)13;
    private static final byte SPI = (byte)0;
    private static final byte KIC = (byte)0x10;
    private static final byte KID = (byte)0x10;
    private static final byte TAR = (byte)0x00;
    private static final byte CNTR = (byte)00;
    private static final byte Padding = (byte)00;

    private static final byte Header0340Len = (byte)3;
    private static final byte Header0348Len = (byte)16;

    private static final byte[] Header0348={TPudhl, Info0340Hi, Info0340Lo,
                            CPLhi, CPLlo, CHL, SPI, SPI, KIC, KID, (byte)0,
                            (byte)0, (byte)3, CNTR, CNTR, CNTR, CNTR, CNTR,
                            Padding};

    private byte[] enviando = {(byte)'E',(byte)'n',(byte)'v',(byte)'i',
                               (byte)'a',(byte)'n',(byte)'d',(byte)'o'};

private Object[] ItemList = { item1, item2, item3, item4, item5, item6,
                              item7, item8, item9, item10};

private ToolkitRegistry reg;
```

```
private byte itemId;
private byte result;
private short aux;
private boolean repeat;

public MenuOperadora() {
  reg = ToolkitRegistry.getEntry();

  itemId = reg.initMenuEntry(menuEntry, (short)0x0000,
                             (short)menuEntry.length,
                             PRO_CMD_DISPLAY_TEXT, false, (byte) 0x00,
                             (short) 0x0000);
}

public static void install(byte bArray[], short bOffset, byte bLength) {
  MenuOperadora MyMenuOperadora = new MenuOperadora ();

  MyMenuOperadora.register();
}

public void processToolkit(byte evento) {
  ProactiveHandler         proHdlr = ProactiveHandler.getTheHandler();
  ProactiveResponseHandler rspHdlr;

  switch(evento) {
    case EVENT_MENU_SELECTION:
      proHdlr.init(PRO_CMD_SELECT_ITEM,(byte)0,DEV_ID_ME);
      proHdlr.appendTLV((byte) (TAG_ALPHA_IDENTIFIER | TAG_SET_CR),
                        menuTitle,(short)0,(short)menuTitle.length);

      for (short i=(short)0; i<(short)10; i++) {
        proHdlr.appendTLV((byte) (TAG_ITEM | TAG_SET_CR),(byte) (i+1),
                          (byte[])ItemList[i],(short)0,
                          (short)((byte[])ItemList[i]).length);
      }

      if ((result = proHdlr.send()) == RES_CMD_PERF) {
        rspHdlr = ProactiveResponseHandler.getTheHandler();

        switch (rspHdlr.getItemIdentifier()) {
          case 1:
                  proHdlr.init(PRO_CMD_SELECT_ITEM,(byte)0,DEV_ID_ME);

                  proHdlr.appendTLV((byte)(TAG_ALPHA_IDENTIFIER|TAG_SET_CR),
                                    (byte[])ItemList[0],(short)0,
                                    (short)((byte[])ItemList[0]).length);

                  for (short i=(short)0; i<(short)4; i++) {
                    proHdlr.appendTLV((byte) (TAG_ITEM | TAG_SET_CR),
                              (byte)(10+i+1),(byte[])ItemList1[i],
                              (short)0,(short)((byte[])ItemList1[i]).length);
                  }

                  proHdlr.send();
                  break;
          case 2:
          case 3:
          case 4:
          case 5:
          case 6:
          case 7:
          case 8: break;
          case 9:
                  proHdlr.init(PRO_CMD_SELECT_ITEM,(byte)0,DEV_ID_ME);

                  proHdlr.appendTLV((byte)(TAG_ALPHA_IDENTIFIER|TAG_SET_CR),
                                    (byte[])ItemList[8],(short)0,
                                    (short)((byte[])ItemList[8]).length);
```

```
for (short i=(short)0; i<(short)3; i++) {
  proHdlr.appendTLV((byte) (TAG_ITEM | TAG_SET_CR),
           (byte)(90+i+1),(byte[])ItemList9[i],
           (short)0,(short)((byte[])ItemList9[i]).length);
}

if ((result = proHdlr.send()) == RES_CMD_PERF) {
  rspHdlr = ProactiveResponseHandler.getTheHandler();
  switch (rspHdlr.getItemIdentifier()) {
    case 91:
            do {
              repeat = false;
              try {
                proHdlr.initGetInput((byte)0,
                               DCS_8_BIT_DATA,
                               textCantidad,
                               (byte)0,
                               (short)textCantidad.length,
                               (short)0,(short)20);

                proHdlr.send();
              }
              catch (ToolkitException MyException) {
                if (MyException.getReason() ==
                  ToolkitException.UNAVAILABLE_ELEMENT) {
                    if (rspHdlr.getGeneralResult() !=
                      EXIT_REQUESTED_BY_USER)

                repeat = true;
                }
              }
            }
            while (repeat);

            Util.arrayFillNonAtomic(cadentrada,
                 (short)0,(short)cadentrada.length,
                 (byte)' ');

            rspHdlr.copyTextString(cadentrada,(short)0);

            aux=(short)multiplica(cadentrada, euros2ptas);

            solucion[(short)aux] = (byte)' ';
            solucion[(short)(aux+1)] = (byte)'p';
            solucion[(short)(aux+2)] = (byte)'t';
            solucion[(short)(aux+3)] = (byte)'a';
            solucion[(short)(aux+4)] = (byte)'s';

            proHdlr.initDisplayText((byte)0,
                      DCS_8_BIT_DATA, solucion,
                      (short)cadentrada[(short)0],
                      (short)(20));

            proHdlr.send();

            break;
    case 92:
            do {
              repeat = false;
              try {
                proHdlr.initGetInput((byte)0x00,
                               DCS_8_BIT_DATA,
                               textCantidad,
                               (byte)0,
                               (short)textCantidad.length,
                               (short)0,(short)20);
```

```
                            proHdlr.send();
                          }
                          catch (ToolkitException MyException) {
                            if (MyException.getReason() ==
                              ToolkitException.UNAVAILABLE_ELEMENT) {
                                if (rspHdlr.getGeneralResult() !=
                                  EXIT_REQUESTED_BY_USER)
                            repeat = true;
                            }
                          }
                        }
                        while (repeat);

                        Util.arrayFillNonAtomic(cadentrada,
                             (short)0,(short)cadentrada.length,
                             (byte)' ');

                        rspHdlr.copyTextString(cadentrada,(short)0);

                        aux=(short)multiplica(cadentrada, ptas2euros);

                        solucion[(short)aux] = (byte)' ';
                        solucion[(short)(aux+1)] = (byte)'e';
                        solucion[(short)(aux+2)] = (byte)'u';
                        solucion[(short)(aux+3)] = (byte)'r';
                        solucion[(short)(aux+4)] = (byte)'o';
                        solucion[(short)(aux+5)] = (byte)'s';

                       proHdlr.initDisplayText((byte)0,DCS_8_BIT_DATA,
                                 solucion,(short)cadentrada[(short)0],
                                 (short)(20));

                        proHdlr.send();
                        break;

                    case 93: break;
                  }
                }
                break;

        case 10:
                Util.arrayFillNonAtomic(mail, (short)0,
                                        (short)255,(byte)' ');
                Util.arrayFillNonAtomic(auxmail, (short)0,
                                        (short)255,(byte)' ');

                asciiToAdn(nservidor, maxSizePhoneNum, auxmail);

                short indice = (short)0;
                mail[(short)indice++] = TP_HEADER;
                mail[(short)indice++] = TP_MR;

                byte len = CalculateADNLength(auxmail,(short)0);

                indice = (short)Util.arrayCopyNonAtomic(auxmail,(short)0,
                                                        mail,indice,len);

                mail[indice++] = TP_PID;
                mail[indice++] = TP_DCS;
                mail[indice++] = TP_VP;

                short saveTPUDLpos = (byte)indice++;

                indice=(short)Util.arrayCopyNonAtomic(Header0348,(short)0,
                        mail,indice,(short)(Header0348Len+Header0340Len));

                do {
                  repeat = false;
```

```
    try {
      proHdlr.initGetInput((byte)1, DCS_8_BIT_DATA,
                           destinatario,(byte)0,(short)
                           destinatario.length,(short)0,
                           (short)20);
      proHdlr.send();
    }
    catch (ToolkitException MyException) {
      if (MyException.getReason() ==
                     ToolkitException.UNAVAILABLE_ELEMENT) {
        if (rspHdlr.getGeneralResult() !=
                                       EXIT_REQUESTED_BY_USER)
          repeat = true;
      }
    }
  }
  while (repeat);

rspHdlr.copyTextString(mail,(short)(indice+OFF_DIRECCION));

  do {
    repeat = false;
    try {
      proHdlr.initGetInput((byte)1, DCS_8_BIT_DATA,
                         asunto,(byte)0,(short)asunto.length,
                         (short)0,(short)20);
      proHdlr.send();
    }
    catch (ToolkitException MyException) {
      if (MyException.getReason() ==
                     ToolkitException.UNAVAILABLE_ELEMENT) {
        if (rspHdlr.getGeneralResult() !=
                                       EXIT_REQUESTED_BY_USER)
          repeat = true;
      }
    }
  }
  while (repeat);

  rspHdlr.copyTextString(mail,(short)(indice+OFF_ASUNTO));

  do {
    repeat = false;
    try {
      proHdlr.initGetInput((byte)1,DCS_8_BIT_DATA,
                           null,(byte)0,(short)0,
                           (short)0,(short)(140-(Header0348Len
                           +Header0340Len+OFF_TEXTO+2)));
      proHdlr.send();
    }
    catch (ToolkitException MyException) {
      if (MyException.getReason() ==
                    ToolkitException.UNAVAILABLE_ELEMENT) {
        if (rspHdlr.getGeneralResult() !=
                                       EXIT_REQUESTED_BY_USER)
          repeat = true;
      }
    }
  }
  while (repeat);

  short dataLength;

  dataLength = (short)indice;
  indice=(short)rspHdlr.copyTextString(mail,(short)
                                         (indice+OFF_TEXTO));

  dataLength = (short)(indice - dataLength - 1);
```

```
                    mail[saveTPUDLpos] = (byte)((byte)dataLength+
                                  (byte)(Header0348Len + Header0340Len + 1));

                    mail[(byte)(saveTPUDLpos+2+Header0340Len)] =
                                       (byte)(dataLength + (byte)(2 + CHL));

                    proHdlr.init(PRO_CMD_SEND_SHORT_MESSAGE,(byte)0,
                                                            DEV_ID_NETWORK );

                    proHdlr.appendTLV(TAG_ALPHA_IDENTIFIER,enviando, (short)0,
                                                     (short)enviando.length);

                    proHdlr.appendTLV(TAG_SMS_TPDU,mail,(short)0,indice);

                    proHdlr.send();

                    break;
          }
        }
        break;
    }
  }

  private static byte CalculateADNLength(byte[] ADN, short off){
    return (byte)((byte)(ADN[off]/(byte)2)+(byte)(ADN[off]%(byte)2)+ (byte)2);
  }

  private final static void shiftLeft(byte[] Data, short size, short times){
    short ind;
    if (size == (short) 0 || size > (short) Data.length)
                                                    size = (short) Data.length;

    for(ind = 0; ind < (short)(size - times); ind++)
                                       Data[ind] = Data[(short)(ind + times)];
  }

  private final static void asciiToAdn(byte[] asciiBuff,
                                       byte asciiLen, byte[] adnBuff){
    byte i, digit;
    byte bcdOff = 2;
    byte start = bcdOff;

   Util.arrayFillNonAtomic(adnBuff,(short)0,(short)adnBuff.length,(byte)0xFF);

    if(asciiBuff[0] == (byte) '+'){
      shiftLeft(asciiBuff, (short) (asciiLen & 0x00FF), (short) 1);
      asciiLen--;
      adnBuff[1] = (byte) 0x91;
    } else adnBuff[1] = (byte) 0x81;

    adnBuff[0] = 0;

    for(i = 0; i < asciiLen; i++){
      switch(asciiBuff[i]){
        case (byte)'*' : digit = (byte) 0xA;
                         break;

        case (byte) '#' : digit = (byte) 0xB;
                          break;

        case (byte) 'p' :
        case (byte) 'P' : digit = (byte) 0xC;
                          break;

       case (byte) '0' : case (byte) '1' :
        case (byte) '2' : case (byte) '3' :
        case (byte) '4' : case (byte) '5' :
```

```
      case (byte) '6' : case (byte) '7' :
      case (byte) '8' : case (byte) '9' :
        digit = (byte) (asciiBuff[i] - (byte) '0');
        break;

      default : continue;
    }

    adnBuff[0]++;

    if((adnBuff[0] % 2) != 0) adnBuff[start] = (byte) (digit << 4);
    else{
      adnBuff[start] = (byte) ((adnBuff[start] & 0xF0) | (digit & 0x0F));
      start++;
    }
  }

  if((adnBuff[0] % 2) != 0){
   adnBuff[start] = (byte) ((adnBuff[start] & 0xF0) | 0x0F);
    start++;
  }

  for(i = bcdOff; i < start; i++)
    adnBuff[i] = (byte) ((byte)((byte)(adnBuff[i] << 4)
                 & 0xF0) | (byte)((byte)(adnBuff[i] >> 4) & 0x0F));
}

public short multiplica (byte[] a, byte[] b) {

  short punto_a = (short)-1;
  short tam_a = (short)0;
  short punto_b = (short)-1;
  short tam_b = (short)0;
  short acarreo = (short)0;
  short i = (short)0;
  short j = (short)0;
  short h = (short)0;
  short pos = (short)0;

  Util.arrayFillNonAtomic(solucion, (short)0,(short)solucion.length,
                          (byte)0);

  aux = (short)0;
  for( ; aux < (short)a.length; aux++) {
    switch ((byte)a[(short)aux]) {
      case (byte)' ': if (punto_a == (short)-1) {
                        a[(short)aux]=(byte)'*';
                        punto_a = (short)aux;
                        a[(short)(aux+1)]=(byte)0;
                        tam_a = (short)(aux+1);
                      } else {
                        tam_a = (short)(aux - (short)1);
                      }
                      aux = (short)a.length;
                      break;
      case (byte)'*': punto_a = (short)aux;
                      break;
      default :       a[(short)aux] = (byte)(a[(short)aux] - (byte)'0');
                      break;
    }
  }

  aux = (short)0;
  tam_b = (short)((short)b.length - (short)1);

  for( ; aux < (short)tam_b; aux++) {
    if (b[(short)aux] == (byte)'*') {
      punto_b = (short)aux;
```

```
        aux = (short)b.length;
      }
    }

    acarreo= (short)0;

    pos = (short)(tam_a + tam_b + (short)1);

    for(j=(short)tam_b; j>=(short)0; j--){
      if ((short)(j) == (short)(punto_b)) {}
      else{
        pos--;
        h=(short)0;
        acarreo=0;
        for (i=(short)tam_a; i>=(short)0; i--){
          if ((short)(i) == (short)(punto_a)) {}
          else{
            aux=(short)((short)(a[(short)i]*b[(short)j]) + acarreo);
            acarreo=(short)(aux/(short)10);
            aux=(short)(aux-(acarreo*(short)10));
            solucion[(short)(pos+h)]=(byte)(solucion[(short)(pos+h)]
                                            +(byte)aux);
            if (i==(short)0){
              solucion[(short)(pos+h-1)]=(byte)
                                  (solucion[(short)(pos+h-1)]+(byte)acarreo);
            }
            h--;
          }
        }
      }
    }

    aux = (short)0;
    acarreo = (short)0;
    pos = (short)(-1);

    for (h=(short)(tam_a + tam_b + (short)0); h>=(short)0; h--){
      if ( h ==(short)(punto_a + punto_b + 0)) {
        pos = (short)(solucion[(short)h]);
        solucion[(short)h]=(byte)'.';
      }else {
        if(pos == (short)(-1)){
          aux=(short)(solucion[(short)h]+(short)acarreo);
        }else {
          aux=(short)(pos+acarreo);
          pos=(short)solucion[(short)h];
        }
        acarreo=(short)(aux/(short)10);
        aux=(short)((short)aux-(short)(acarreo*(short)10));
        solucion[(short)h]= (byte)((byte)aux + (byte)'0');
      }
    }

    for(aux = (short)0; solucion[(short)aux] == (byte)'0'; aux++);

    cadentrada[(short)0] = (byte)aux;

    if (solucion[(short)aux] == (byte)'.') cadentrada[(short)0]--;

    return (short)(punto_a + punto_b+ (short)3);
  }

  public void process(APDU apdu) {
    ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
  }
}
```

9.3 ENTORNO DE DESARROLLO ESTÁNDAR Y EJEMPLOS

9.3.1 EL ENTORNO DE DESARROLLO

En esta sección se explica como desarrollar aplicaciones basadas en la plataforma Java Card y OpenCard Framework, usando herramientas de desarrollo estándar. Se recuerda que el OpenCard Framework (OCF) provee una solución independiente de la plataforma del lado del terminal, soportando Java.

En primer lugar se verá detalladamente el proceso de instalación de los paquetes y herramientas necesarias para desarrollar las aplicaciones, tanto en plataformas Windows como Linux.

Y en segundo lugar se explicarán algunos ejemplos de aplicaciones y como usar las herramientas de simulación estándares para comprobar si dichas aplicaciones se comportan correctamente.

9.3.1.1 Instalación del software

9.3.1.1.1 Software necesario

Para desarrollar una aplicación usando la plataforma Java Card y OpenCard Framework, se necesitan los siguientes paquetes:

- Obtenga el Java Development Kit (JDK), que se puede conseguir en la siguiente dirección: `http://java.sun.com/j2se/`.
- Obtenga el Java Card 2.2 Development Kit, en la dirección: `http://java.sun.com/products/javacard/`.
- Obtenga `javax.comm`. No hace falta si no se va a trabajar con un lector de tarjetas. Se puede conseguir en: `http://java.sun.com/products/javacomm`. Para instalarlo solo hay que seguir las instrucciones que se incluyen en fichero `Readme.html`. Hay que asegurarse de que el fichero `comm.jar` se añade a la variable `CLASSPATH`.
- Obtenga el OpenCard Framework (OCF) versión 1.2. Esta versión se encuentra comprimida (formato zip) en la siguiente dirección: `http://www.opencard.org/index-downloads.html`.

9.3.1.1.2 Instalación del software

A continuación se explica como instalar el software anteriormente mencionado, según el tipo de sistema operativo (plataformas Windows y Linux).

9.3.1.1.2.1 Procedimiento de instalación en Linux

9.3.1.1.2.1.1 Iniciar una sesión como administrador (root).

Para ello se debe abrir una ventana de comandos y teclear:

```
su
```

A continuación el sistema pedirá la contraseña de root.

9.3.1.1.2.1.2 Descomprimir el paquete de instalación

En el caso de que el paquete descargado sea del tipo `*.bin` ó `*.sh`, debe ejecutarse en un directorio. Para ello se debe teclear lo siguiente:

```
./j2sdk-1_4_1_02-linux-i586-rpm.bin
```

ó

```
./j2sdk-1_3_0-linux.sh
```

A continuación se muestran en pantalla una serie de advertencias. Si se está de acuerdo con las condiciones impuestas por la licencia, hay que teclear `yes`. Después se obtiene un fichero con extensión `rpm` en el mismo directorio.

9.3.1.1.2.1.3 Instalar el fichero `rpm`.

Para ello se debe introducir la siguiente orden:

```
rpm -iv j2sdk-1_4_1_02-fcs-linux-i586.rpm
```

ó

```
rpm -iv j2sdk-1_3_0-linux.rpm
```

Cuando el comando finaliza, se puede comprobar que se ha instalado el JDK en el directorio `/usr/java/j2sdk1.4.1_02` (esto es para el caso en que se haya instalado el JDK 1.4) ó en el directorio `/usr/java/jdk1.3` (para el caso en que se haya instalado el JDK 1.3). A partir de ahora solo se hará referencia a la versión 1.3 del JDK. Se puede ir al directorio `/usr/java/jdk1.3/bin` y comprobar que efectivamente se ejecuta el compilador y el intérprete de Java. Para ello teclear lo siguiente desde dicho directorio:

```
./javac
./java
```

Con lo que se obtienen mensajes característicos de ayuda.

9.3.1.1.2.1.4 Descomprimir el paquete de desarrollo de Java Card

En cuanto al kit de desarrollo de Java Card para Linux, sólo se encuentra disponible la versión 2.1.1 para Unix. La versión 2.1.1 de JavaCard es diferente de la 2.2. en cuanto a estructura de directorios, número de ejemplos (con lo que la sección de ejemplos que se detalla más adelante no describe correctamente el procedimiento para hacer funcionar los ejemplos de la versión 2.1.1) y requisitos. Para descomprimir esta versión, teclear lo siguiente en el directorio donde se descargó el fichero:

```
gzip -d java_card_kit-2_1_1-unix.tar.Z
```

El resultado es la obtención del fichero `java_card_kit-2_1_1-unix.tar`. Todavía hace falta volver a descomprimir:

```
tar -xf java_card_kit-2_1_1-unix.tar
```

Tras esta orden aparecerá un directorio llamado `jc211` que se puede copiar donde se desee. Para que esté junto al JDK se puede copiar el directorio `jc211` en `/usr/java`.

9.3.1.1.2.1.5 Descomprimir el paquete `javax.comm`

Para hacer funcionar el kit de desarrollo de Java Card 2.1.1 para Linux, hace falta el paquete `javax.comm` en su versión para procesadores x86. Para descomprimir este paquete, teclear lo siguiente en el directorio donde se descargó el fichero:

```
gzip -d javacomm20-x86[1].tar.Z
```

Con lo que se obtiene el fichero `javacomm20-x86[1].tar`. Todavía hace falta volver a descomprimir:

```
tar -xf javacomm20-x86[1].tar
```

Tras esta orden aparecerá un directorio llamado `commapi` que se puede copiar donde se desee si se tiene abierta una sesión como superusuario. Para que esté junto al JDK se puede copiar el directorio `commapi` en `/usr/java`.

A continuación se debe copiar el fichero `comm.jar` que se encuentra en el directorio `/usr/java/commapi`, en el directorio `/usr/java/jc211/lib`. Para evitar problemas al resto de los usuarios, el usuario root debe dar todos los permisos a los directorios que cuelgan de `/usr/java`.

9.3.1.1.2.1.6 Establecer las variables de entorno.

Para ello, hay que abrir el fichero `/etc/profile` en modo superusuario (root) e incluir las siguientes líneas:

```
JAVA_HOME=/usr/java/jdk1.3
JC21BIN=/usr/java/jc211/bin
CLASSPATH=$CLASSPATH:$JC21BIN/comm.jar:$JC21BIN/api21.jar:
$JAVA_HOME/src.jar
PATH=$PATH:$JAVA_HOME/bin:$JC21BIN
export JAVA_HOME
export JC21BIN
```

Después se debe salir de la sesión abierta. Se ha detectado que desde las ventanas de terminal del KDE, las variables de entorno no se ajustan bien. Se pueden utilizar las consolas accesibles pulsando `Control+Alt+1...6` ó las ventanas abiertas con `Eterm` si se prefiere usar un entorno gráfico. Para probar el correcto funcionamiento, se puede abrir una ventana o una consola como un usuario normal y comprobar que desde cualquier directorio se puede ejecutar lo siguiente:

```
javac
java
```

Si aparecen otros mensajes distintos a los que se obtuvo la última vez, se puede deber a que la versión de Linux ya incluye un entorno de desarrollo de Java. Para llamar

a las herramientas del JDK se les puede realizar un alias como se explica a continuación. El procedimiento consiste en editar el archivo `/etc/bashrc` como usuario root e incluir la siguiente línea por cada comando en el que se detecten problemas. Por ejemplo, en el caso en que no se ejecute el comando `java` del JDK, se debe incluir esta línea:

```
alias java=/usr/java/jdk1.3/bin/java
```

Después de incluir la línea, conviene salir de la sesión y volver a entrar como un usuario normal, para que los cambios surjan efecto. A continuación se puede comprobar si se pueden ejecutar las herramientas del JDK.

9.3.1.1.2.1.7 Notas adicionales

Las operaciones anteriores van orientadas a conseguir que todos los usuarios de la máquina Linux puedan usar las herramientas que se han instalado. Si se quiere configurar las herramientas de desarrollo para ciertos usuarios, se pueden realizar las operaciones (y no las anteriores) que se detallan a continuación (por cada usuario). En el fichero `/home/<usuario>/.bash_profile` hacer las mismas operaciones que se explican para el archivo `/etc/profile`. De la misma manera que para `/etc/bashrc`, editar el fichero `/home/<usuario>/.bashrc` e incluir los alias que haga falta.

9.3.1.1.2.2 Procedimiento de instalación en Windows 98

9.3.1.1.2.2.1 Ejecutar el programa de instalación del JDK

Este programa, previamente descargado, tendrá un nombre parecido al siguiente `j2sdk-1_3_1_07-windows-i586.exe` (se recomienda el uso del jdk1.3, ya que el jdk1.4 puede causar problemas con las herramientas del kit de desarrollo de Java Card, en particular con `converter`). Después de guardar el fichero, se debe ejecutar y aceptar las condiciones de la licencia. A continuación, el proceso de instalación pedirá que se le especifique el directorio de instalación, por ejemplo: `C:\java\jdk1.3.1_07`. Luego preguntará si se desea que Internet Explorer o Netscape utilicen la máquina virtual de Java que se está instalando. Lo siguiente es escoger los componentes que se quieran instalar (se recomienda que se instale lo que ya viene seleccionado por defecto). Tras este paso, termina el proceso de instalación de este software.

9.3.1.1.2.2.2 Descomprimir el paquete de desarrollo de Java Card

Para ello, se debe descomprimir el fichero `java_card_kit-2_2_01-win-gl.zip`, con alguna utilidad para descomprimir archivos `.zip` (por ejemplo con Winzip que se puede encontrar en `http://www.winzip.com`). La descompresión se puede hacer en el directorio que se prefiera, por ejemplo en: `C:\pfc\java\java_card_kit-2_2`.

9.3.1.1.2.2.3 Establecer las variables de entorno.

Para llevar a cabo esta etapa, hay que dirigirse a `Inicio ► Ejecutar` y teclear `edit c:\autoexec.bat`. A continuación, las siguientes líneas se deben insertar al final del archivo `autoexec.bat`:

```
SET JAVA_HOME= C:\java\jdk1.3.1_07
SET CLASSPATH=%JAVA_HOME%\src.jar
SET JC_HOME=C:\PFC\JAVA\java_card_kit-2_2
SET PATH=%PATH%;%JC_HOME%\bin;%JAVA_HOME%\bin
```

La variable `JAVA_HOME` indica el directorio donde está instalado el JDK y la variable `JC_HOME` donde se encuentra el kit de desarrollo de Java Card. En cuanto a la variable `PATH`, permite que los programas que se encuentran en un directorio especificado se puedan ejecutar desde cualquier otro directorio. Después de insertar las líneas hay que salvar el archivo `autoexec.bat` y reiniciar el ordenador para que los cambios surjan efecto.

Lo siguiente es comprobar que efectivamente todo funciona correctamente. Para ello dirigirse a `Inicio ► Programas ► MS-DOS` y ejecutar lo siguiente:

```
javac
java
```

Deberían aparecer mensajes de ayuda del compilador y del intérprete de Java.

9.3.1.1.2.2.4 Notas adicionales

Se ha detectado que al ejecutar `build_samples.bat` (que se describe más adelante) el sistema se queda sin espacio de variables de entorno, debido a la gran cantidad de variables entorno que se declaran en dicho archivo. Se ha comprobado que definiendo las variables en el archivo `autoexec.bat`, se consigue retrasar la falta de espacio. Lo mejor es ir viendo los ejemplos uno a uno, tecleando las órdenes a mano.

9.3.1.1.2.3 Procedimiento de instalación en Windows XP

9.3.1.1.2.3.1 Ejecutar el programa de instalación del JDK

Este programa tendrá un nombre parecido al siguiente `j2sdk-1_3_1_07-windows-i586.exe` (se recomienda el uso del jdk1.3, ya que el jdk1.4 puede causar problemas con las herramientas del kit de desarrollo de Java Card, en particular con `converter`). Después de guardar el fichero, se debe ejecutar y aceptar las condiciones de la licencia. A continuación, el proceso de instalación pedirá que se le especifique el directorio de instalación, por ejemplo: `C:\java\jdk1.3.1_07`. Luego preguntará si se desea que Internet Explorer o Netscape utilicen la máquina virtual de Java que se está instalando. Lo siguiente es escoger los componentes que se quieran instalar (se recomienda que se instale lo que ya viene seleccionado por defecto). Tras este paso, termina el proceso de instalación de este software.

9.3.1.1.2.3.2 Descomprimir el paquete de desarrollo de Java Card

Desde el explorador de Windows se puede ver el archivo `java_card_kit-2_2_01-win-gl.zip` como una carpeta con una cremallera (si no se ha instalado anteriormente alguna utilidad para descomprimir). Se debe entrar en esa carpeta y llevar el directorio `java_card_kit-2_2` al directorio que se elija, por ejemplo en: `C:\pfc\java`. La estructura de directorios que se obtendría, sería parecida a esta: `C:\pfc\java\java_card_kit-2_2`.

9.3.1.1.2.3.3 Establecer las variables de entorno.

Para ajustar las variables de entorno se debe desplegar el menú `Inicio ► Configuración ► Panel de control ► Sistema`. En la pestaña de `Opciones avanzadas` hay que pulsar el botón `Variables de entorno`. Como se podrá observar, se pueden incluir las variables de entorno para un usuario (el actual) o para todos los usuarios (si el usuario es un administrador del sistema). Las variables de entorno que se deben crear y sus valores, son los siguientes:

Nombre de la variable	Valor de la variable
`JAVA_HOME`	`C:\pfc\java\jdk1.3.1_07`
`JC_HOME`	`C:\pfc\java\java_card_kit-2_2`
`CLASSPATH`	`%JAVA_HOME%\src.jar`

Donde `JAVA_HOME` y `JC_HOME` son los directorios donde se encuentran el JDK y el entorno de desarrollo de Java Card, respectivamente. Por último, se puede modificar la variable `PATH`, añadiendo lo siguiente a su valor: `;%JC_HOME%\bin;%JAVA_HOME%\bin` . Así se pueden ejecutar los programas que se encuentran en el directorio especificado, desde cualquier otro directorio. Para salir del menú de `Variables de entorno`, se debe pulsar el botón de `Aceptar`.

Para comprobar que todo funciona correctamente hay que dirigirse a `Inicio ► Programas ► Accesorios ► Símbolo del sistema` y ejecutar lo siguiente:

```
javac
java
```

Deberían aparecer mensajes de ayuda del compilador y del intérprete de Java.

9.3.1.1.2.4 Copiar los ficheros de OpenCard Framework

Para ejecutar las demostraciones de Java Card RMI (Remote Method Invocation) se tiene que llevar a cabo este proceso tanto para plataformas Linux (si se consigue la versión 2.2 del kit de desarrollo de Java Card) como Windows. En primer lugar se debe navegar hasta el directorio donde se ha descargado el fichero `BaseOCF.zip` y luego descomprimirlo, con lo que se obtendrá un directorio con nombre `OCF1.2`. La siguiente tarea es introducirse en el directorio `./OCF1.2/lib` (sustituir las "/" por "\" si se está trabajando en plataformas Windows) y copiar los ficheros `base-core.jar y base-opt.jar` en el directorio `$JC_HOME/lib` (para Linux) ó `%JC_HOME%\lib` (para Windows).

9.3.1.1.3 Ficheros instalados

En la siguiente tabla se presenta una relación de archivos y directorios que se encuentran en el directorio `java_card_kit-2_2` instalado anteriormente:

Directorio/Archivo	Descripción
`api_export_files`	Directorio que contiene los ficheros de exportación para los paquetes de la API de Java Card 2.2.
`bin`	Directorio que contiene shell scripts o archivos de procesos por lotes para ejecutar las herramientas y el ejecutable `cref`.
`doc`	En el directorio `doc/en/guides` se encuentran el Java Card 2.2 Development Kit User's Guide y el Java Card 2.2 Application Programming Notes. En el directorio `doc/en/whitepapers` se encuentran las siguientes hojas blancas "white papers": Java Card 2.2 Off-Card Verifier y el Java Card 2.2 RMI Client Application Programming Interface.
`lib`	Este directorio contiene todos los archivos `.jar` de Java, necesarios para las herramientas. A continuación se especifica la utilidad de cada uno de los ficheros que se encuentran en este directorio. El fichero `api.jar` es necesario para escribir los applets y las librerias de los applets de Java Card. El compilador RMIC de Java usa el fichero `javacardframework.jar`, que se encuentra en dicho directorio. Este compilador se usa para generar stubs para las aplicaciones JCRMI (RMI para Java Card). El archivo `apduio.jar` lo utiliza `apdutool`. El `jcwde.jar` lo utiliza el JCWDE (Java Card Workstation Development Environment. El archivo `jcclientsamples.jar` contiene la parte del cliente de los ejemplos de RMI de Java Card. El fichero `jcrmiclientframework.jar` contiene las clases de las API's de RMI de Java Card.
`samples`	Este directorio contiene los applets de ejemplo y los programas de demostración.
`COPYRIGHT_gl` ó `COPYRIGHT_gl.txt` (en Windows)	Contiene el aviso del copyright de Java Card Kit 2.2.
`README.html`	Contiene información general acerca de esa versión de Java Card Kit.
`RELEASENOTES.html`	Contiene información importante acerca de esa versión.
`LICENSE.html`	Contiene el texto de acuerdo.

9.3.1.1.4 Programas de ejemplo y demostraciones

Todos los ejemplos se encuentran en el directorio `samples` (`$JC_HOME/samples`). La siguiente tabla describe el contenido del directorio `samples`:

Directorio/Archivo	Descripción
`classes`	Directorio que contiene las clases de ejemplo.
`build_samples` ó `build_samples.bat`	Es un script o archivo de proceso por lotes para construir los ejemplos automáticamente.
`src`	Directorio que contiene los códigos fuente de los applets de ejemplo que pertenecen a los paquetes `com.sun.javacard.samples.*`.
`src/demo`	Este directorio contiene los scripts de las APDU's y los archivos de salida esperados para las demostraciones. Este directorio también contiene los archivos `jcwde.app`, `opencard.properties`, y los ficheros de procesos por lotes para las demostraciones de RMI de Java Card y Secure RMI de Java Card.
`src_client`	Directorio que contiene programas cliente de los dispositivos de aceptación de tarjetas (CAD), para las demostraciones de RMI de Java Card y Secure RMI de Java Card.

9.3.2 EJEMPLOS Y DEMOSTRACIONES DE JAVA CARD

Esta versión 2.2 de Java Card incluye varios programas de demostración que ilustran el uso de las API's de Java Card y un caso hipotético de instalación después de la fabricación.

9.3.2.1 Las demostraciones

El directorio `demo` (que se encuentra en `java_card_kit-2_2/samples/src/demo`) contiene los programas de demostración que se describen en la siguiente tabla:

Tabla 18: Resumen de las demostraciones inluidas en el Java Card Development Kit

Demostración	Descripción
`demo1`	Ilustra el uso de los paquetes incluidos en la memoria ROM de la tarjeta: `JavaPurse`, `JavaLoyalty`, `Wallet` y `SampleLibrary`.
`demo2`	Esta demostración carga estos paquetes en el JCRE, utilizando el applet instalador: `JavaPurse`, `JavaLoyalty`, `Wallet`, `SampleLibrary`, `RMIDemo` y `SecureRMIDemo`.
`demo3`	Ilustra la segunda vez que se enciende la tarjeta. Utiliza el fichero de estado creado en la demostración `demo2`.
`RMIDemo`	Demuestra el uso de las API's de Java Card Remote Method Invocation (JCRMI). El ejemplo básico usado, es un programa que gestiona un contador de forma remota. Es capaz de decrementarlo, incrementarlo y de devolver el valor de la cuenta. `RMIDemo` utiliza el fichero de estado de la tarjeta, creado en la demostración `demo2`.
`SecureRMIDemo`	Es similar a `RMIDemo`, pero demuestra como aportar seguridad adicional al nivel de transporte. También utiliza el fichero de estado de la tarjeta creado en la demostración `demo2`.
`odDemo1`	Demuestra el borrado de applets y paquetes, y además, el mecanismo de borrado de objetos que borra los objetos no alcanzables.
`odDemo2`	Demuestra el borrado de paquetes y comprueba que la memoria persistente es devuelta al gestor de memoria.
`channelDemo`	Demuestra la utilización de canales lógicos que permite seleccionar múltiples applets al mismo tiempo.

9.3.2.1.1 Archivos del directorio demo

Los archivos que se proveen se describen en la siguiente tabla:

Tabla 19: Archivos incluidos en el directorio `demo`

Archivo	Descripción
`jcwde.app`	Se trata de una lista para el JCWDE de todos los applets (y todos sus AID's) que se descargarán en la máscara (de la memoria ROM) simulada.
`*.scr`	Demostración de ficheros script de la herramienta `apdutool`.
`rmidemo` ó `rmidemo.bat`	Ficheros de procesos por lotes y shell scripts para ejecutar las demostraciones de RMI de Java Card.
`securermidemo` ó `securermidemo.bat`	Ficheros de procesos por lotes y shell scripts para ejecutar las demostraciones de RMI de Java Card.
`*.expected`	Son ficheros para comparar con las salidas del programa `apdutool` (ó el programa cliente para `RMIDemo` y `SecureRMIDemo`) cuando se han ejecutado las demostraciones.
`opencard.`	Contiene los ajustes del OCF para las demostraciones de RMI de Java Card.

`properties`	

9.3.2.2 Preliminares

Todos los programas de demostración están ya construidos. Si se realizan cambios a las demostraciones, las siguientes secciones explican como se pueden reconstruir.

9.3.2.3 Fichero script para construir los ejemplos

El Java Card Kit provee un fichero script para construir los ejemplos. Para comprender que es lo que se está haciendo, es muy interesante echar un vistazo en el interior del archivo script.

El archivo script tiene la siguiente ruta `$JC_HOME/samples/build_samples` (para Linux) ó `%JC_HOME%\samples\build_samples.bat` (para Windows).

9.3.2.3.1 Ejecución del script

La sintaxis del script en la línea de comandos, es la siguiente:

```
build_samples [opciones]
```

Las opciones se recogen en la siguiente tabla:

Tabla 20: Opciones de `build_samples`

Valor de la opción	Descripción
`-clean`	Borra todos los ficheros producidos por el script.
`-help`	Solamente imprime mensajes de ayuda por pantalla.
`[sin opciones]`	Construye todos los applets de ejemplo, clientes de ejemplo, y scripts de demostración.

9.3.2.3.2 Ajuste de las variables de entorno

El script `build_samples` usa las variables de entorno `JC_HOME` y `JAVA_HOME`. Por lo tanto es importante que las variables de entorno estén correctamente ajustadas (ver sección 9.3.1.1.2 Instalación del software y ajustar las variables según el sistema operativo utilizado).

9.3.2.4 Construcción de los applets de ejemplo

Para construir los ejemplos, se debe ejecutar el script sin parámetros:

```
build_samples
```

9.3.2.4.1 Preparación para compilar los applets de ejemplo

Esta sección detalla los pasos que sigue el fichero script, pero también se pueden ejecutar los comandos uno a uno si se prefiere.

En primer lugar se debe crear un directorio llamado `classes` en el directorio `$JC_HOME/samples`, con lo que se obtendría: `$JC_HOME/samples/classes` (`%JC_HOME%\samples\classes` en Windows). En segundo lugar se deben copiar los

directorios de los ficheros de exportación de la API de Java Card 2.2 al directorio creado anteriormente. Es decir, en el directorio `$JC_HOME/samples/classes` se deben encontrar los directorios `java`, `javacard` y `javacardx`, que se encuentran en `$JC_HOME\api_export_files`.

9.3.2.4.2 Compilación de los applets de ejemplo

El próximo paso es compilar los archivos que contienen el código fuente Java de los applets de ejemplo. Para el siguiente ejemplo, hay que situarse en el directorio `samples`, y teclear los siguientes comandos:

Para Linux

```
javac –g –classpath ./classes:../lib/api.jar
src/com/sun/javacard/samples/HelloWorld/*.java
```

Para Windows

```
javac –g –classpath .\classes;..\lib\api.jar
src\com\sun\javacard\samples\HelloWorld\*.java
```

El archivo `api.jar` contiene las API's de Java Card y el directorio `./classes` es necesario para los paquetes que importan otros paquetes de ejemplo.

9.3.2.4.3 Conversión de los ficheros class

En esta sección se explica como convertir los ficheros `class` obtenidos en el paso anterior. Los parámetros que intervienen en la conversión de cada paquete están especificados en un fichero de configuración (son ficheros con extensión `.opt`). Por ejemplo, un archivo de configuración (el que se encuentra en `%JC_HOME%\samples\src\com\sun\javacard\samples\HelloWorld`) tiene esta forma:

```
-out EXP JCA CAP
-exportpath .
-applet 0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x1:0x1 com.sun.
javacard.samples.HelloWorld.HelloWorld
com.sun.javacard.samples.HelloWorld
0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x1 1.0
```

En este ejemplo, el convertidor (`converter`) sacará tres tipos de archivos: el de exportación (`*.exp`), el CAP (`*.cap`) y el JCA (`*.jca`).

La estructura general de uso del comando converter es la siguiente:

```
converter [opciones] <nombre_paquete> <aid_paquete>
<versión_mayor>.<versión_menor>
```

El comando `converter` se trata de un fichero shell script (en Linux) o un fichero de procesos por lotes (en Windows).

9.3.2.4.3.1 Argumentos de la línea de comandos

Los argumentos de la línea de comandos, son los siguientes:

Tabla 21: Argumentos de la línea de comandos de `converter`

Argumento	Descripción
`<nombre_paquete>`	Nombre completo del nombre del paquete a convertir.
`<aid_paquete>`	Este argumento consta de 5 a 16 números decimales, hexadecimales u octales, separados por dos puntos. Cada uno de los números debe ser de un byte de longitud.
`<versión_mayor>. <versión_menor>`	Versión del paquete definida por el usuario.

9.3.2.4.3.2 Opciones de la línea de comandos

Las opciones que permite este comando, son las siguientes:

Tabla 22: Opciones de la línea de comando de `converter`

Argumento	Descripción
`-applet <AID> <nombre_clase>`	Ajusta el AID por defecto del applet y el nombre de la clase que define el applet. Si el applet contiene múltiples clases de applets, se debe especificar esta opción para cada clase.
`-classdir <directorio_raíz_de_la_jerarquía_de_la_clase>`	Ajusta el directorio raíz donde el convertidor buscará las clases. Si no se especifica esta opción, el convertidor usa el directorio del usuario actual como el directorio raíz.
`-d <directorio_raíz_de_la salida>`	Ajusta el directorio raíz de la salida.
`-debug`	Genera el componente opcional de depuración de un fichero CAP. Si también se especifica la opción `–mask`, el fichero `debug.msk` se generará en el directorio de salida. Para generar el componente de depuración, primero se deben compilar los ficheros de las clases con la opción `–g` del compilador de Java (`javac`).
`-exportmap`	Usa el mapeo de testigos a partir del fichero de exportación predefinido del paquete que está siendo convertido. El convertidor buscará el fichero de exportación en el `exportpath`.
`-exportpath <lista_de_directorios>`	Especifica los directorios raíces, en los que el convertidor buscará los ficheros de exportación. El carácter para indicar múltiples caminos depende de la plataforma. Para Windows, el carácter separador es el punto y coma (;). Para Linux, el carácter de separación es el dos puntos (:). Si no se especifica esta opción, el convertidor ajusta el `exportpath` a la variable de entorno `CLASSPATH`.
`-help`	Imprime un mensaje de ayuda por pantalla.
`-i`	Le indica al convertidor que soporte el tipo entero de 32 bits.
`-mask`	Indica que este paquete es para una máscara, por ello las restricciones sobre los métodos nativos se relajan.
`-nobanner`	Suprime todos los mensajes.
`-noverify`	Suprime la verificación de los archivos de entrada y de salida.
`-nowarn`	Le indica al convertidor que no muestre los mensajes de aviso (warnings).
`-out [CAP] [EXP] [JCA]`	Le indica al convertidor que produzca el fichero CAP, y/o el fichero de exportación y/o el fichero JCA. Por defecto (si no se especifica esta opción). el convertidor produce un fichero CAP y un fichero de exportación.

`-v, -verbose`	Permite la salida de mensajes de progreso como “opening file”, “closing file”, y si el paquete requiere el soporte del tipo entero.
`-V, -version`	Imprime por pantalla la versión del convertidor.

9.3.2.4.3.3 Uso de delimitadores en las opciones de la línea de comandos

Si el argumento de opciones de la línea de comandos contiene un símbolo de espacio, se deben usar delimitadores en ese argumento. El delimitador para Linux es la barra / y comillas dobles (“/). Los delimitadores en Windows, son las comillas dobles (“).

En el siguiente ejemplo de la línea de comandos, el convertidor comprobará los ficheros de exportación en `.\ficheros_exportación`, `.\java_card_kit-2_2\api_export_files` y en los directorios actuales:

Para Linux

```
converter –exportpath \”./ficheros_exportación:.:./java_
card_kit-2_2/api_export_files\” MyWallet
0xa0:0x00:0x00:0x00:0x62:0x12:0x34 1.0
```

Para Windows

```
converter –exportpath ”.\ficheros_exportación;.;.\java_
card_kit-2_2\api_export_files” MyWallet
0xa0:0x00:0x00:0x00:0x62:0x12:0x34 1.0
```

9.3.2.4.3.4 Uso del fichero de configuración

En lugar de teclear todos los argumentos y opciones en la línea de comandos, se pueden incluir en un fichero de configuración (es un fichero de texto). Esto es conveniente si se usa frecuentemente el mismo conjunto de argumentos y de opciones.

La sintaxis para especificar un fichero de configuración es la siguiente:

```
converter –config <nombre_fichero_de_configuración>
```

El argumento `<nombre_fichero_de_configuración>` contiene el camino y el nombre del fichero del archivo de configuración.

Tanto para Linux como para Windows, se deben usar los delimitadores de las dobles comillas (“) para las opciones de la línea de comandos que requieran argumentos en el fichero de configuración. Por ejemplo, si las opciones del ejemplo de la línea de comandos (usadas anteriormente) se pusieran en un fichero de configuración, el resultado se parecería a esto:

Para Linux

```
–exportpath ”./ficheros_exportación:.:.\java_ card_kit-
2_2/api_export_files” MyWallet 0xa0:0x00:0x00:0x00:0x62:
0x12:0x34 1.0
```

Para Windows

```
–exportpath ”.\ficheros_exportación;.;.\java_ card_kit-
2_2\api_export_files” MyWallet 0xa0:0x00:0x00:0x00:0x62:
0x12:0x34 1.0
```

9.3.2.5 Ejecución de las demostraciones

Las siguientes secciones describen las demostraciones de Java Card y como ejecutarlas. Se supone que las demostraciones se encuentran en el directorio `$JC_HOME/samples/src/demo` (para Linux) ó en `%JC_HOME%\samples\src\demo` (para Windows).

Una demostración puede usar una imagen de la EEPROM creada por otra demostración. La opción `-o` del comando `cref` permite salvar la imagen de la EEPROM en un fichero, después de una sesión simulada de la tarjeta. La opción –i restaura la imagen a partir del fichero que la guarda, para comenzar una nueva sesión simulada con la tarjeta.

9.3.2.5.1 Demostración `demo1`

La demostración demo1 muestra la simulación de transacciones (de crédito y de débito) entre los applets `JavaPurse`, `JavaLoyalty` y `Wallet`. La demostración comienza encendiendo la tarjeta Java y creando los applets `JavaPurse`, `JavaLoyalty` y `Wallet`.

El applet `JavaPurse` (en `%JC_HOME%\samples\src\com\sun\javacard\samples\JavaPurse`) demuestra una aplicación simple de dinero electrónico. El applet se selecciona e inicializa con varios parámetros tales como la identidad del monedero (PurseID), la fecha de expiración de la tarjeta, el PIN principal y el de usuario, el balance máximo y la máxima cantidad que puede intervenir en una transacción. Las operaciones de la transacción realizan los créditos o los débitos en el monedero electrónico. Si hay un applet loyalty (que contabiliza puntos por usar el monedero electrónico) configurado, asignado por el CAD que lleva a cabo la transacción, el applet `JavaPurse` se comunica con él para conceder puntos de confianza. En este caso, `JavaLoyalty` es el applet loyalty provisto.

Se simulan un número de sesiones de transacciones donde los importes se cargan en la cuenta de la tarjeta. En una sesión adicional, se intentan realizar transacciones con errores intencionados para probar las características de seguridad de la tarjeta.

El applet `JavaLoyalty` está diseñado para interaccionar con el applet `JavaPurse`, y para demostrar el uso de interfaces compartidas. La interfaz compartida `JavaLoyaltyInterface` se define en un paquete de librería separado que se encuentra en `$JC_HOME/samples/src/com/sun/javacard/samples/SampleLibrary` (para Linux) `%JC_HOME%\samples\sr\com\sun\javacard\samples\SampleLibrary` (para Windows).

El applet `JavaLoyalty` se registra con el applet cuando se ejecuta una APDU de comando de actualización de parámetro, con la etiqueta adecuada del parámetro, y cuando la parte AID del parámetro corresponda con el AID del applet `JavaLoyalty`. Este applet contiene un método llamado `grantPoints`. Este método implementa la interacción principal con el cliente. Cuando los dos primeros bytes del identificador del CAD, usados en una solicitud de una transacción del applet `JavaPurse`, correspondan con los dos bytes del identificador del CAD en la APDU de comando de actualización

del parámetro, se llama al método `grantPoints` que implementa la interfaz `JavaLoyaltyInterface`.

El applet `JavaLoyalty` mantiene el balance de los puntos. El applet contiene métodos para cargar puntos en la cuenta y para conseguir y ajustar el balance.

El applet `Wallet` demuestra una aplicación simplificada de tarjeta monedero. Mantiene el balance, y emplea alguna de las características de las API´s de Java Card, tales como el uso de un PIN para controlar el acceso al applet.

9.3.2.5.1.1 Ejecución de `demo1`

La `demo1` se ejecuta en el JCWDE. Para ello hay que situarse en el directorio `%JC_HOME%\samples\src\demo` (para Windows) y teclear lo siguiente:

```
jcwde jcwde.app
```

En otra ventana o terminal de comandos, hay que ejecutar la herramienta `apdutool` (situarse primero en el mismo directorio especificado anteriormente), usando el siguiente comando:

```
apdutool –nobanner –noatr demo1.scr > demo1.scr.jcwde.out
```

Si la ejecución sale bien, el archivo `demo1.scr.jcwde.out`, es idéntico al archivo `demo1.scr.expected.out`. El resultado es un fichero (que se encuentra en el mismo directorio donde se ejecutó `jcwde`) de trazas que muestra las APDU's intercambiadas entre la tarjeta y el CAD. En él se distinguen los campos y los valores de dichos campos que forman las APDU's (para más información ver la sección 3.2.4.3 El protocolo APDU, que se encuentra en la página 31).

9.3.2.5.2 Demostración `demo2`

La demostración `demo2` ilustra la carga de los paquetes de Java Card en el interior de la tarjeta. Esta demostración contiene el applet instalador en la imagen de la memoria EEPROM. Después del encendido de la tarjeta, se descargan los paquetes `SampleLibrary`, `JavaPurse`, `JavaLoyalty`, `Wallet`, `RMIDemo` y `SecureRMIDemo`. Finalmente se apaga la tarjeta.

9.3.2.5.2.1 Ejecución de `demo2`

La demostración `demo2` se ejecuta en el C-JCRE (implementación del entorno de ejecución de Java Card, escrita en lenguaje C y que se incluye en el kit de desarrollo de Java Card) debido a que el JCWDE (Java Card Workstation Development Environment) no soporta la descarga de los ficheros CAP. Para ejecutar la demostración se usa la herramienta `cref` (desde el mismo directorio usado en la demostración `demo1`):

```
cref –o demoee
```

En otra ventana o terminal de comandos, hay que ejecutar `apdutool`, usando la siguiente orden:

```
apdutool –nobanner –noatr demo2.scr > demo2.scr.cref.out
```

Si todo sale bien, el registro de la herramienta `apdutool`, que se encuentra en el archivo `demo2.scr.cref.out` , debería ser idéntico al archivo `demo2.scr.expected.out`. Después de que `cref` complete su ejecución, se guarda una imagen de la memoria EEPROM en el fichero `demoee` (gracias a la opción `–o` de `cref`). En este fichero se puede observar con un editor de ficheros en hexadecimal, como el comienzo del archivo contiene números con valores distintos (correspondientes a la memoria que ha sido usada) mientras que el resto se encuentra con valor 0xFF (memoria no usada).

9.3.2.5.3 Demostración `demo3`

La demostración `demo3` ilustra la capacidad de Java Card de salvar su estado entre sesiones. Después de la ejecución de `demo2`, se salvó el estado de la tarjeta en el fichero `demoee` (más bien la imagen de la memoria EEPROM de la tarjeta). Este estado de la tarjeta se debe usar como el estado inicial para ejecutar la demostración `demo3`.

9.3.2.5.3.1 Ejecución de `demo3`

La demostración `demo3` se debe ejecutar después de la demostración `demo2`. `demo3` se ejecuta en el C-JCRE porque se debe restaurar el estado de la máquina virtual de Java después de la ejecución inicial. Para ello hay que ejecutar `cref` usando la siguiente orden:

```
cref –i demoee
```

La herramienta `cref` restaura la imagen de la EEPROM a partir del fichero `demoee`, gracias a la opción `–i`. En una ventana separada, hay que situarse en el directorio `demo` y ejecutar la herramienta `apdutool` de la siguiente forma:

```
apdutool –nobanner –noatr demo3.scr > demo3.scr.cref.out
```

Si la ejecución ha ido bien, el fichero `log.demo3.scr.cref.out`, debería ser idéntico al fichero `demo3.scr.expected.out`.

9.3.2.5.4 Demostración RMI de Java Card

Cada aplicación de RMI de Java Card se compone de dos partes: un applet de la tarjeta y un programa cliente que se comunica con él. En este caso, el applet `RMIDemo` se instala en la imagen de la EEPROM obtenida de la ejecución de `demo2`.

El soporte del cliente usa el OCF Framework, versión 1.2 (archivos: `base-core.jar` y `base-opt.jar`), para comunicarse con el C-JCRE. Los ajustes del OCF se guardan en el fichero `opencard.properties`.

Hay que señalar que OCF busca este fichero en las siguientes localizaciones:

```
[java.home]/lib/opencard.properties
[user.home]/.opencard.properties
[user.dir]/opencard.properties
[user.dir]/.opencard.properties
```

Donde `[java.home]` corresponde al directorio `%JAVA_HOME%\jre`.

El applet `RMIDemo` utiliza el applet `PurseApplet` de la tarjeta, la interfaz `Purse` y su implementación `PurseImpl` (todos ellos se encuentran en `%JC_HOME%\samples\src\com\sun\javacard\samples\RMIDemo`). La parte cliente del programa, `PurseClient`, se encuentra en el paquete `com.sun.javacard.clientsamples.purseclient` (en el directorio `%JC_HOME%\samples\src_client\com\sun\javacard\clientsamples\purseclient`). La interfaz `Purse` describe la funcionalidad soportada: métodos para conseguir el balance de la cuenta, retirar dinero de la cuenta, y conseguir y cambiar el número de la cuenta. La interfaz también define las constantes utilizadas para informar de los errores. La clase `PurseImpl` implementa la interfaz `Purse`.

El applet `PurseApplet` de la tarjeta, crea y registra instancias del despachador y del servicio de RMI de Java Card.

El programa cliente, `PurseClient`, representa un cliente simple de RMI de Java Card y usa el Open Card Framework (OCF) como el soporte del cliente. El programa inicializa el OCF, crea la instancia `Connect` de RMI de Java Card, y selecciona el applet de Java Card (en este caso, `PurseApplet`). Entonces el programa toma una referencia inicial del applet `PurseApplet` (la referencia a una instancia de `PurseImpl`) y le hace un cast al tipo de la interfaz `Purse`. Esto permite que `PurseImpl` sea tratado como un objeto local. Entonces el programa puede probar la tarjeta realizando operaciones de crédito y de débito con cantidades diferentes, y ajustando y obteniendo el número de cuenta. El programa también demuestra el tratamiento de errores intentando, intencionadamente, asignar un número de cuenta con un tamaño incorrecto. Esto causará una `UserException` que se lanza con el código de causa apropiado.

La parte cliente de la demostración `RMIDemo` se puede ejecutar sin parámetros o con el parámetro `–i`:

- Si la demostración se ejecuta sin parámetros, las referencias remotas se identifican usando el nombre de la clase del objeto remoto.
- Si la demostración se ejecuta con el parámetro –i, las referencias remotas se identifican utilizando la lista de interfaces remotas implementadas por el objeto remoto.

9.3.2.5.4.1 Ejecución de la demostración de RMI de Java Card

`RMIDemo` solo se puede ejecutar una vez que la demostración `demo2` se ha completado satisfactoriamente. La demostración `RMIDemo` solo se ejecuta en el C-JCRE debido a que el JCWDE no soporta RMI de Java Card. Para hacer funcionar la demostración se tiene que ejecutar la herramienta `cref`:

```
cref –i demoee
```

Después hay que ejecutar el cliente de RMI de Java Card en otra ventana de comandos, tecleando cualquiera de estas dos órdenes:

```
rmidemo > rmidemo.out
rmidemo –i > rmidemo.out
```

Si la ejecución se realiza de forma satisfactoria, el resultado que se encuentra en el fichero `rmidemo.out` será el mismo que se halla en el fichero `rmidemo.scr.expected.out`.

9.3.2.5.5 Demostración de Secure RMI de Java Card

Se puede pensar en la demostración `SecureRMIDemo` como una versión de `RMIDemo` con un servicio de seguridad añadido. La demostración `SecureRMIDemo` utiliza el applet `SecurePurseApplet`, la interfaz `Purse` y su implementación `SecurePurseImpl`. Estas clases residen en el paquete `com.sun.javacard.samples.SecureRMIDemo` (en el directorio `%JC_HOME%\samples\src\com\sun\javacard\samples\SecureRMIDemo`). La demostración también utiliza el programa cliente `SecurePurseClient` y la clase que accede a la tarjeta `SecureOCFCardAccessor`. Estas clases residen en el paquete `com.sun.javacard.clientsamples.securepurseclient`. (en el directorio `%JC_HOME%\samples\src_client\com\sun\javacard\clientsamples\securepurseclient`).

La interfaz `Purse` es similar a la interfaz usada en el caso no seguro, sin embargo, hay una constante extra: `REQUEST_DENIED`. Esta constante se usa para informar de situaciones donde el cliente intenta invocar un método al que no tiene permitido el acceso.

La clase `MySecurityService` es un servicio de seguridad que es responsable de asegurar la integridad de los datos, verificando la suma de comprobación (checksum) de los comandos que recibe y de añadir la suma de comprobación a las respuestas que transmite. El programa también solicita que el cliente se autentifique como la aplicación principal o el titular de la tarjeta, mediante el envío de un PIN de dos bytes.

La implementación de `Purse`, `SecurePurseImpl`, es similar al caso no seguro. Sin embargo, al comienzo de la llamada al método, hay una llamada a un servicio de seguridad que asegura que las reglas comerciales se satisfacen y que los datos no están corrompidos.

El applet `SecurePurseApplet` es similar al del caso no seguro, pero también crea y registra una instancia de `MySecurityService`.

El programa cliente, `SecurePurseClient`, es similar al del caso no seguro, pero en lugar de tener una clase genérica que acceda a la tarjeta, usa su propia implementación: `SecureOCFCardAccessor`, que lleva a cabo un preprocesado y un postprocesado de los datos y soporta el comando adicional `authenticateUser`.

`SecurePurseClient` también requiere la verificación del usuario. Se le debe pasar un PIN al applet de la tarjeta mediante la llamada al método `authenticateUser` de la clase `SecureOCFCardAccessor`.

Cuando se llama a `authenticateUser`, `SecureOCFCardAccessor` prepara y envía el siguiente comando:

Tabla 23: Comando preparado y enviado por `SecureOCFCardAccessor`

CLA_AUTH	INS_AUTH	P1	P2	Lc	PIN (dos bytes)	
0x80	0x39	0	0	2	xx	xx

En la tarjeta, `MySecurityService` procesa el comando. Si el PIN es correcto, entonces se ajustan las banderas apropiadas en el servicio de seguridad y se le devuelve al cliente una respuesta de confirmación. Una vez que se pasa el proceso de autenticación, el programa cliente recibe el balance, carga el importe, y recibe el balance de nuevo. El programa demuestra como tratar los errores cuando el cliente intenta cargar el importe en la cuenta. Esto causa que el programa lance una `UserException` con el código `REQUEST_DENIED`.

Al igual que en la demostración `RMIDemo`, la parte cliente de `SecureRMIDemo` puede ejecutarse sin parámetros o con el parámetro `–i`:

- Si la demostración se ejecuta sin parámetro, las referencias remotas se identifican utilizando el nombre de la clase del objeto remoto.
- Si la demostración se ejecuta con el parámetro –i, las referencias remotas se identifican utilizando la lista de interfaces remotas implementadas por el objeto remoto.

9.3.2.5.5.1 Ejecución de la demostración de Secure RMI de Java Card

El applet `SecureRMI` de demostración, está instalado en la imagen de la EEPROM cuando se ejecuta la demostración `demo2`. `SecureRMIDemo` sólo se ejecuta en el C-JCRE debido a que JCWDE no soporta RMI. Para ver la demostración, hay que ejecutar `cref` en el directorio `demo` usando la siguiente orden:

```
cref –i demoee
```

En otra ventana separada se debe ejecutar el programa cliente de Secure RMI de Java Card con cualquiera de estos dos comandos:

```
securermidemo > securermidemo.out
securermidemo –i > securermidemo.out
```

Si la ejecución se completa satisfactoriamente, el fichero `securermidemo.out` tendrá el mismo contenido que el fichero `securermidemo.scr.expected.out`.

9.3.2.5.6 Demostración `odDemo1`

Esta demostración ilustra el mecanismo de borrado de objetos, de borrado de applets y el de borrado de paquetes. La demostración `odDemo1` tiene tres partes:

- `odDemo1-1.scr` enseña el mecanismo de borrado de objetos y verifica que la memoria para objetos referenciados desde memoria transitoria del tipo `CLEAR_ON_DESELECT`, se ha recuperado después de que el applet se haya deseleccionado.

  `odDemo1-1.scr` no depende de ninguna otra demostración. El estado final de la memoria se debe salvar en un fichero, para que lo use la demostración `odDemo1-2.scr`.

- `odDemo1-2.scr` demuestra el mecanismo de borrado de objetos y verifica que la memoria para objetos referenciados desde memoria transitoria del tipo `CLEAR_ON_RESET`, se ha recuperado después del reset de la tarjeta.

  La demostración `odDemo1-2.scr` se debe ejecutar después de `odDemo1-1.scr` debido a que el estado inicial debe ser el mismo que el estado final cuando se ejecuta `odDemo1-1.scr`. Después de ejecutar `odDemo1.2.scr`, el estado final se debe salvar en un fichero para que `odDemo1-3.scr` pueda usarlo.

- `odDemo1-3.scr` lleva a cabo el borrado de objetos y el borrado de paquetes, y verifica que toda la memoria transitoria del tipo `CLEAR_ON_RESET` y `CLEAR_ON_DESELECT` se devuelva al gestor de memoria.

  La demostración `odDemo1-3.scr` se debe ejecutar después de `odDemo1-2.scr` debido a que el estado inicial necesario, es el mismo que el estado final que se obtiene después de ejecutar `odDemo1-2.scr`.

9.3.2.5.6.1 Ejecución de `odDemo1`

`odDemo1` solo se ejecuta en el C-JCRE. Esto es debido a que el JCWDE no soporta el mecanismo de borrado de objetos, de applets ni de paquetes. Para ver esta demostración, hay que ejecutar `cref` de la siguiente forma en una ventana de comandos:

```
cref –o crefState
```

En otra ventana, se debe ejecutar `APDUTool` usando esta orden:

```
apdutool –nobanner –noatr odDemo1-1.scr >
odDemo1-1.scr.cref.out
```

Si la ejecución sale bien, el registro de `APDUTool`, `odDemo1-1.scr.cref.out` debería ser igual al fichero `odDemo1-1.scr.expected.out`.

A continuación, hay que volver a ejecutar `cref` tecleando lo siguiente:

```
cref –i crefState –o crefState
```

En la otra ventana, se debe ejecutar la `APDUTool` usando este comando:

```
apdutool –nobanner –noatr odDemo1-2.scr >
odDemo1-2.scr.cref.out
```

Si la ejecución sale bien, el registro de `APDUTool`, `odDemo1-2.scr.cref.out` debería ser igual al fichero `odDemo1-2.scr.expected.out`.

Y para `odDemo1-3.scr`, hay que volver a ejecutar `cref` introduciendo la siguiente orden:

```
cref –i crefState
```

Y a continuación ejecutar la herramienta `APDUTool` de esta forma:

```
apdutool –nobanner –noatr odDemo1-3.scr >
odDemo1-3.scr.cref.out
```

Si la ejecución se completa satisfactoriamente, el registro de `APDUTool`, `odDemo1-3.scr.cref.out` debería ser igual al fichero `odDemo1-3.scr.expected.out`.

9.3.2.5.7 Demostración odDemo2

`odDemo2` ilustra el borrado de paquetes y comprueba que el gestor de memoria ha recuperado la memoria persistente. Esta demostración tiene un script llamado `odDemo2.scr`. La ejecución de `odDemo1` no es requisito para ejecutar `odDemo2`.

9.3.2.5.7.1 Ejecución de odDemo2

`odDemo2` solo se ejecuta en el C-JCRE ya que el JCWDE no soporta el mecanismo de borrado de objetos, de applets ni de paquetes. Para llevar a cabo la ejecución, hay que ejecutar en una ventana la herramienta `cref`:

```
cref
```

En otra ventana, se debe lanzar la `APDUTool` con la siguiente orden:

```
apdutool –nobanner –noatr odDemo2.scr > odDemo2.scr.cref.out
```

Si la ejecución sale bien, el registro de `APDUTool`, `odDemo2.scr.cref.out` debería ser igual al fichero `odDemo2.scr.expected.out.`

9.3.2.5.8 Demostración de uso de canales lógicos

Esta demostración muestra el comportamiento de los canales lógicos en la plataforma Java Card. En este ejemplo, dos applets, que interaccionan el uno con el otro, pueden estar seleccionados al mismo tiempo.

La demostración de canales lógicos simula el comportamiento de un dispositivo inalámbrico conectado a un servicio de red. Un gestor de red rastrea si el dispositivo está conectado al servicio y si la conexión es local o remota. Mientras está conectado, se va descontando la cuenta de usuario en unidades de tiempo; la tasa de descuento se basa en si la conexión es local o remota. La demostración emplea dos applets para simular esta situación: el applet `ConnectionManager` que gestiona la conexión mientras que el applet `AccountAccessor` gestiona la cuenta.

Cuando el usuario enciende el dispositivo, el applet `ConnectionManager` se selecciona. En cada unidad de tiempo, el terminal envía a la tarjeta un mensaje que contiene el código del área. Cuando el usuario quiere usar el servicio, se selecciona el applet `AccountAccessor` en otro canal lógico para que el terminal pueda solicitar el balance de la cuenta. Entonces, la conexión ya está establecida. El applet `ConnectionManager` rastrea el estado de la conexión y basándose en el valor variable del código de área, determina si la conexión es local o remota. El applet `AccountAccessor` usa esta información para descontar la cuenta con la tasa apropiada. La conexión se corta cuando el usuario completa la llamada o cuando la cuenta se agota.

9.3.2.5.8.1 Ejecución de la demostración de uso de canales lógicos

La demostración del uso de los canales lógicos solo se ejecuta en el C-JCRE debido a que el JCWDE no es capaz de soportar la descarga de ficheros CAP. Para ejecutar la demostración hay que ejecutar `cref` de la siguiente forma:

```
cref
```

En otra ventana, se debe ejecutar la herramienta `APDUTool` usando esta orden:

```
apdutool –nobanner –noatr channelDemo.scr >
channelDemo.scr.cref.out
```

Si la ejecución sale bien, el registro de `APDUTool`, `channelDemo.scr.cref.out` debería ser igual al fichero `channelDemo.scr.expected.out`.

9.3.2.5.9 Ejemplos del Java Card Kit 2.1.1 para Linux

En esta versión de Java Card no hay un fichero script que compile y lleve a cabo las demostraciones de forma automática. Por ello hay que hacerlo todo paso a paso. Se pueden compilar los ejemplos que se encuentran en el directorio `samples` del entorno de desarrollo. Antes de compilar hay que asegurarse de que no se va a hacer en una sesión como superusuario y de que se tienen los permisos adecuados (el root debe habilitarlo todo) en los directorios que se encuentran en `/usr/java/`. Si por ejemplo se desea compilar el applet `Wallet`, hay que teclear lo siguiente desde el directorio `samples`:

```
javac –g -classpath $CLASSPATH com/sun/javacard/samples
/wallet/Wallet.java
```

Si lo que se quiere es simular alguno de los escenarios propuestos (como `demo1`) en el directorio `demo` del kit de desarrollo, se debe realizar lo que se explica a continuación. Descomprimir el fichero `comm.jar` en el directorio `demo` (recuerdar lo de los permisos) y desde dicho directorio teclear lo siguiente en una sesión de usuario:

```
jcwde -p 9025 jcwde.app
```

Desde otra sesión o consola de usuario teclear lo siguiente desde el directorio `demo`:

```
apdutool demo1.scr > demo1.scr.out
```

Y comprobar que `demo1.scr.out` es igual que `demo1.scr.expected.out`.

9.3.2.6 Estudio de un applet de Java Card

En esta sección se estudiará la estructura de un applet de Java Card y como se simula su comportamiento. En particular se estudiará el applet `Wallet.java`, que se encuentra en `C:\pfc\java\java_card_kit-2_2\samples\src\com\sun\javacard\samples\wallet`.

9.3.2.6.1 Código fuente

A continuación se presentará el código fuente de `Wallet.java` y los comentarios introducidos irán explicando la estructura y la funcionalidad del applet:

```
/*
 * Copyright © 2002 Sun Microsystems, Inc. All rights reserved.
 * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license
 * terms.
 */

//
// Workfile:@(#)Wallet.java     1.8
// Version:1.8
// Original author: Zhiqun Chen
//

package com.sun.javacard.samples.wallet;

//Importacion de los paquetes javacardx.framework
import javacard.framework.*;

//Todo applet de Java Card debe extender la clase base Applet
public class Wallet extends Applet {

 /* Declaracion de constantes que hacen más legible el codigo */

  // Codigo del byte CLA de la cabecera de una APDU de
  // comando y que identifica a la clase Wallet
  final static byte Wallet_CLA =(byte)0x80;

  // Codigos del byte INS de la cabecera de una APDU de comando
  // que indentifica las operaciones que puede realizar este
  // applet
  final static byte VERIFY = (byte) 0x20;
  final static byte CREDIT = (byte) 0x30;
  final static byte DEBIT = (byte) 0x40;
  final static byte GET_BALANCE = (byte) 0x50;

  // Definicion del balance máximo que puede guardar la tarjeta,
  // en este caso es de 32767 unidades monetarias
  final static short MAX_BALANCE = 0x7FFF;
  // Cantidad máxima de una transacción
  final static byte MAX_TRANSACTION_AMOUNT = 127;

  // Numero maximo de intentos incorrectos permitidos, antes de
  // que el PIN se bloquee
  final static byte PIN_TRY_LIMIT =(byte)0x03;
  // Tamaño maximo del PIN
  final static byte MAX_PIN_SIZE =(byte)0x08;

  // Palabra de estado que indica en una APDU de respuesta que
  // la verificacion del PIN ha fallado
  final static short SW_VERIFICATION_FAILED = 0x6300;
  // Palabra de estado que indica en una APDU de respuesta que
  // se solicita la validación del PIN en el caso de una
  // transaccion de credito o de debito
  final static short SW_PIN_VERIFICATION_REQUIRED = 0x6301;
  // Palabra de estado que indica en una APDU de respuesta que
```

```
// la cantidad especificada en la transaccion es incorrecta,
// es decir, cantidad > MAX_TRANSACTION_AMOUNT o cantidad < 0
final static short SW_INVALID_TRANSACTION_AMOUNT = 0x6A83;
// Palabra de estado que indica en una APDU de respuesta que
// el balance excede el máximo permitido en la tarjeta
final static short SW_EXCEED_MAXIMUM_BALANCE = 0x6A84;
// Palabra de estado que indica en una APDU de respuesta que
// el balance se hace negativo
final static short SW_NEGATIVE_BALANCE = 0x6A85;

/* Declaracion de variables */
OwnerPIN pin;
short balance;

// Constructor
private Wallet (byte[] bArray,short bOffset,byte bLength){

  // Es un buen hábito de programación el declarar en el
  // constructor todas las instancias de variables que van a
  // hacer falta durante toda la vida del applet.

  // Declaracion de la instancia del PIN, caracterizada por el
  // numero maximo de intentos erroneos y el tamaño del PIN
  pin = new OwnerPIN(PIN_TRY_LIMIT, MAX_PIN_SIZE);

  // Longitud del identificador AID de la instancia del applet
  // Wallet y que se encuentra en bArray a partir de la
  // posicion bOffset. bArray contiene los parametros de
  // instalacion
  byte iLen = bArray[bOffset];
  // Incrementos de la variable bOffset para acceder al
  // al valor de inicialización del PIN
  bOffset = (short) (bOffset+iLen+1);
  byte cLen = bArray[bOffset]; // Informacion de longitud
  bOffset = (short) (bOffset+cLen+1);
  byte aLen = bArray[bOffset]; // Longitud del valor del PIN

  // Los parametros de instalacion contienen el valor de
  // inicialización del PIN
  pin.update(bArray, (short)(bOffset+1), aLen);

  // Se llama a este metodo register para que el applet tome
  //el AID que contenga el fichero CAP y no el que viene en
  // los parametros de instalacion
  register();
} // Fin del constructor

// Metodo install
public static void install(byte[] bArray, short bOffset, byte
bLength){
  // Creacion de la instancia de un applet Wallet
  new Wallet(bArray, bOffset, bLength);
} // Fin del metodo install

// Metodo select
public boolean select() {
```

```
    // El applet no se selecciona si el objeto pin esta
    // bloqueado
    if ( pin.getTriesRemaining() == 0 )
       return false;

    // El applet permite su seleccion si el pin no esta
    // bloqueado
    return true;
  }// Fin del metodo select

  // Metodo deslect
  public void deselect() {
    // Resetea la bandera de validacion, es decir, la proxima
    // vez que se quiera seleccionar el applet hay que volver a
    // introducir el PIN correcto, que ya se introdujo en esta
    // selección del applet
    pin.reset();
  } // Fin del metodo deselect

  // Metodo process
  public void process(APDU apdu) {

    // El objeto de la clase APDU contiene un array de bytes
    // (buffer) que se usa para transferir los datos y las
    // cabeceras de las APDU's entrantes y salientes, entre la
    // tarjeta y el CAD

    // En este punto, en el buffer APDU solo estan disponibles
    // los bytes de la cabecera [CLA, INS, P1, P2, P3]
    // La interfaz javacard.framework.ISO7816 declara constantes
    // que indican la posicion donde se encuentran los campos de
    // la cabecera y hacen mas legible el codigo

    // Se guarda el buffer APDU en un buffer
    byte[] buffer = apdu.getBuffer();

    // A continuacion se comprueba si la APDU SELECT de comando
    // va dirigida a este applet
    // Se obtiene el valor del byte CLA
    buffer[ISO7816.OFFSET_CLA] =
    (byte)(buffer[ISO7816.OFFSET_CLA] & (byte)0xFC);
    if ((buffer[ISO7816.OFFSET_CLA] == 0) &&
       (buffer[ISO7816.OFFSET_INS] == (byte)(0xA4)) )
      return;
    // Si la clase especificada en el byte CLA no coincide con
    // la del applet (Wallet_CLA), lanza una excepcion que
    // indica que no se soporta esa clase
    if (buffer[ISO7816.OFFSET_CLA] != Wallet_CLA)
       ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

    // Ya se ha comprobado si la clase es correcta
    // El byte INS especifica la operacion que tiene que
    // realizar el applet
    switch (buffer[ISO7816.OFFSET_INS]) {
      case GET_BALANCE:   getBalance(apdu);
                          return;
```

```
    case DEBIT:         debit(apdu);
                        return;
    case CREDIT:        credit(apdu);
                        return;
    case VERIFY:        verify(apdu);
                        return;
    default:       ISOException.throwIt
                  (ISO7816.SW_INS_NOT_SUPPORTED);
  }
} // Fin del metodo process

// Metodo credit
private void credit(APDU apdu) {
  // Lo primero es comprobar que el PIN introducido
  // es correcto
  if ( ! pin.isValidated() ) ISOException.throwIt(
    SW_PIN_VERIFICATION_REQUIRED);

  // Se guarda el buffer APDU en un buffer
  byte[] buffer = apdu.getBuffer();

  // El byte Lc da el numero de bytes del campo de datos
  // de la APDU de comando
  byte numBytes = buffer[ISO7816.OFFSET_LC];

  // Indica que esta APDU tiene datos entrantes y que los
  // datos recibidos comienzan a partir del offset
  // ISO7816.OFFSET_CDATA (después de los 5 bytes de la
  // cabecera)
  byte byteRead = (byte)(apdu.setIncomingAndReceive());

  // Se lanza una excepcion si el numero de bytes recibidos de
  // datos no coincide con el numero indicado en Lc
  if ( ( numBytes != 1 ) || (byteRead != 1) )
    ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

  // Leyendo en el buffer se obtiene la cantidad del credito
  byte creditAmount = buffer[ISO7816.OFFSET_CDATA];

  // Se comprueba que la cantidad del credito no supera la
  // maxima permitida o es menor que cero
  if ( ( creditAmount > MAX_TRANSACTION_AMOUNT)
    || ( creditAmount < 0 ) )
    ISOException.throwIt (SW_INVALID_TRANSACTION_AMOUNT);

  // Ademas la suma del credito mas la cantidad que hay en la
  // tarjeta no debe superar el balance maximo permitido en la
  // tarjeta
  if ( (short)( balance + creditAmount) > MAX_BALANCE )
    ISOException.throwIt(SW_EXCEED_MAXIMUM_BALANCE);

  // Despues de realizar las comprobaciones, se suma el
  // credito al balance que habia en la tarjeta
  balance = (short)(balance + creditAmount);

} // Fin del metodo credit
```

```
// Metodo debit
private void debit(APDU apdu) {
  // Lo primero es comprobar que el PIN introducido
  // es correcto
  if ( ! pin.isValidated() )
    ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);

  // Se guarda el buffer APDU en un buffer
  byte[] buffer = apdu.getBuffer();

  // El byte Lc da el numero de bytes del campo de datos
  // de la APDU de comando
  byte numBytes = (byte)(buffer[ISO7816.OFFSET_LC]);

  // A continuacion se leen los datos que lleva la APDU
  byte byteRead = (byte)(apdu.setIncomingAndReceive());

  // El numero de bytes de datos leidos de la APDU y el campo
  // Lc de la APDU deben tener el valor 1, ya que la cantidad
  // que participa en esta transaccion de debito cabe en un
  // byte
  if ( ( numBytes != 1 ) || (byteRead != 1) )
    ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

  // Se recupera la cantidad a deber
  byte debitAmount = buffer[ISO7816.OFFSET_CDATA];

  // Se comprueba dicha cantidad, que tiene que ser menor que
  // la cantidad maxima permitida y mayor que cero
  if ( ( debitAmount > MAX_TRANSACTION_AMOUNT)
    ||  ( debitAmount < 0 ) )
    ISOException.throwIt(SW_INVALID_TRANSACTION_AMOUNT);

  // Si la cantidad a deber es correcta, se comprueba que el
  // balance resultante de la tarjeta no sea menor que cero
  // (la tarjeta no puede estar en numeros rojos)
  if ( (short)( balance - debitAmount ) < (short)0 )
    ISOException.throwIt(SW_NEGATIVE_BALANCE);

  // Si las comprobaciones anteriores son correctas, se
  // actualiza el valor del balance de la tarjeta
  balance = (short) (balance - debitAmount);
} // Fin del metodo debit

// Metodo getBalance
private void getBalance(APDU apdu) {
  // A diferencia de los metodos anteriores, no hace falta
  // comprobar el estado del objeto de la clase PIN para
  // consultar el balance de la tarjeta
  // Se guarda el buffer APDU en un buffer
  byte[] buffer = apdu.getBuffer();

  // El metodo setOutgoing ajusta el modo de transferencia de
  // la tarjeta a transmision. El metodo devuelve el número de
  // bytes de datos de respuesta (Le) que el host espera como
```

```
    // respuesta a la APDU de comando
    short le = apdu.setOutgoing();

    // El numero de bytes de datos de respuesta debe ser mayor o
    // igual a dos
    if ( le < 2 ) ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

    // setOutgoingLength informa al host del numero real de
    // bytes de respuesta
    apdu.setOutgoingLength((byte)2);

    // Se trasladan los datos del balance al buffer APDU
    buffer[0] = (byte)(balance >> 8);
    buffer[1] = (byte)(balance & 0xFF);

    // Envio de los dos bytes del balance a partir del offset
    // 0 del buffer APDU
    apdu.sendBytes((short)0, (short)2);

  } // Fin del metodo getBalance

  // Metodo verify
  private void verify(APDU apdu) {
    // Se guarda el buffer APDU en un buffer
    byte[] buffer = apdu.getBuffer();

    // Se recuperan los datos del PIN, para validarlo
    byte byteRead = (byte)(apdu.setIncomingAndReceive());

    // Comprobacion del PIN
    // Los datos del PIN se leen en el buffer APDU a partir del
    // offset ISO7816.OFFSET_CDATA
    // La longitud de los datos del PIN viene dada por byteRead
    if ( pin.check(buffer, ISO7816.OFFSET_CDATA,
      byteRead) == false )
      ISOException.throwIt(SW_VERIFICATION_FAILED);
  } // Fin del metodo validate
} // Fin de la clase Wallet
```

9.3.2.6.2 Compilación del applet

El siguiente paso es compilar el ejemplo. Antes de empezar y por comodidad, se puede incluir el archivo `%JC_HOME%\lib\api.jar` en la variable de entorno `CLASSPATH`. Así, no habrá que descomprimir ni teclear el camino del fichero `api.jar` cada vez que se desee compilar (como se mostró en el apartado 9.3.2.4.2 de la página 237).

Para compilar el applet Wallet se puede situar en el directorio `C:\pfc\java\java_card_kit-2_2\samples` (en el caso de que el sistema operativo sea Windows) y teclear lo siguiente:

```
javac -g com\sun\javacard\samples\wallet\*.java
```

Con lo que se obtiene el fichero `Wallet.class`.

9.3.2.6.3 Conversión de los ficheros `.class`

A continuación se deben convertir los ficheros `.class` pertenecientes a un mismo paquete. Para ello se utiliza la herramienta `converter`, de la que ya se habló en la página 237. Lo más práctico es usar un fichero de configuración. En el caso del applet que se está estudiando, se incluye un fichero de configuración (con extensión `*.opt`) en el mismo directorio donde se encuentra el fichero `Wallet.java`. Su contenido es el siguiente:

```
-out EXP JCA CAP
-exportpath .
-applet  0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x6:0x1
com.sun.javacard.samples.wallet.Wallet
com.sun.javacard.samples.wallet
0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xC:0x6 1.0
```

Con la opción `–out` se pueden especificar los ficheros a producir por el convertidor (CAP, EXP, JCA). La opción `-exportpath` indica los directorios donde el convertidor buscará los ficheros de exportación. Los ficheros de exportación que hacen falta son los pertenecientes a los paquetes `javacard.framework.*`, que se encuentran en el directorio `C:\pfc\java\java_card_kit-2_2 \api_export_files\`. Por lo tanto y si no se quiere copiar el contenido del directorio `api_export_files` al directorio de trabajo `src`, hay que modificar la línea asociada a `exportpath` de la siguiente forma:

```
-exportpath E:\pfc\java\java_card_kit-2_2\api_export_files
```

Antes de modificar esta línea, es conveniente realizar una copia de seguridad del fichero `Wallet.opt`, por si se quisiera ejecutar el fichero script para construir los ejemplos automáticamente (su uso se describe en la página 236). Si hiciera falta incluir más ficheros de exportación (de otros paquetes), hay que incluir sus rutas en la línea anterior, separadas por ; (en Windows) ó por : (en Linux).

Con la opción `–applet` se asignan un AID por defecto al applet y el nombre de la clase que define al applet. Las siguientes líneas:

```
com.sun.javacard.samples.wallet
0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xC:0x6 1.0
```

indican el nombre, el AID y la versión del paquete.

Ya que se ha explicado la estructura del fichero de configuración, se puede utilizar la herramienta de conversión de la siguiente forma en el directorio de trabajo:

```
converter -config com\sun\javacard\samples\wallet\Wallet.opt
```

Los ficheros resultantes de la ejecución de la herramienta de conversión (especificados en el fichero de configuración) se encuentran en el directorio `E:\pfc\java\java_card_kit-2_2\samples\src\com\sun\javacard \samples\wallet\javacard`.

9.3.2.6.4 Simulación

El último paso en el desarrollo de un applet, es poder probar su comportamiento. Si no dispone de un lector de tarjetas y una tarjeta Java Card, se puede simular la

respuesta del applet ante APDU's de comando. El primer paso para llevar a cabo la simulación, es establecer el conjunto de APDU's de comando u órdenes, que se van a pasar al applet. Un ejemplo de fichero (al que se le ha dado el nombre de `wallet.scr`) que contiene estas APDU's puede ser el siguiente:

```
// *** COMIENZO DEL SCRIPT ***

// Nota: en los comentarios que se encuentran debajo de la APDU
// de comando, se indica la respuesta esperada

// Encendido de la tarjeta
powerup;

// Seleccion del applet instalador
0x00 0xA4 0x04 0x00 0x09 0xa0 0x00 0x00 0x00 0x62 0x03 0x01 0x08
0x01 0x7F;
// 90 00 = SW_NO_ERROR

// Creacion del applet wallet. En el campo de datos se
// encuentran los parametros de instalacion. El PIN vale 0x01
// 0x02 0x03 0x04 0x05
// El AID es 0xa0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x6:0x1
0x80 0xB8 0x00 0x00 0x14 0x0a 0xa0 0x0 0x0 0x0 0x62 0x3 0x1 0xc
0x6 0x1 0x08 0 0 0x05 0x01 0x02 0x03 0x04 0x05 0x7F;

// Seleccion del applet Wallet (el AID se indica en el campo de
// datos
0x00 0xA4 0x04 0x00 0x0a 0xa0 0x0 0x0 0x0 0x62 0x3 0x1 0xc 0x6
0x1 0x7F;
// 90 00 = SW_NO_ERROR

// Validación mediante la introduccion del valor del PIN
0x80 0x20 0x00 0x00 0x05 0x01 0x02 0x03 0x04 0x05 0x7F;
// 90 00 = SW_NO_ERROR

// Se pide el balance de la tarjeta
0x80 0x50 0x00 0x00 0x00 0x02;
// 0x00 0x00 0x00 0x00 0x90 0x00 = Balance = 0 y no hay error
// (0x9000)

// Se intenta extraer 0x64 = 100 de una cuenta vacia
0x80 0x40 0x00 0x00 0x01 0x64 0x7F;
// 0x6A85 = SW_NEGATIVE_BALANCE

// Se introducen 0x64 = 100 a la cuenta vacía
0x80 0x30 0x00 0x00 0x01 0x64 0x7F;
// 0x9000 = SW_NO_ERROR

// Se pide el balance de la cuenta
0x80 0x50 0x00 0x00 0x00 0x02;
// 0x00 0x64 0x9000 = Balance = 100 y SW_NO_ERROR

// Se sacan 0x32 = 50 de la cuenta
0x80 0x40 0x00 0x00 0x01 0x32 0x7F;
```

```
// 0x9000 = SW_NO_ERROR

// Se pide el balance de la cuenta
0x80 0x50 0x00 0x00 0x00 0x02;
// 0x00 0x32 0x9000 = Balance = 50 y SW_NO_ERROR

// Se introducen 0x80 = 128 en la cuenta
0x80 0x30 0x00 0x00 0x01 0x80 0x7F;
// 0x6A83 = SW_INVALID_TRANSACTION_AMOUNT ya que 128 supera la
// cantidad definida en MAX_TRANSACTION_AMOUNT (127)

// Se pide el balance de la cuenta
0x80 0x50 0x00 0x00 0x00 0x02;
// 0x00 0x32 0x9000 = Balance = 50 y SW_NO_ERROR

// Se sacan 0x33 = 51 de la cuenta
0x80 0x40 0x00 0x00 0x01 0x33 0x7F;
//0x6A85 = SW_NEGATIVE_BALANCE ya que el balance anterior es de
// 50

// Se pide el balance de la cuenta
0x80 0x50 0x00 0x00 0x00 0x02;
// 0x00 0x32 0x9000 = Balance = 50 y SW_NO_ERROR

// Se sacan 0x80 = 128 de la cuenta
0x80 0x40 0x00 0x00 0x01 0x80 0x7F;
// 0x6A83 = SW_INVALID_TRANSACTION_AMOUNT ya que 128 supera la
// cantidad definida en MAX_TRANSACTION_AMOUNT (127)

// Se pide el balance de la cuenta
0x80 0x50 0x00 0x00 0x00 0x02;
// 0x00 0x32 0x9000 = Balance = 50 y SW_NO_ERROR

// Se vuelve a seleccionar el applet Wallet para volver a
// introducir el PIN
0x00 0xA4 0x04 0x00 0x0a 0xa0 0x0 0x0 0x0 0x62 0x3 0x1 0xc 0x6
0x1 0x7F;
// 90 00 = SW_NO_ERROR

// Se introducen 0x7F = 127 a la cuenta antes de la verificacion
// del PIN
0x80 0x30 0x00 0x00 0x01 0x7F 0x7F;
// 0x6301 = SW_PIN_VERIFICATION_REQUIRED

// Se introduce un valor erroneo de PIN
0x80 0x20 0x00 0x00 0x04 0x01 0x03 0x02 0x66 0x7F;
// 0x6300 = SW_VERIFICATION_FAILED

// Se introduce un valor correcto de PIN
0x80 0x20 0x00 0x00 0x05 0x01 0x02 0x03 0x04 0x05 0x7F;
// 0x9000 = SW_NO_ERROR

// Se pide el balance de la cuenta con un valor incorrecto de Le
// (0x01)
0x80 0x50 0x00 0x00 0x00 0x01;
// 0x6700 = ISO7816.SW_WRONG_LENGTH
```

```
// Se pide el balance de la cuenta
0x80 0x50 0x00 0x00 0x00 0x02;
// 0x00 0x32 0x9000 = Balance = 50 y SW_NO_ERROR

// Apagado de la tarjeta
powerdown;

// *** FIN DEL SCRIPT ***
```

Con este escenario planteado, solo queda definir los applets que van a intervenir en esta simulación. El fichero (`wallet.app`) que especifica los applets que intervendrán puede tener la siguiente estructura:

```
// Este fichero indica la localizacion y el AID de los applets
// involucrados en la simulacion. En este caso estan
// involucrados el applet instalador y el applet Wallet

// applet                                    AID
com.sun.javacard.installer.InstallerApplet   0xa0:0x0:0x0:0x0:0x0:0x62:0x3:0x1:0x8:0x1
com.sun.javacard.samples.wallet.Wallet       0xa0:0x0:0x0:0x0:0x0:0x62:0x3:0x1:0xc:0x6:0x1
```

En este ejemplo solo intervendrán el applet instalador y el applet Wallet. Con estos dos ficheros solo queda simular el escenario expuesto en el fichero `wallet.scr`, tecleando lo siguiente (hay que situarse en el directorio `src`, donde también deben estar los dos ficheros creados anteriormente):

```
jcwde -p 9025 wallet.app
```

Lo siguiente es abrir otra consola de comandos, volver a situarse en el directorio `src` y teclear lo siguiente:

```
apdutool wallet.scr > wallet.scr.out
```

El contenido del fichero `wallet.scr.out` se muestra a continuación:

```
Java Card 2.2 ApduTool (version 0.20)
Copyright 2002 Sun Microsystems, Inc. All rights reserved. Use is subject to
license terms.
Opening connection to localhost on port 9025.
Connected.
Received ATR = 0x3b 0xf0 0x11 0x00 0xff 0x00
CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 09, a0, 00, 00, 00, 62, 03, 01, 08, 01,
Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: b8, P1: 00, P2: 00, Lc: 14, 0a, a0, 00, 00, 00, 62, 03, 01, 0c,
06, 01, 08, 00, 00, 05, 01, 02, 03, 04, 05, Le: 0a, a0, 00, 00, 00, 62, 03,
01, 0c, 06, 01, SW1: 90, SW2: 00
CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 0a, a0, 00, 00, 00, 62, 03, 01, 0c, 06,
01, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: 20, P1: 00, P2: 00, Lc: 05, 01, 02, 03, 04, 05, Le: 00, SW1: 90,
SW2: 00
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 00, SW1: 90, SW2: 00
CLA: 80, INS: 40, P1: 00, P2: 00, Lc: 01, 64, Le: 00, SW1: 6a, SW2: 85
CLA: 80, INS: 30, P1: 00, P2: 00, Lc: 01, 64, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 64, SW1: 90, SW2: 00
CLA: 80, INS: 40, P1: 00, P2: 00, Lc: 01, 32, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 32, SW1: 90, SW2: 00
```

```
CLA: 80, INS: 30, P1: 00, P2: 00, Lc: 01, 80, Le: 00, SW1: 6a, SW2: 83
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 32, SW1: 90, SW2: 00
CLA: 80, INS: 40, P1: 00, P2: 00, Lc: 01, 33, Le: 00, SW1: 6a, SW2: 85
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 32, SW1: 90, SW2: 00
CLA: 80, INS: 40, P1: 00, P2: 00, Lc: 01, 80, Le: 00, SW1: 6a, SW2: 83
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 32, SW1: 90, SW2: 00
CLA: 00, INS: a4, P1: 04, P2: 00, Lc: 0a, a0, 00, 00, 00, 62, 03, 01, 0c, 06,
01, Le: 00, SW1: 90, SW2: 00
CLA: 80, INS: 30, P1: 00, P2: 00, Lc: 01, 7f, Le: 00, SW1: 63, SW2: 01
CLA: 80, INS: 20, P1: 00, P2: 00, Lc: 04, 01, 03, 02, 66, Le: 00, SW1: 63,
SW2: 00
CLA: 80, INS: 20, P1: 00, P2: 00, Lc: 05, 01, 02, 03, 04, 05, Le: 00, SW1: 90,
SW2: 00
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 00, SW1: 67, SW2: 00
CLA: 80, INS: 50, P1: 00, P2: 00, Lc: 00, Le: 02, 00, 32, SW1: 90, SW2: 00
```

En cada renglón se puede observar la APDU de comando y después del campo Le se muestra la APDU de respuesta. Se puede comprobar que las respuestas a las APDU's son las esperadas.

9.3.2.6.5 Conclusión del estudio de applet Java Card

El caso planteado en este estudio de un applet de Java Card es muy sencillo. Las aplicaciones reales exigen applets con características cada vez más complejas (codificación, decodificación, borrado de objetos,...) y que no se contemplan en el ejemplo expuesto. Estas características no las soporta la herramienta `jcwde`, por lo que hay que usar la herramienta `cref`, cuya utilización se ilustra a partir de la sección 9.3.2.5.2 Demostración `demo2` que se encuentra en la página 241. Pero esto no soluciona el problema de que a medida que aumenta la complejidad de los applets, también aumenta la dificultad de depurar los programas. La herramienta `cref` se basa en ficheros que contienen la imagen de la memoria de la tarjeta, pero `cref` no permite la observación de la evolución del estado de la tarjeta en un modo de ejecución paso a paso.

Por lo tanto, se puede recurrir al uso de herramientas de desarrollo de applets de Java Card. El inconveniente es que tienen un coste. Una de estas aplicaciones es Sm@rtCafé Professional PE (desarrollada por Giesecke & Devrient) de la que se puede encontrar una versión de evaluación en la página: `http://www.gi-de.com/eng/main/special/index.php4?special_id=86`. Esta herramienta permite al desarrollador visualizar el valor de la variables, la pila e incluso ejecutar un applet paso a paso. El usuario también debe introducir las APDU's en una tabla, con la posibilidad de ir enviándolas una a una a la tarjeta. A continuación se presenta una vista de esta aplicación:

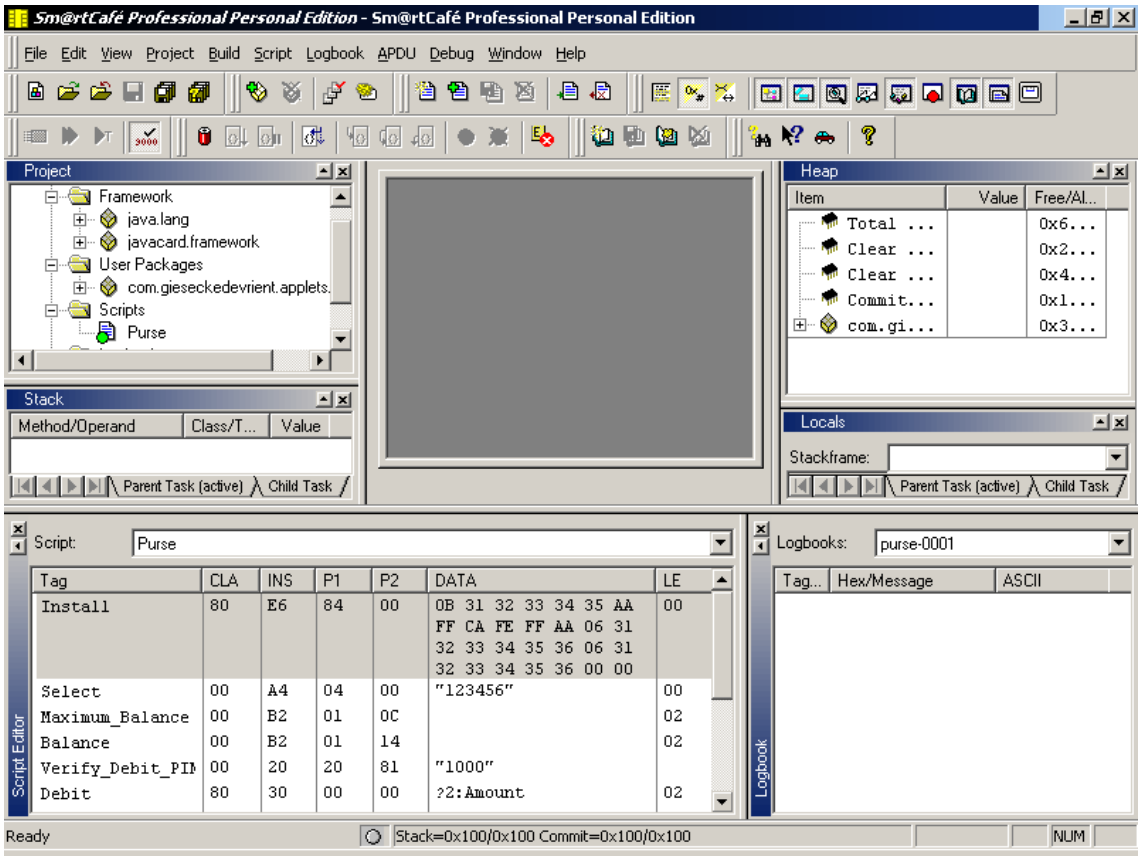


Figura 62: Vista de Sm@rtCafé Professional Edition

Es interesante ejecutar el tutorial guiado para aprender el manejo de la aplicación. El proyecto de muestra, prueba un applet muy parecido al applet Wallet ya estudiado, con lo que solo hay que familiarizarse con el manejo del programa. La única pega es que el programa no deja grabar los proyectos realizados por el usuario y también hay limitaciones en cuanto a la longitud del código fuente de los applets.

9.3.3 EJEMPLO DE UN APPLET TOOLKIT

9.3.3.1 Funcionalidad del applet

En esta sección se expone la estructura de un applet toolkit. Este applet toolkit se diferencia del realizado en que se puede comportar como un applet Java Card normal y además muestra como gestionar los ficheros y la recepción de mensajes SMS.

La funcionalidad de este applet es muy sencilla. Para elegir este applet (al igual que el de la aplicación realizada), de entre todos los presentes en la tarjeta, hay que seleccionar el nombre que lo identifica y que viene dado por la variable `menuEntry`. Cuando se activa este applet, presenta en la pantalla el título del menú (que se encuentra en la variable `menuTitle`) y cuatro opciones. Las tres primeras opciones presentan en pantalla el mensaje `Hello world`. La última le pregunta el nombre al usuario y presenta su nombre por pantalla. Además, mediante el envío de mensajes cortos por parte de un servidor, éste puede actualizar el contenido de un fichero GSM o cambiar el título del menú del applet (esta es la parte que hace interesante esta aplicación).

Además, el ME también puede seleccionar este applet mediante una APDU SELECT (la aplicación realizada no permite esta opción). Cuando el applet ya está seleccionado, la única instrucción que puede ejecutar el applet es `MY_INSTRUCTION`. Esta instrucción también permite cambiar el título del menú del applet.

9.3.3.2 Código fuente del applet `MyToolkitApplet`

A continuación se presenta el código fuente de este applet (llamado `MyToolkitApplet`):

```
/**
 * Ejemplo de un applet toolkit
 */

package ToolkitAppletExample;

import sim.toolkit.*;
import sim.access.*;
import javacard.framework.*;

// La clase debe estender la clase Applet e implementar
//ToolkitInterface y ToolkitConstants
public class MyToolkitApplet extends javacard.framework.Applet
implements ToolkitInterface, ToolkitConstants{

  // Se define una instruccion personaliza, cuya funcion se
  // puede ver en el metodo process
  public static final byte MY_INSTRUCTION = (byte)0x46;

  // Operación del servidor
  public static final byte SERVER_OPERATION = (byte)0x0F;

  // Calificador de un comando, se usa en los objetos TLV
  public static final byte CMD_QUALIFIER = (byte)0x80;

  // Salida solicitada por el usuario
  public static final byte EXIT_REQUESTED_BY_USER = (byte)0x10;

  // menuEntry (entrada al menu del applet) es un array que contiene
  // inicialmente la cadena: Service1
  private byte[] menuEntry = {(byte)'S',(byte)'e',(byte)'r',(byte)'v',
                              (byte)'i',(byte)'c',(byte)'e', (byte)'1'};

  // menuTitle (titulo del menu) es un array que contiene
  // la cadena: MyMenu
  private byte[] menuTitle = {(byte)'M',(byte)'y',(byte)'M',(byte)'e',
                              (byte)'n' ,(byte)'u'};

  // item1 (elemento 1) es un array que contiene la cadena: ITEM1
  private byte[] item1 = {(byte)'I',(byte)'T',(byte)'E',(byte)'M',(byte)'1' };

  // item2 (elemento 2) es un array que contiene la cadena: ITEM2
  private byte[] item2 = {(byte)'I',(byte)'T',(byte)'E',(byte)'M',(byte)'2' };

  // item3 (elemento 3) es un array que contiene la cadena: ITEM3
  private byte[] item3 = {(byte)'I',(byte)'T',(byte)'E',(byte)'M',(byte)'3' };

  // item3 (elemento 4) es un array que contiene la cadena: ITEM4
  private byte[] item4 = {(byte)'I',(byte)'T',(byte)'E',(byte)'M',(byte)'4' };

  // ItemList es un array de clases Object (superclase de byte)
  private Object[] ItemList = { item1, item2, item3, item4 };

  // textdText contiene la cadena: Hello world2
  private byte[] textDText = {(byte)'H',(byte)'e',(byte)'l',(byte)'l',
```

```
                            (byte)'o',(byte)' ',(byte)'w',(byte)'o',
                            (byte)'r',(byte)'l',(byte)'d',(byte)'2'};
  // textGInput contiene la cadena: Your name?
  private byte[] textGInput = {(byte)'Y',(byte)'o',(byte)'u',(byte)'r',
                               (byte)' ',(byte)'n',(byte)'a',(byte)'m',
                               (byte)'e',(byte)'?'};

  // AID de la aplicacion GSM
  private byte[] baGSMAID = {(byte)0xA0,(byte)0x00,(byte)0x00,(byte)0x00,
                            (byte)0x09,(byte)0x00,(byte)0x01};

  // La clase ToolkitRegistry permite al applet toolkit registrarse
  // en ciertos eventos (eventos soportados)
  private ToolkitRegistry reg;

  // El applet GSM mediante la interfaz SIMView ofrece servicios,
  // sin comprometer la integridad del sistema de ficheros GSM
  private SIMView gsmFile;

  // buffer de bytes
  private byte buffer[] = new byte[10];

  // itemId guarda el identificador del item seleccionado
  private byte itemId;

  // result guarda el resultado de la ejecucion de un comando proactivo
  private byte result;

  // Variable auxiliar utilizada en un bucle while
  private boolean repeat;

  /**
  * Constructor del applet
  */
  public MyToolkitApplet() {
    // Consigue una vista del sistema de ficheros GSM (o aplicacion GSM)
    gsmFile = SIMSystem.getTheSIMView();

    // Consigue una referencia de su entrada en el registro toolkit del
    // SIM Toolkit Framework para que pueda controlar el registro
    // a ciertos eventos.
    reg = ToolkitRegistry.getEntry();

    // Define el menu de entrada y se registra en el EVENT_MENU_SELECTION
    itemId = reg.initMenuEntry(menuEntry, (short)0x0000,
                               (short)menuEntry.length,
                               PRO_CMD_DISPLAY_TEXT, false, (byte) 0x00,
                               (short) 0x0000);
    // Tambien se registra en el evento de recepcion de datos
    // a traves de SMS-PP
    reg.setEvent(EVENT_UNFORMATTED_SMS_PP_ENV);
  }

  /**
  * El JCRE llama a este metodo para instalar el applet
  */
  public static void install(byte bArray[], short bOffset, byte bLength) {
    MyToolkitApplet MyApplet = new MyToolkitApplet ();

    // Se llama a este metodo register para que el applet tome
    // el AID que contenga el fichero CAP
    MyApplet.register();
  }

  /**
  * El GSM Framework llama a este metodo
  * Este applet se comporta como un servidor de objetos SIO
  */
```

```
  public Shareable getShareableInterfaceObject ( AID clientAID, byte
parameter){
    // Si el parametro parameter es 0
    if (parameter == (byte) 0x00) {
      // Si el AID de la aplicacion GSM coincide parcialmene con el AID
      // especificado en baGSMAID
      if ( clientAID.partialEquals(baGSMAID,
                    (byte) 0x00, (byte) baGSMAID.length) == true )
        // Concede el SIO
        return ((Shareable) this);
    }
    // No se concede el SIO en el resto de los casos
    return(null);
  }

  /**
  * El SIM Toolkit Framework llama a este metodo. cuando se produce
  * alguno de los eventos a los que esta registrado este applet
  */
  public void processToolkit(byte event) {
    // Se obtienen las referencias de los manejadores
    EnvelopeHandler          envHdlr = EnvelopeHandler.getTheHandler();
    ProactiveHandler         proHdlr = ProactiveHandler.getTheHandler();
    ProactiveResponseHandler rspHdlr;

    // Segun el evento ...
    switch(event) {
      case EVENT_MENU_SELECTION:
        // Prepara el lanzamiento del comando SELECT ITEM
        proHdlr.init(PRO_CMD_SELECT_ITEM,(byte)0x00,DEV_ID_ME);
        // Envia el Titulo del menu
        // TAG_ALPHA_IDENTIFIER indica que se van a enviar
        // identificadores numericos
        // TAG_SET_CR indica que el ME debe soportar el item por defecto
        // (p. ej. el item anteriormente seleccionado)
        proHdlr.appendTLV((byte) (TAG_ALPHA_IDENTIFIER | TAG_SET_CR),
                          menuTitle,(short)0x0000,(short)menuTitle.length);
        // Y añade las 4 opciones que se quieren representar en pantalla
        // En el segundo parametro se pone el identificador de la opcion
        for (short i=(short) 0x0000; i<(short) 0x0004; i++) {
          proHdlr.appendTLV((byte) (TAG_ITEM | TAG_SET_CR),(byte) (i+1),
                            (byte[])ItemList[i],(short) 0x0000,
                            (short)((byte[])ItemList[i]).length);
        }
        // Se pide al SIM Toolkit Framework que envie el comando proactivo
        // y se comprueba el resultado con RES_CMD_PERF (indica que el
        // comando se ha llevado a cabo correctamente)
        if ((result = proHdlr.send()) == RES_CMD_PERF) {
          // Se consigue una referencia al manejador ProactiveResponseHandler
          rspHdlr = ProactiveResponseHandler.getTheHandler();
          // Devuelve el identificador del elemento actual de datos
          // de respuesta
          switch (rspHdlr.getItemIdentifier()) {
            case 1:
            case 2:
            case 3: // Muestra texto por pantalla
              // Prepara el lanzamiento del comando DISPLAY TEXT
              proHdlr.init(PRO_CMD_DISPLAY_TEXT, CMD_QUALIFIER,
                           DEV_ID_DISPLAY);

              // Envia el mensaje: Hello world
              // TAG_TEXT_STRING indica que se van a enviar
              // una cadena de texto
              // TAG_SET_CR indica que el ME debe soportar
              // el item por defecto (p. ej. el item anteriormente
              // seleccionado)
              // DCS_8_BIT_DATA indica que se va a utilizar
              // un esquema de codificacion de 8 bits
```

```
          proHdlr.appendTLV((byte)(TAG_TEXT_STRING| TAG_SET_CR),
                                DCS_8_BIT_DATA, textDText,
                                (short)0x0000,
                                (short)textDText.length);
          // Se pide al SIM Toolkit Framework que envie el
          // comando proactivo
          proHdlr.send();
          break;
        case 4: // Se pide que el usuario introduzca datos
          do {
            repeat = false;
            try {
              // Al usuario se le pide que introduzca su nombre
              proHdlr.initGetInput((byte)0x01, DCS_8_BIT_DATA,
                                textGInput,(byte)0x00,
                                (short)textGInput.length,(short)0x0001,
                                (short)0x0002);
              // Envio del comando proactivo
              proHdlr.send();
              // Recepcion de los datos de respuesta
              rspHdlr.copyTextString(textDText,(short)0x0000);
              // Se mandan los datos recibidos a la pantalla del ME
              proHdlr.initDisplayText((byte)0x00,DCS_8_BIT_DATA,
                                    textDText, (short)0x0000,
                                    (short) textDText.length);
              // Envio del comando proactivo
              proHdlr.send();
            }
            catch (ToolkitException MyException) {
              // Si no hay un elemento disponible
              if (MyException.getReason() ==
                  ToolkitException.UNAVAILABLE_ELEMENT) {
                if (rspHdlr.getGeneralResult() != EXIT_REQUESTED_BY_USER)
                  // Si el usuario no ha solicitado salir del formulario
                  // vuelve a entrar en el bucle
                  repeat = true;
                  break;
              }
            }
          }
          while (repeat);
          break;
      }
    }
    break;
  case EVENT_UNFORMATTED_SMS_PP_ENV: // Llegada de un SMS-PP
    // Consigue el offset de la instruccion contenida en el campo
    // TP-UD (definido en TS 23.040)
    short TPUDOffset = (short) (envHdlr.getTPUDLOffset() +
                       SERVER_OPERATION);
    // Comienzo de la accion solicitada por el servidor
    // Se recoge la instruccion que se encuentra en el byte
    // dado por TPDUOffset
    switch (envHdlr.getValueByte((short)TPUDOffset) ) {
      case 0x41 : // Actualizacion de un fichero GSM
        // Se consiguen los datos del SMS con los
        // que se quiere actualizar
        envHdlr.copyValue((short)(TPUDOffset+1), buffer,
                          (short)0x0000,(short)0x0003);
        // Se escriben los datos en EFpuct
        // Este directorio contiene informacion acerca del cobro
        // de las llamadas, en la moneda elegida por el abonado
        gsmFile.select(SIMView.FID_DF_GSM);
        gsmFile.select(SIMView.FID_EF_PUCT);
        // En este caso solo se actualiza el codigo de moneda
        gsmFile.updateBinary((short)0x0000,buffer,(short)0x0000,
                             (short)0x0003);
        break;
```

```
          case 0x36 : // Faltan los valores solicitados
            // Cambia el MenuTitle por el SelectItem
            // Se consiguen los datos del SMS con los que se quiere
            // actualizar menuTitle
            envHdlr.copyValue((short)(TPUDOffset+1), menuTitle,
                              (short)0x0000,(short)0x0006);
            break;
        }
        break;
    }
  }

  /**
  * El JCRE llama a este metodo una vez que se selecciona el applet
  */
  public void process(APDU apdu) {
    // Maneja la APDU SELECT que contiene el AID de este applet

    // Si la APDU se esta usando para seleccionar el applet,
    // se sale del metodo
    // Asi se espera a la proxima instruccion
    if (selectingApplet()) {}
    else {
      switch(apdu.getBuffer()[1]) {
        // Una APDU especifica para este applet para configurar
        // el MenuTitle desde SelectItem
        case (byte)MY_INSTRUCTION:
          if (apdu.setIncomingAndReceive()>(short)0) {
            // Si la APDU contiene datos, los guarda en menuTitle
            // Cambia el titulo del menu
            Util.arrayCopy(apdu.getBuffer(),(short)0x0005,menuTitle,
                           (short)0x0000, (short)0x0006);
          }
          break;
        default:
          ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
      }
    }
  }
}
```

9.3.3.3 Simulación del ejemplo MyToolkitApplet

Para simular el ejemplo solo se tendrán que seguir los pasos expuestos para la para configurar y utilizar la herramienta Aspects Developer. Después, para probar la funcionalidad de applet como si fuera un applet Java Card normal, hay que realizar lo que se explica a continuación. Activar la opción `a View ► APDU Script`. Esto provocará la aparición de una ventana nueva donde se podrá insertar las APDU's una a una o cargar un fichero `.scr` (visto en ejemplos anteriores de applets Java Card). El fichero `.scr` que corresponde a este ejemplo, se encuentra en la ruta: `C:\Archivos de programa\Aspects Software Limited\Aspects developer 2.0\samples\SATSamples\GSM0319Example`. Una vez cargado, hay que pulsar Play en la ventana que se creó anteriormente y ver las APDU's intercambiadas con el applet.

10 APÉNDICE 1: CONTENIDO DEL PAQUETE JAVA.LANG

Esta clase provee clases que son fundamentales para el diseño del subconjunto (usado en la tecnología Java Card) del lenguaje de programación Java.

10.1 CLASES

10.1.1 CLASE OBJECT

`public class Object` : La clase `Object` es la raíz de la jerarquía de clases de Java Card.

10.1.1.1 Constructores

```
public Object()
```

10.1.1.2 Métodos

`public boolean equals(Object obj)` : Este método compara dos objetos para ver si son iguales. El parámetro `obj` es la referencia del objeto con el que se quiere comparar. El método devuelve `true` si el objeto es el mismo que el dado por el argumento `obj`. Devuelve `false` en caso contrario.

10.1.2 CLASE THROWABLE

`public class Throwable` : La clase `Throwable` es la superclase de todos los errores y excepciones en el subconjunto del lenguaje de Java.

10.1.2.1 Constructores

public Throwable() : Construye un objeto `Throwable`.

10.2 EXCEPCIONES

10.2.1 EXCEPCIÓN ARITHMETICEXCEPTION

`public class ArithmeticException extends Runtime Exception` : Se lanza una instancia de una `ArithmeticException` del JCRE cuando ocurre una condición aritmética especial. Por ejemplo, una división entre cero es una condición aritmética especial.

10.2.1.1 Constructores

`public ArithmeticException()` : Construye una `ArithmeticException`.

10.2.2 EXCEPCIÓN ARRAYINDEXOUTOFBOUNDSEXCEPTION

`public class ArrayIndexOutOfBoundsException extends IndexOutOfBoundsException` : Se lanza una instancia del JCRE de esta excepción para indicar que se ha accedido al array con un índice ilegal. Un índice es ilegal cuando es negativo o mayor que el tamaño del array.

10.2.2.1 Constructores

`public ArrayIndexOutOfBoundsException()` : Construye una `ArrayIndexOutOfBoundsException`.

10.2.3 EXCEPCIÓN ARRAYSTOREEXCEPTION

`public class ArrayStoreException extends RuntimeException` : Se lanza una instancia del JCRE de este tipo de excepción para indicar que se ha intentado guardar un tipo incorrecto de objeto en un array de objetos diferentes.

10.2.3.1 Constructores

`public ArrayStoreException()` : Construye una `ArrayStoreException`.

10.2.4 EXCEPCIÓN CLASSCASTEXCEPTION

`public class ClassCastException extends RuntimeException` : Se lanza una instancia de una `ClassCastException` del JCRE para indicar que el código ha intentado hacer un cast del tipo de una subclase que no es una instancia. Por ejemplo:

```
Object x = new OwnerPIN((byte)3, (byte)8);
JCSystem.getAppletShareableInterfaceObject((AID)x, (byte)5);
```

10.2.4.1 Constructores

`public ClassCastException()` : Construye una `ClassCastException`.

10.2.5 EXCEPCIÓN EXCEPTION

`public class Exception extends Throwable` : La clase `Exception` y sus subclases son una forma de `Throwable` que indica condiciones que quizás un applet sensato quiera capturar.

10.2.5.1 Constructores

`public Exception()` : Construye una instancia `Exception`.

10.2.6 EXCEPCIÓN `INDEXOUTOFBOUNDSEXCEPTION`

`public class IndexOutfBoundsException extends RuntimeException` : Se lanza una instancia de una `IndexOutOfBoundsException` del JCRE para indicar que un índice de algún tipo está fuera de rango.

10.2.6.1 Constructores

`public IndexOutOfBoundsException()` : Construye una `IndexOutOfBoundsException`.

10.2.7 EXCEPCIÓN `NEGATIVEARRAYSIZEEXCEPTION`

`public class NegativeArraySizeException extends RuntimeException` : Se lanza una instancia de una `NegativeArraySizeException` del JCRE si un applet intenta crear un array con tamaño negativo.

10.2.7.1 Constructores

`public NegativeArraySizeException()` : Construye una `NegativeArraySizeException()`.

10.2.8 EXCEPCIÓN `NULLPOINTEREXCEPTION`

`public class NullPointerException extends RuntimeException` : Se lanza una instancia de una `NullPointerException` del JCRE cuando un applet intenta usar `null` en el caso donde se requiere un objeto. Esto incluye:

- La llamada de la instancia de un método de un objeto `null`.
- El acceso o modificación del campo de un objeto `null`.
- La consulta del tamaño de `null` como si fuera un array.
- El acceso o modificación de los campos de `null` como si fuera un array.
- El lanzamiento de `null` como si fuera un valor `Throwable`.

10.2.8.1 Constructores

`public NullPointerException()` : Construye una `NullPointerException`.

10.2.9 EXCEPCIÓN `RUNTIMEEXCEPTION`

`public class RuntimeException extends Exception` : RuntimeException es la superclase de aquellas excepciones que se pueden lanzar durante la operación normal de la máquina virtual de Java Card.

10.2.9.1 Constructores

`public RuntimeException()` : Construye una instancia de la `RuntimeException`.

10.2.10EXCEPCIÓN `SECURITYEXCEPTION`

`public class SecurityException extends RuntimeException` : La máquina virtual de Java Card lanza una instancia de `SecurityException` del JCRE para indicar una violación de seguridad.

10.2.10.1 Constructores

`public SecurityException()` : Construye una `SecurityException`.

11 APÉNDICE 2: CONTENIDO DEL PAQUETE JAVACARD.FRAMEWORK

Este paquete proporciona el soporte de las clases y las interfaces para la funcionalidad principal de un applet de Java Card.

11.1 INTERFACES

11.1.1 INTERFAZ ISO7816

`public interface ISO 7816` : La interfaz 7816 encapsula las constantes relacionadas con la ISO 7816-3 e ISO 7816-4. La interfaz ISO7816 sólo contiene campos estáticos.

11.1.1.1 Campos

`public static final byte CLA_ISO7816` : Campo CLA de una APDU de comando (CLA = 0x00).

`public static final byte INS_EXTERNAL_AUTHENTICATE` : La instrucción EXTERNAL AUTHENTICATE de una APDU de comando tiene el valor 0x82.

`public static final byte INS_SELECT` : La instrucción SELECT de una APDU de comando tiene el valor 0xA4.

`public static final byte OFF_CDATA` : Offset de los datos de una APDU de comando (CDATA = %).

`public static final byte OFFSET_CLA` : Offset del campo CLA en la cabecera de una APDU de comando (CLA = 0).

`public static final byte OFFSET_INS` : Offset del campo INS en la cabecera de una APDU de comando (INS =1).

`public static final byte OFFSET_LC` : Offset del campo Lc en la cabecera de una APDU de comando (LC = 4).

`public static final byte OFFSET_P1` : Offset del campo P1 en la cabecera de una APDU de comando (P1 = 2).

`public static final byte OFFSET_P2` : Offset del campo P2 en la cabecera de una APDU de comando (P2 = 3).

`public static final short SW_APPLET_SELECT_FAILED` : Indica, en una respuesta, que la selección del applet ha fallado. Tiene el valor 0x6999.

`public static final short SW_BYTES_REMAINING_00` : Indica, en una respuesta, que todavía quedan datos de la respuesta.

`public static final short SW_CLA_NOT_SUPPORTED` : Indica, en una respuesta, que no se soporta el valor contenido en el campo CLA. Tiene el valor 0x6E00.

`public static final short SW_COMMAND_NOT_ALLOWED` : Indica que no está permitido ese comando. Tiene el valor 0x6986.

`public static final short SW_CONDITIONS_NOT_SATISFIED` : Indica que las condiciones de uso no se satisfacen. Tiene el valor 0x6985.

`public static final short SW_CORRECT_LENGTH_00` : Indica la longitud correcta esperada (Le). Tiene el valor 0x6C00.

`public static final short SW_DATA_INVALID` : Indica que los datos de la APDU de comando no son válidos. Tiene el valor 0x6984.

`public static final short SW_FILE_FULL` : Indica que no hay suficiente espacio de memoria en el fichero. Tiene el valor 0x6A84.

`public static final short SW_FILE_INVALID` : Indica que el fichero no es válido. Tiene el valor 0x6983.

`public static final short SW_FUNC_NOT_SUPPORTED` : Indica que no se soporta la función solicitada. Tiene el valor 0x6A81.

`public static final short SW_INCORRECT_P1P2` : Indica que los parámetros P1 y P2 son incorrectos. Tiene el valor 0x6A86.

`public static final short SW_INS_NOT_SUPPORTED` : Indica que no se soporta el valor contenido en el campo INS. Tiene el valor 0x6D00.

`public static final short SW_NO_ERROR` : Indica que no se ha producido ningún error. Tiene el valor 0x9000.

`public static final short SW_RECORD_NOT_FOUND` : Informa de que no se ha encontrado el registro. Tiene el valor 0x6A83.

`public static final short SW_SECURITY_STATUS_NOT_SATISFIED` : Informa de que no se satisface la condición de seguridad. Tiene el valor 0x6982.

`public static final short SW_UNKNOWN` : Indica que la tarjeta no tiene un diagnóstico preciso del error. El valor es 0x6F00.

`public static final short SW_WRONG_DATA` : Informa de que los datos son erróneos. El valor es 0x6A80.

`public static final short SW_LENGTH` : Informa de que la longitud es incorrecta. Su valor es 0x6700.

`public static final short SW_WRONG_P1P2` : Informa de que los parámetros P1 y P2 son incorrectos. Su valor es 0x6B00.

11.1.2 INTERFAZ `PIN`

Esta interfaz representa un PIN. Una implementación debe mantener estos valores internos:

- El valor del PIN.
- El número máximo de intentos en los que se pueden presentar un PIN incorrecto antes de que se bloquee el PIN.
- El máximo tamaño del PIN, es decir, la longitud máxima permitida del PIN.
- El contador de intentos, que representa el número de veces que se puede presentar un PIN incorrecto antes de que se bloquee el PIN.
- La bandera de validación, que vale `true` si se presenta un PIN válido. Esta bandera se resetea en cada reset de la tarjeta.

11.1.2.1 Métodos

`public boolean check(byte[] pin, short offset, byte length)` : Compara el contenido de `pin`, que es un array de bytes, con el valor del PIN. Si coinciden y el objeto `PIN` no está bloqueado, ajusta la bandera de validación y resetea el contador de intentos. Si no coinciden, decrementa el contador de intentos y si llega a cero, bloquea el `PIN`. `offset` indica el comienzo del PIN en el array `pin` y `length` indica el tamaño. El método devuelve `true` si los datos coinciden y `false` en caso contrario.

`public getTriesRemaining()` : Devuelve el número de intentos incorrectos que quedan, antes de que se bloquee el `PIN`.

`public boolean isValidated()` : Devuelve `true` si se ha presentado un PIN válido desde el último reset de la tarjeta o desde la última llamada al método `reset()`.

`public void reset()` : Si la bandera de validación está ajustada, este método la resetea. Si la bandera no está ajustada, este método no hace nada.

11.1.3 INTERFAZ `SHAREABLE`

`public interface Shareable` : Esta interfaz sirve para identificar todos los objetos compartidos. Cualquier objeto que se necesite compartir a través del applet firewall debe (directa o indirectamente) implementar esta interfaz. Solo aquellos métodos especificados en una interfaz compartida están disponibles a través del firewall. Las clases que la implementen pueden implementar cualquier número de interfaces compartidas y pueden extender otras clases que implementen interfaces compartidas.

11.2 CLASES

11.2.1 CLASE `AID`

`public final class ADI` : Esta clase encapsula el identificador de aplicación (AID) asociado a un applet. En la ISO 7816-5 se define un AID como una secuencia de 5 a 16 bytes de longitud. El JCRE crea instancias de la clase `AID` para identificar y gestionar cada applet de la tarjeta. Los applets no necesitan crear instancias de esta clase. Un applet puede solicitar y usar las instancias del JCRE para identificarse a si mismo y a otras instancias de applets. Las instancias pertenecientes al JCRE son objetos permanentes de punto de entrada al JCRE y se puede acceder a ellas desde cualquier contexto. Las referencias a estos objetos permanentes se pueden guardar y reutilizar.

11.2.1.1 Constructores

`public AID(byte[] bArray, short offset, byte length)` : El JCRE usa este constructor para crear una nueva instancia de `AID` que encapsula los bytes del AID especificado en el array de bytes `bArray`.(`offset` indica el comienzo de los bytes del AID y `length` la longitud en bytes del AID). Lanza una `SystemException.ILLEGAL_VALUE` si el parámetro `length` es menor que 5 o si es mayor de 16.

11.2.1.2 Métodos

`public boolean equals(byte[] bArray. short offset, byte length)` : Comprueba si los bytes del AID especificados en el array `bArray` son los mismos que los que se encuentran encapsulados en ese objeto `AID`. El método devuelve `true` si el array `bArray` no es `null` y los bytes del AID encapsulados en el objeto `AID` son iguales a los especificados en dicho array. `offset` y `length` marcan el inicio y la longitud de los bytes del AID en el array `bArray`.

`public boolean equals(Object anObject)` : Este método compara los bytes del AID de esta instancia de `AID` con los bytes del AID especificado en el objeto `anObject`. El resultado es `true` si y solo si el argumento no es `null` y es un objeto `AID` que encapsula los mismos bytes de AID que el objeto que llama al método.

`public byte getBytes(byte[] dest, short offset)` : Este método devuelve el número de bytes del AID encapsulados en el objeto. El argumento `dest` indica el array de destino donde se quieren copiar los bytes del AID y `offset` indica la posición del array a partir de la cual se empezará la copia.

`public boolean partialEquals(byte[] bArray, short offset, byte length)` : Este método comprueba si la secuencia parcial de bytes del AID coincide con los primeros `length` bytes de los bytes del AID encapsulado en el objeto `AID`. El resultado es `true` si y solo si el argumento `bArray` no es `null`, la entrada `length` es menor o igual que la longitud de los bytes del AID encapsulados en el objeto `AID` y los bytes especificados coinciden. `offset` indica la posición de `bArray` donde empieza la comparación.

`public boolean RIDEquals(AID otherAID)` : Comprueba si el RID (National Registered Application Provider Identifier) del AID encapsulado en el objeto `otherAID` coincide con los del objeto `AID` que llama al método. El resultado es `true` si y solo si el argumento no es `null` y `otherAID` contiene los mismos bytes del RID que el objeto `AID`.

11.2.2 CLASE APDU

`public final class APDU` : Esta clase solo soporta los mensajes que cumplen con las estructuras de comando y respuesta definidos en la ISO 7816-4. El comportamiento de los mensajes que usan una estructura propietaria de mensajes (por ejemplo el byte CLA en el rango 0xD0-0xFE) está indefinido. El objeto `APDU` pertenece al JCRE. La clase APDU mantiene un buffer que es un array de bytes. Este buffer se usa para transferir la cabecera de la APDU y sus bytes de datos, tanto para APDU's entrantes como salientes.

11.2.2.1 Campos

`public static final PROTOCOL_T0` : Especifica el protocolo de transporte ISO 7816 del tipo T=0.

`public static final byte PROTOCOL_T1` : Especifica el protocolo de transporte ISO 7816 del tipo T=0.

11.2.2.2 Métodos

`public bytes[] getBuffer()` : Devuelve el buffer ADPU en forma de un array de bytes.

`public static short getInBlockSize()` : Devuelve el tamaño de bloque entrante configurado. En el protocolo T=1 corresponde al IFSC, es decir, el máximo tamaño de los bloques de datos entrantes. En el protocolo T=0, este método devuelve 1.

`public byte getNAD()` : En el protocolo T=1, este modo devuelve el byte de la dirección del nodo. (Node Address). En el protocolo T=0, este método devuelve 0.

`public static short getOutBlockSize()` : Devuelve el tamaño de bloque saliente configurado. En el protocolo T=1, este corresponde con el IFSD, es decir, el máximo tamaño de los bloques de datos salientes. En el protocolo T=0, este método devuelve 258

`public static byte getProtocol()` : Este método devuelve el tipo de protocolo de transporte ISO 7816, T=1 (PROTOCOL_T1) ó T=0 (PROTOCOL_T0).

`public short receiveBytes(short bOff)` : Consigue tantos bytes de datos como quepan, sin provocar un overflow en el buffer APDU, a partir del offset `bOff`.

`public void sendBytes(short bOff, short len)` : Envía `len` bytes más desde el buffer APDU a partir del offset `bOffset` especificado. Si se está enviando la última parte de la respuesta gracias a la invocación de este método, el buffer APDU no se debería alterar. Si los datos se alteran, quizás se envíen datos incorrectos al CAD. El parámetro `len` indica los bytes de datos que se quieren enviar.

`public void sendBytesLong(byte[] outData, short bOff, short len)` : Este método envía `len` bytes más desde el array de bytes `outData`, empezando a partir del offset `bOff`. Si se está enviando la última parte de los datos de respuesta gracias a la invocación de este método, no se debe alterar el buffer APDU. El argumento `len` indica la cantidad de bytes de datos que se van enviar.

`public short setIncomingAndReceive()` : Este es el método primario de recepción. La llamada a este método indica que la APDU tiene datos entrantes. Este método consigue tantos bytes como quepan sin provocar el overflow del buffer APDU. Consigue todos los bytes entrantes si caben en el buffer APDU.

`public short setOutgoing()` : Este método se usa para ajustar la dirección de transferencia de datos hacia fuera y para obtener la longitud esperada de la respuesta (Le).

`public void setOutgoingAndSend(short bOff, short len)` : Este es el método de envío más conveniente. Provee la forma más eficiente de enviar una respuesta corta que quepa en el buffer con la menor carga de protocolo posible. Este método es una combinación de `setOutgoing()`, `setOutgoingLength(len)` seguido de `sendBytes(bOff, len)`. Además, una vez que el método se ha invocado, no se pueden invocar a los métodos `sendBytes()` y `sendBytesLong()` y el buffer APDU no se debe alterar. Este método envía `len` bytes de respuesta desde el buffer APDU empezando a partir de offset `bOff`.

`public void setOutgoingLength(short len)` : Ajusta la longitud (indicada por `len`) real de los datos de respuesta. Por defecto es 0.

`public short setOutgoingNoChaining()` : Este método se usa para ajustar la dirección de transferencia de datos hacia fuera sin usar el encadenamiento de bloques y para obtener la longitud esperada de la respuesta.

`public static void waitExtension()` : Solicita al CAD tiempo de procesamiento adicional. La implementación debería asegurar que este método se necesita invocar solo bajo condiciones inusuales de tiempos excesivos de procesamiento.

11.2.3 CLASE APPLET

`public abstract class Applet` : Esta clase abstracta define un applet de Java Card. Cualquier applet que se cargue, instale y ejecute en una smart card de Java Card, debería extender la clase `Applet`.

11.2.3.1 Constructores

`protected Applet()` : Solo el método `install()` de esta clase debería crear un objeto de un applet.

11.2.3.2 Métodos

`public void deselect()` : El JCRE llama a este método para informar al applet actualmente seleccionado de que se seleccionará otro (o el mismo) applet. Se llama cuando el JCRE recibe una APDU SELECT de comando. Una subclase de la clase `Applet` debería sobrescribir este método si tiene que realizar alguna operación antes de que se seleccione otro applet.

`public Shareable getShareableInterfaceObject(AID clientAID, byte parameter)` : El JCRE llama a este método para obtener un objeto de interfaz compartida del applet servidor, de parte de una solicitud de un applet cliente. El applet cliente inicia esta solicitud mediante una llamada al método `JCSystem.getAppletShareableInterfaceObject()`. El parámetro `clientAID` representa el objeto AID del objeto cliente y el parámetro `parameter` es un byte opcional que quizás use el applet cliente para especificar qué objeto de interfaz compartida está solicitando.

`public static void install(byte[] bArray, short bOffset, byte bLength)` : El JCRE realizará una llamada a este método estático para crear una instancia de la subclase `Applet`. Los parámetros de la instalación se suministran en el parámetro array de bytes (`bArray`) y deben estar en el formato definido en el applet. `bOffset` y `bLength` indican la posición y la longitud de los parámetros en `bArray`. El applet debería realizar cualquier inicialización necesaria y debe llamar a uno de los métodos `register()`. La instalación se considera exitosa cuando la llamada a `register()` se completa sin que se lance una excepción. Si la instalación no se completa de forma exitosa, el JCRE llevará a cabo cualquier operación de limpieza cuando recupere el control. Una instalación exitosa hace que una APDU SELECT de comando sea capaz de seleccionar la instancia del applet.

`public abstract void process(APDU apdu)` : Este método lo llama el JCRE para procesar una APDU de comando entrante. En este método se espera que el applet lleve a cabo la acción solicitada y devuelva datos de respuesta al terminal. Después y si no ha ocurrido ningún error, el JCRE envía la palabra de estado 9000 en una APDU de respuesta. Los cinco bytes de la cabecera de la APDU de comando están disponibles en el instante en el que se llama a este método.

`protected final void register()` : El applet usa este método para registrar la instancia del applet en el JCRE y para asignar los bytes del AID de la subclase `Applet` a los bytes del AID de la instancia. Se debe llamar a uno de los dos método `register()` desde el método `install()`.

`protected final void register(byte[] bArray, short bOffset, byte bLength)` : Se usa este método para registrar esta instancia del applet en el JCRE y para asignarle los bytes del AID especificado (en `bArray`) a los bytes de su AID. Se debe llamar a uno de los dos métodos `register()` desde el método

`install()`. `bOffset` indica el comienzo de los bytes del AID en `bArray`. `bLength` indica la longitud de los bytes del AID en `bArray`.

`public boolean select()` : El JCRE llama al método para informar a este applet de que ha sido seleccionado. Se llama cuando se recibe una APDU SELECT de comando, incluso cuando ya estaba seleccionado. Una subclase de `Applet` debería sobrescribir este método si requiere realizar cualquier inicialización para procesar las APDU's de comando que quizás vengan. Este método devuelve `true` para indicar que está preparado para aceptar APDU's de comando a través de su método `process()`.

`protected final boolean selectingApplet()` : El método `process()` del applet usa este método para distinguir de la APDU SELECT de comando que seleccionó este applet, de otras de APDU's SELECT de comando que quizás tengan que ver con un fichero o la selección de un estado interno del applet. El método devuelve `true` si el applet está siendo seleccionado.

11.2.4 CLASE `JCSYSTEM`

`public final class JCSystem` : La clase `JCSystem` incluye una colección de métodos para controlar la ejecución del applet, la gestión de recursos, la gestión de transacciones atómicas y la compartición de objetos entre applets de Java Card. Todos los métodos de la clase `JCSystem` son métodos estáticos. La clase `JCSystem` también incluye métodos para controlar la persistencia o la transitoriedad de los objetos.

11.2.4.1 Campos

`public static final byte CLEAR_ON_DESELECT` : Este código de evento indica que los contenidos del objeto transitorio se ponen al valor por defecto cuando se produce el evento de la deselección del applet o en los casos de `CLEAR_ON_RESET`.

`public static final byte CLEAR_ON_RESET` : Este código de evento indica que los contenidos de los objetos transitorios se ponen a sus valores por defecto en un reset de la tarjeta (o encendido).

`public static final byte NOT_A_TRANSIENT_OBJECT` : Este código de evento indica que el objeto no es transitorio.

11.2.4.2 Métodos

`public static native void abortTransaction()` : Interrumpe la transacción atómica. Los contenidos del "commit buffer" se desechan.

`public static native void beginTransaction()` : Indica el comienzo de una transacción atómica. Si una transacción está todavía en progreso, se lanza una `TransactionException`.

`public static native void commitTransaction()` : Compromete una transacción atómica. Los contenidos del "commit buffer" se comprometen atómicamente. Si una transacción no está en progreso entonces se lanza una `TransactionException`.

`public static ADI getAID()` : Devuelve la instancia del objeto `AID` (perteneciente al JCRE) asociado al contexto del applet actualmente seleccionado. Devuelve `null` si el método `Applet.register()` no se ha invocado todavía.

`public static Shareable getAppletShareableInterfaceObject (AID serverAID, byte parameter)` : Un applet cliente llama a este método para conseguir un objeto de interfaz compartida del applet servidor. Este método devuelve `null` si todavía no se ha llamado a `Applet.register()` o si el servidor no existe o si el servidor devuelve `null`. El argumento `serverAID` indica el AID del applet servidor y `parameter` es un parámetro opcional de datos.

`public static native short getMaxCommitCapacity` : Devuelve el número total de bytes del "commit buffer". Este es aproximadamente el número de bytes de datos persistentes que se pueden modificar durante una transacción. Sin embargo, el subsistema de transacción requiere bytes adicionales de datos de sobrecarga que se incluyen en el "commit buffer", y que dependen del número de campos modificados y de la implementación del subsistema de transacción.

`public static AID getPreviousContext()` : Se llama a este método para obtener la instancia (perteneciente al JCRE) del objeto AID asociado al contexto del applet activo previamente. Los applets servidores suelen usar este método en la ejecución de un método de interfaz compartida, para determinar la identidad de su cliente y así controlar los privilegios de acceso.

`public static native byte getTransactionDepth()` : Devuelve el nivel de profundidad de anidamiento de transacciones actual. En un instante solo puede haber una transacción en progreso a la vez. El método devuelve 1 si hay una transacción en progreso y 0 si no hay ninguna.

`public static native short getUnusedCommitCapacity()` : Devuelve el número de bytes que quedan en el "commit buffer".

`public static short getVersion()` : Devuelve la versión actual mayor y menor de la API de Java Card. El número devuelto tiene el formato byte.byte (mayor.menor).

`public static native byte isTransient(Object theObj)` : Se usa para comprobar si el objeto especificado (por el parámetro `theObj`) es transitorio. El método devuelve `NOT_A_TRANSIENT_OBJECT` si el objeto especificado es `null` o no es un tipo array. Los valores que puede devolver son: `NOT_A_TRANSIENT_OBJECT`, `CLEAR_ON_RESET` ó `CLEAR_ON_DESELECT`.

`public static AID lookupAID(byte[] buffer, short offset, byte length)` : Devuelve la instancia (perteneciente al JCRE) del objeto `AID` que encapsula los bytes del AID especificados en parámetro `buffer` si existe un applet instalado de forma exitosa en la tarjeta, cuya instancia de AID coincide exactamente con

los bytes del AID especificado. `buffer` contiene los bytes de AID, `offset` indica donde comienzan los bytes del AID y `length` la longitud en bytes del AID.

`public static native boolean[] makeTransientBooleanArray (short length, byte event)` : Crea un array transitorio de variables booleanas. El parámetro `length` especifica la longitud del array y el parámetro `event` (de la forma `CLEAR_ON...`) indica el evento que provoca el borrado del array.

`public static native byte[] makeTransientByteArray(short length, byte event)` : Crea un array transitorio de enteros. El parámetro `length` especifica la longitud del array y el parámetro `event` (de la forma `CLEAR_ON...`) indica el evento que provoca el borrado del array.

`public static native Object[] makeTransientObjectArray (short length, byte event)` : Crea un array de objetos transitorios. El parámetro `length` especifica la longitud del array y el parámetro `event` (de la forma `CLEAR_ON...`) indica el evento que provoca el borrado del array.

`public static native short[] makeTransientShortArray(short length, byte event)` : Crea un array transitorio de variables del tipo `short`. El parámetro `length` especifica la longitud del array y el parámetro `event` (de la forma `CLEAR_ON...`) indica el evento que provoca el borrado del array.

11.2.5 CLASE `OWNERPIN`

`public class OwnerPIN implements PIN` : Esta clase representa un PIN del propietario. Implementa la funcionalidad de número personal de identificación tal y como se define en la interfaz `PIN`.

11.2.5.1 Constructores

`public OwnerPIN(byte tryLimit, byte maxPINSize)` : Asigna una nueva instancia de `PIN` con la bandera de validación puesta a `false`. `tryLimit` indica el número máximo de veces que se puede presentar un PIN incorrecto. `maxPINSize` (que debe ser >=1) es el máximo tamaño permitido para el PIN.

11.2.5.2 Métodos

`public boolean check(byte[] pin, short offset, byte length)` : Compara `pin` con el valor PIN. Si coinciden (el método devuelve `true`) y el `PIN` no está bloqueado, ajusta la bandera de validación y reseta el contador de intentos al máximo. Si no coinciden, decrementa el contador de intentos y, si el contador ha llegado a cero, bloquea el `PIN`. `offset` da el comienzo en el array del valor PIN que se va a comprobar y `length` da la longitud de `pin`.

`public byte getTriesRemaining()` : Devuelve el número de veces que se puede presentar un PIN incorrecto antes de que el objeto `PIN` se bloquee.

`protected boolean getValidatedFlag()` : Este método protegido devuelve la bandera de validación.

`public boolean isValidated()` : Devuelve `true` si se ha presentado un PIN desde el último reset de la tarjeta o la última llamada a `reset()`.

`public void reset()` : Si la bandera de validación está ajustada, este método la resetea. Si la bandera de validación no está ajustada, este método no hace nada.

`public void resetAndUnblock()` : Este método resetea la bandera de validación y resetea el contador de intentos de `PIN` a un valor dado por un límite de intentos que se encuentra en `PIN`.

`protected void setValidatedFlag(boolean value)` : Este método protegido ajusta el valor de la bandera de validación, al valor dado por el parámetro `value`.

`public void update(byte[] pin, short offset, byte length)` : Este método ajusta un nuevo valor (que se encuentra en `pin`)para el PIN y resetea el contador de intentos de PIN al valor dado por un límite de intentos que se encuentra en `PIN`. También resetea la bandera de validación. `offset` indica el comienzo del PIN en el array `pin`. `length` da la longitud del nuevo PIN.

11.2.6 CLASE `UTIL`

`public class Util` : La clase `Util` contiene funciones útiles. Algunos de los métodos quizás estén implementados como métodos nativos por razones de rendimiento. Todos los métodos de la clase `Util`, son métodos estáticos. Los métodos `arrayCopy()`, `arrayCopyNonAtomic()`, `arrayFillNonAtomic()` y `setShort()`, hacen referencia a la persistencia de los arrays de objetos.

11.2.6.1 Métodos

`public static final native byte arrayCompare(byte[] src, short srcOff, byte[] dest, short destOff, short length)` : Este método compara un array dado por el array de origen especificado (`src`), empezando a partir de una posición especificada (`srcOff`), con la posición especificada del array de destino (`dest`) de izquierda a derecha. El método devuelve lo siguiente: 0 si son idénticos, -1 si `src` < `dest` y 1 si `src` > `dest`. `length` indica el número de bytes a comparar.

`public static final native short arrayCopy(byte[] src, short srcOff, byte[] dest, short destOff, short length)` : Copia un array desde el array de origen especificado, empezando a partir de la posición especificada (por `srcOff`), hasta la posición especificada (por `destOff`) del array destino (`dest`). `length` indica el número de bytes a copiar. Si el array de destino es persistente, la copia entera se lleva a cabo atómicamente. La operación de copia está sujeta a la capacidad del "commit buffer". Si se excede su capacidad, la copia no se realiza y se lanza una `TransactionException`. Si los argumentos `src` y `dest` se refieren al mismo array, entonces la copia se lleva a cabo como si los componentes de la posición `srcOff` hasta la `srcOff+length-1` se copiaran primero a un array temporal

y luego los `length` componentes del array temporal se volvieran a copiar desde la posición `destOff` hasta la `destOff+length-1` del array `dest`. El método devuelve `destOff+length`.

`public static final native short arrayCopyNonAtomic(byte[] src, short src srcOff, byte[] dest, short destOff, short length)` : Copia un array desde el array especificado de origen, empezando a partir de la posición especificada (`srcOff`), hasta la posición (`destOff`) especificada del array de destino (`dest`) de forma no atómica. `length` indica el número de bytes a copiar. Este método no utiliza la facilidad de transacción durante la operación de copia aunque una transacción esté en progreso. Este método es deseable para usarlo solamente cuando los contenidos del array de destino se puedan dejar en un estado de modificación parcial si ocurre una pérdida de energía en mitad de la operación de copia. El método devuelve `destOff+length`.

`public static final native short arrayFillNonAtomic(byte[] bArray, short bOff, short bLen, byte bValue)` : Este método llena el array (`bArray`) de bytes (de forma no atómica) empezando en la posición especificada `bOff`. Llena un número `bLen` de bytes con el valor `bValue`. Este método no utiliza la facilidad de transacción durante la operación de llenado aunque una transacción esté en progreso. De este modo, el método es deseable para usarlo solamente cuando los contenidos del array se puedan dejar en un estado de modificación parcial si ocurre una pérdida de energía en mitad de la operación de relleno. El método devuelve `bOff+bLen`.

`public static final short getShort(byte[] bArray, short bOff)` : Concatena dos bytes para formar un valor del tipo short (que es el que devuelve el método). `barray` es el array de bytes donde se encuentran los dos bytes. `bOff` es la posición donde se encuentra el primero de los bytes (el byte de mayor orden).

`public static final short makeShort(byte b1, byte b2)` : Concatena los dos parámetros del tipo byte para formar un valor del tipo short (que es el que devuelve la función).

`public static final native short setShort(byte[] bArray, short bOff, short sValue)` : Deposita el valor de tipo short (`sValue`) en dos bytes sucesivos a partir del offset especificado (`bOff`) pertenecientes al array de bytes especificado (`bArray`). El método devuelve el valor `bOff+2`. Si el array de bytes es persistente, esta operación se realiza de forma atómica. Si se excede la capacidad del "commit buffer", la operación no se realiza y se lanza una `TransactionException`.

11.3 EXCEPCIONES

11.3.1 EXCEPCIÓN `APDUEXCEPTION`

`public class APDUException extends CardRuntimeException` : La `APDUException` representa una excepción relacionada con la clase `APDU`.

11.3.1.1 Campos

`public static final short BAD_LENGTH` : El método `APDU.setOutgoingLength()` utiliza este código de causa para indicar que el parámetro de longitud es mayor de 256 ó si se solicita una transferencia de datos no encadenados y `len` es mayor que IFSD-2 (donde el IFSD es el tamaño de bloque saliente).

`public static final short BUFFER_BOUNDS` : El método `APDU.sendBytes()` utiliza este código de causa para indicar que la suma del parámetro del offset del buffer y el parámetro de longitud excede el tamaño del buffer APDU.

`public static final short ILLEGAL_USE` : Este código de causa de una `APDUException` indica que el método no se debería invocar, basándose en el estado actual de la APDU.

`public static final short IO_ERROR` : Este código de causa indica que ocurrió un error irrecuperable en el nivel E/S de transmisión.

`public static final short NO_T0_GETRESPONSE` : Este código de causa indica que durante una transmisión con el protocolo T=0, el CAD no devolvió un comando GET RESPONSE en respuesta a una palabra de estado del tipo 61XX para enviar datos adicionales. La transferencia de datos hacia el CAD se ha interrumpido. No se pueden enviar más datos o estados al CAD en ese método `APDU.process()`.

`public static final short T1_IFD_ABORT` : Este código de causa indica que durante la transmisión con el protocolo T=1, el CAD devolvió un comando ABORT S-Block y abortó la transferencia de datos. Se han interrumpido las transferencias entrante y saliente. No se pueden recibir más datos del CAD. Tampoco se pueden enviar datos o palabras de estado al CAD en ese método `APDU.process()`:

11.3.1.2 Constructores

`public APDUException(short reason)` : Construye una `APDUException`. Para ahorrar recursos utilice `throwIt()` para usar la instancia de esta clase perteneciente al JCRE. `reason` contiene la causa de la excepción.

11.3.1.3 Métodos

`public short getReason()` : Con este método se obtiene el código de causa de la excepción.

`public void public setReason()` : Cambia el código de causa. `reason` contiene la causa de la excepción.

`public static void throwIt(short reason)` : Lanza la instancia de `APDUException` perteneciente al JCRE con el código de causa especificado.

11.3.2 EXCEPCIÓN `CardException`

`public class CardException extends Exception` : La clase `CardException` define un campo `reason` y dos métodos a los que se accede: `getReason()` y `setReason()`. El campo `reason` encapsula el identificador de la causa de la excepción en Java Card. Todas las clases de excepción con comprobación de Java Card deberían extender `CardException`. Esta clase también provee un mecanismo de ahorro de recursos (el método `throwIt()`) para usar la instancia de esta clase, perteneciente al JCRE.

11.3.2.1 Constructores

`public CardException(short reason)` : Construye una instancia de `CardException` con la causa especificada. Para ahorrar recursos, utilice el método `throwIt()` para usar la instancia de esta clase perteneciente al JCRE. `reason` contiene la causa de la excepción.

11.3.2.2 Métodos

`public short getReason()` : Consigue el código de causa.

`public void setReason(short reason)` : Cambia el código de causa. `reason` contiene la causa de la excepción.

`public static void throwIt(short reason)` : Lanza la instancia de la clase `CardException` perteneciente al JCRE, con la causa especificada en `reason`.

11.3.3 EXCEPCIÓN `CardRuntimeException`

`public class CardRuntimeException extends RuntimeException` : La clase `CardRuntimeException` define un campo `reason` y dos métodos de acceso: `getReason()` y `setReason()`. El campo `reason` encapsula el identificador de la causa de la excepción en Java Card. Todas las clases de excepción sin comprobación de Java Card deberían extender `CardRuntimeException`. Esta clase también provee un mecanismo de ahorro de recursos (el método `throwIt()`) para usar la instancia de esta clase, perteneciente al JCRE.

11.3.3.1 Constructores

`public CardRuntimeException(short reason)` : Construye una instancia de `CardRuntimeException` con la causa especificada. Para ahorrar recursos, utilice el método `throwIt()` para usar la instancia de esta clase perteneciente al JCRE. `reason` contiene la causa de la excepción.

11.3.3.2 Métodos

`public short getReason()` : Consigue el código de causa.

`public void setReason(short reason)` : Cambia el código de causa. `reason` contiene la causa de la excepción.

`public static void throwIt(short reason)` : Lanza la instancia de la clase `CardException` perteneciente al JCRE, con la causa especificada en `reason`.

11.3.4 EXCEPCIÓN `ISOEXCEPTION`

`public class ISOException extends CardRuntimeException` : La clase `ISOException` encapsula una palabra de estado de respuesta ISO 7816-4 como su código de causa. La clase APDU lanza instancias de `ISOException` pertenecientes al JCRE.

11.3.4.1 Constructores

`public ISOException(short sw)` : Construye una instancia de `ISOException` con la palabra de estado especificada. Para ahorrar recursos, utilice el método `throwIt()` para usar la instancia de esta clase perteneciente al JCRE. `sw` contiene la palabra de estado definida en la ISO 7816-4.

11.3.4.2 Métodos

`public short getReason()` : Consigue la palabra de estado.

`public void setReason(short sw)` : Cambia la palabra de estado. `sw` contiene la palabra de estado definida en la ISO 7816-4.

`public static void throwIt(short sw)` : Lanza la instancia de la clase `CardException` perteneciente al JCRE, con la palabra de estado dada en `sw`.

11.3.5 EXCEPCIÓN `PINEXCEPTION`

`public class PINException extends CardRuntimeException` : La excepción `PINException` representa una excepción relacionada con el acceso a la clase `OwnerPIN`. La clase `OwnerPIN` lanza instancias de `PINException` pertenecientes al JCRE.

11.3.5.1 Campos

`public static short ILLEGAL_VALUE` : Este código de causa se usa para indicar que uno o más parámetros de entrada están fuera de los límites permitidos.

11.3.5.2 Constructores

`public PINException(short reason)` : Construye una `PINException`. Para ahorrar recursos, utilice el método `throwIt()` para usar la instancia de esta clase perteneciente al JCRE. `reason` contiene la causa de la excepción.

11.3.5.3 Métodos

`public static void throwIt(short reason)` : Lanza la instancia de la clase `PINException` perteneciente al JCRE, con la causa especificada en `reason`.

11.3.6 EXCEPCIÓN `SYSTEMEXCEPTION`

`public class SystemException extends CardRuntimeException` : La excepción `JCSystemException` representa una excepción relacionada con la clase `JCSystem`. También la lanzan los métodos `javacard.framework.Applet.register()` y los constructores de la clase `AID`. Estas clases de API's lanzan instancias de `SystemException` pertenecientes al JCRE.

11.3.6.1 Campos

`public static final ILLEGAL_AID` : Este código de causa lo utiliza el método `javacard.framework.Applet.register()` para indicar que el parámetro de entrada AID no es un valor de AID legal.

`public static final short ILLEGAL_TRANSIENT` : Se usa este código de causa para indicar que la solicitud de crear un objeto transitorio no se permite en el contexto del applet actual.

`public static final short ILLEGAL_VALUE` : Este código de causa indica que uno o más parámetros están fuera de los límites permitidos.

`public static final short NO_RESOURCE` : Este código de causa se usa para indicar que no hay recursos suficientes en la tarjeta para atender la solicitud. Por ejemplo, la máquina virtual de Java Card quizás lance este código de excepción cuando no haya suficiente espacio para crear una nueva instancia.

`public static final short NO_TRANSIENT_SPACE` : Los métodos `makeTransient...()` utilizan este código de causa para indicar que no hay suficiente espacio en la memoria volátil para crear el objeto solicitado.

11.3.6.2 Constructores

`public SystemException(short reason)` : Construye una `SystemException`. Para ahorrar recursos utilice el método `throwIt()` para usar la instancia de esta clase perteneciente al JCRE. `reason` contiene la causa de la excepción.

11.3.6.3 Métodos

`public static void throwIt(short reason)` : Lanza una instancia de `SystemException` perteneciente al JCRE con la causa (`reason`) especificada.

11.3.7 EXCEPCIÓN TRANSACTIONEXCEPTION

`public class TransactionException extends CardRuntimeException` : `TransactionException` representa una excepción en el subsistema de transacción. Los métodos que hacen referencia a esta clase están en la clase `JCSystem`. La clase `JCSystem` y la facilidad de transacción, lanzan instancias de la `TransactionException` pertenecientes al JCRE.

11.3.7.1 Campos

`public static final short BUFFER_FULL` : Se usa este código de causa durante una transacción para indicar que el "commit buffer" está lleno.

`public static final short IN_PROGRESS` : El método `beginTransaction` utiliza este código de causa para indicar que una transacción está todavía en progreso.

`public static final short INTERNAL_FAILURE` : Este código de causa se usa durante una transacción para indicar un problema interior del JCRE (un error fatal).

`public static final short NOT_IN_PROGRESS` : Los métodos `abortTransaction` y `commitTransaction` utilizan este código de causa cuando una transacción no está en progreso.

11.3.7.2 Constructores

`public TransactionException(short reason)` : Construye una `TransactionException` con la causa especificada (`reason`). Para ahorrar recursos utilice `throwIt()` para usar la instancia de esta clase perteneciente al JCRE.

11.3.7.3 Métodos

`public static void throwIt(short reason)` : Lanza una instancia de `TransactionException` perteneciente al JCRE con la causa (`reason`) especificada.

11.3.8 EXCEPCIÓN USEREXCEPTION

`public calss UserException extends CardException` : UserException representa una excepción de usuario. Esta clase también provee un mecanismo de ahorro de recursos (el método `throwIt()`). Este método consiste en el uso de la instancia perteneciente al JCRE para las excepciones de usurario.

11.3.8.1 Constructores

`public UserException()` : Construye una `UserException` con la causa = 0. Para ahorrar recursos utilice `throwIt()` para usar la instancia de esta clase perteneciente al JCRE.

`public UserException(short reason)` : Construye una `UserException` con la causa especificada en el parámetro `reason`. Para ahorrar recursos utilice `throwIt()` para usar la instancia de esta clase perteneciente al JCRE.

11.3.8.2 Métodos

`public static void throwIt(short reason)` : Lanza una instancia de `UserException` perteneciente al JCRE con la causa (`reason`) especificada.

12 APÉNDICE 3: CONTENIDO DEL PAQUETE JAVACARD.SECURITY

12.1 INTERFACES

12.1.1 INTERFAZ DESKEY

`public interface DESKey extends SecretKey` : `DESKey` contiene una clave de 8/16/24 bytes para las operaciones de triple DES. Cuando los datos de la clave están ajustados, la clave está inicializada y lista pasa usarse.

12.1.1.1 Métodos

`public byte getKey(byte[] keyData, short kOff)` : Devuelve los datos de `Key` en texto plano de `Key`. La longitud de los datos de la clave es de 8 bytes para DES, 16 bytes para triple DES con 2 claves y 24 bytes para triple DES con 3 claves. El formato de los datos es big-endian y alineados a la derecha (el bit menos significante es el bit menos significante del último byte). `keyData` es un array de bytes donde se devuelven los datos de la clave a partir de la posición dada por `kOff`. El método devuelve la longitud (en bytes) de los datos de la clave devueltos.

`public void setKey(byte[] keyData, short kOff)` : Ajusta los datos de `Key`. La longitud del texto plano de los datos de entrada de la clave es de 8 bytes para DES, 16 bytes para triple DES con 2 claves y 24 bytes para triple DES con 3 claves. El formato de los datos es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte. Los datos de entrada de la clave se copian en el interior de la representación interna. `keyData` es un array que contiene los datos de inicialización de la clave a partir de la posición dada por `kOff`. Si el objeto clave que implementa la interfaz `javacardx.crypto.KeyEncryption` y el objeto `Cipher` especificado a través de `setKeyCipher()` no es nulo.

12.1.2 INTERFAZ DSAKEY

`public interface DSAKey` : La interfaz `DSAKey` es la interfaz base para las implementaciones de claves privadas y públicas para los algoritmos DSA. La implementación de una clave privada DSA debe implementar también los métodos de la interfaz `DSAPrivateKey`. La implementación de una clave pública DSA debe implementar también los métodos de la interfaz `DSAPublicKey`. Cuando se ajustan los cuatro componentes de la clave (X ó Y, P, Q, G), la clave está inicializada y lista para su uso.

12.1.2.1 Métodos

`public short getG(byte[] buffer, short offset)` : Devuelve el parámetro por debajo del principal de la clave en texto plano. El formato de los datos es big-endian y alineados a la derecha (el bit menos significativo es el bit menos

significativo del último byte). `buffer` es el buffer de salida y `offset` da la posición del buffer a partir de la cual empieza el valor del parámetro por debajo del principal. El método devuelve la longitud (en bytes) del parámetro por debajo del principal devuelto.

`public short getP(byte[] buffer, short offset)` : Devuelve el valor del parámetro base de la clave en texto plano. El formato de los datos es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). `buffer` es el buffer de salida y `offset` da la posición a partir de la cual empieza el valor del parámetro base. El método devuelve la longitud (en bytes) del parámetro base devuelto.

`public short getQ(byte[] buffer, short offset)` : Devuelve el valor del parámetro principal de la clave en texto plano. El formato de los datos es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). `buffer` es el buffer de salida y `offset` da la posición a partir de la cual empieza el valor del parámetro principal. El método devuelve la longitud (en bytes) del parámetro principal devuelto.

`public void setG(byte[] buffer, short offset, short length)` : Ajusta el valor del parámetro por debajo del principal de la clave. El formato de los datos en texto plano es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte).El parámetro por debajo del principal se copia en la representación interna. `buffer` es el buffer de entrada y `offset` da la posición del buffer a partir de la cual empieza el valor del parámetro por debajo del principal. `length` es la longitud (en bytes) del parámetro por debajo del principal. Si el objeto de clave implementa la interfaz `javacardx.crypto.KeyEncryption` y el objeto `Cipher` especificado a través de `setKeyCipher()` no es `null`, el valor del parámetro por debajo del principal se decodifica usando el objeto `Cipher`.

`public void setP(byte[] buffer, short offset, short length)` : Ajusta el valor del parámetro base de la clave. El formato de los datos en texto plano es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte).El parámetro base se copia en la representación interna. `buffer` es el buffer de entrada y `offset` da la posición del buffer a partir de la cual empieza el valor del parámetro base. `length` es la longitud (en bytes) del parámetro base. Si el objeto de clave implementa la interfaz `javacardx.crypto.KeyEncryption` y el objeto `Cipher` especificado a través de `setKeyCipher()` no es `null`, el valor del parámetro base se decodifica usando el objeto `Cipher`.

`public void setQ(byte[] buffer, short offset, short length)` : Ajusta el valor del parámetro principal de la clave. El formato de los datos en texto plano es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte).El parámetro principal se copia en la representación interna. `buffer` es el buffer de entrada y `offset` da la posición del buffer a partir de la cual empieza el valor del parámetro principal. `length` es la longitud (en bytes) del parámetro principal. Si el objeto de clave implementa la interfaz `javacardx.crypto.KeyEncryption` y el objeto `Cipher` especificado a través de `setKeyCipher()` no es `null`, el valor del parámetro principal se decodifica usando el objeto `Cipher`.

12.1.3 INTERFAZ DSAPRIVATEKEY

`public interface DSAPrivateKey extends PrivateKey, DSAKey` : Se usa la interfaz `DSAPrivateKey` para firmar datos usando el algoritmo DSA. Una implementación de la interfaz `DSAPrivateKey` debe implementar también los métodos de la interfaz `DSAKey`. Cuando se ajustan los cuatro componentes de la clave (X, P, Q, G), la clave se inicializa y está preparada para su uso.

12.1.3.1 Métodos

`public short getX(byte[] buffer, short offset)` : Devuelve el valor de la clave en texto plano. El formato de los datos es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). `buffer` es el buffer de salida y `offset` da la posición del buffer a partir de la cual empieza el valor de la clave. El método devuelve la longitud del valor de la clave devuelta.

`public void setX(byte[] buffer, short offset, short length)` : Ajusta el valor de la clave. Cuando se inicializan los parámetro base, principal y por debajo del principal, y el valor de la clave se ajusta, la clave está lista para usarse. El formato de los datos en texto plano es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). Los datos de entrada de la clave se copian en la representación interna. `buffer` es el buffer de entrada y `offset` da la posición del buffer a partir de la cual empieza el valor del módulo. `length` es la longitud (en bytes) del módulo. Si el objeto de clave implementa la interfaz `javacardx.crypto.KeyEncryption` y el objeto `Cipher` especificado a través de `setKeyCipher()` no es `null`, el valor del parámetro principal se decodifica usando el objeto `Cipher`.

12.1.4 INTERFAZ DSAPUBLICKEY

`public interface DSAPublicKey extends PublicKey, DSAKey` : Se usa la interfaz `DSAPublicKey` para firmar datos usando el algoritmo DSA. Una implementación de la interfaz `DSAPublicKey` debe implementar también los métodos de la interfaz `DSAKey`. Cuando se ajustan los cuatro componentes de la clave (Y, P, Q, G), la clave se inicializa y está preparada para su uso.

12.1.4.1 Métodos

`public short getY(byte[] buffer, short offset)` : Devuelve el valor de la clave en texto plano. El formato de los datos es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). `buffer` es el buffer de salida y `offset` da la posición del buffer a partir de la cual empieza el valor de la clave. El método devuelve la longitud del valor de la clave devuelta.

`public void setY(byte[] buffer, short offset, short length)` : Ajusta el valor de la clave. Cuando se inicializan el parámetro base, principal y por debajo del principal, y el valor de la clave se ajusta, la clave está lista para usarse. El

formato de los datos en texto plano es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). Los datos de entrada de la clave se copian en la representación interna. `buffer` es el buffer de entrada y `offset` da la posición del buffer a partir de la cual empieza el valor de la clave. `length` es la longitud (en bytes) del valor de la clave. Si el objeto de clave implementa la interfaz `javacardx.crypto.KeyEncryption` y el objeto `Cipher` especificado a través de `setKeyCipher()` no es `null`, el valor del parámetro principal se decodifica usando el objeto `Cipher`.

12.1.5 INTERFAZ KEY

`public interface Key` : La interfaz `Key` es la interfaz base para todas las claves.

12.1.5.1 Métodos

`public void clearKey()` : Borra la clave y ajusta su estado de inicialización a `false`.

`public short getSize()` : Devuelve el tamaño de la clave en número de bits.

`public byte getType()` : Devuelve el tipo de la interfaz de clave.

`public boolean isInitialized()` : Informa acerca del estado de inicialización de la clave (`true` si la clave se ha inicializado). La clave se debe inicializar antes de usarla. Un objeto `Key` ajusta su estado de inicialización a `true` cuando todos los métodos de ajuste asociados se han invocado por lo menos una vez desde que el estado de inicialización estuviera ajustado a `false`. Cuando se crea un objeto `Key`, el objeto ajusta su estado de inicialización a `false`. La invocación del método `clearKey()` ajusta el estado de inicialización a `false`. Una clave con datos de clave transitorios, ajusta su estado de inicialización a `false` según los eventos de borrado asociados.

12.1.6 INTERFAZ PRIVATEKEY

`public interface PrivateKey extends Key` : La clase `PrivateKey` es la clase base para las claves privadas usadas en algoritmos asimétricos.

12.1.7 INTERFAZ PUBLICKEY

`public interface PublicKey extends Key` : La clase `PublicKey` es la clase base para las claves públicas usadas en algoritmos asimétricos.

12.1.8 INTERFAZ RSAPRIVATECRTKEY

`public interface RSAPrivateCrtKey extends PrivateKey` : La interfaz `RSAPrivateCrtKey` se usa para firmar datos usando el algoritmo en la forma

Chinese Remainder Theorem. La clase `javacardx.crypto.Cipher` quizás también la use para codificar o decodificar mensajes. Haciendo $S=m^d \bmod n$, donde m es el dato que se quiere firmar, d es el exponente de la clave privada, y n es el módulo de la clave privada que consta de dos números primos p y q. Los siguientes nombres se usan en los métodos de inicialización de esta interfaz:

P, el factor principal p.

Q, el factor principal q.

$PQ = q^{-1} \bmod p$.

DP1 = d mod (p-1).

DQ1 = d mod (q-1).

Cuando se ajustan todos los componentes de la clave (P, Q, PQ, DP1, DQ1), la clave se inicializa y está lista para usarse.

12.1.8.1 Métodos

`public short getDP1(byte[] buffer, short offset)` : Devuelve el valor del parámetro DP1 en texto plano. El formato de los datos es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). `buffer` es el buffer de salida y `offset` da la posición del buffer a partir de la cual empieza el valor del parámetro. El método devuelve la longitud del valor del parámetro DP1 devuelto.

`public short getDQ1(byte[] buffer, short offset)` : Devuelve el valor del parámetro DQ1 en texto plano. El formato de los datos es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). `buffer` es el buffer de salida y `offset` da la posición del buffer a partir de la cual empieza el valor del parámetro. El método devuelve la longitud del valor del parámetro DQ1 devuelto.

`public short getP(byte[] buffer, short offset)` : Devuelve el valor del parámetro P en texto plano. El formato de los datos es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). `buffer` es el buffer de salida y `offset` da la posición del buffer a partir de la cual empieza el valor del parámetro. El método devuelve la longitud del valor del parámetro P devuelto.

`public short getPQ(byte[] buffer, short offset)` : Devuelve el valor del parámetro PQ en texto plano. El formato de los datos es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). `buffer` es el buffer de salida y `offset` da la posición del buffer a partir de la cual empieza el valor del parámetro. El método devuelve la longitud del valor del parámetro PQ devuelto.

`public short getQ(byte[] buffer, short offset)` : Devuelve el valor del parámetro Q en texto plano. El formato de los datos es big-endian y alineados

a la derecha (el bit menos significativo es el bit menos significativo del último byte). `buffer` es el buffer de salida y `offset` da la posición del buffer a partir de la cual empieza el valor del parámetro. El método devuelve la longitud del valor del parámetro Q devuelto.

`public void setDP1(byte[] buffer, short offset, short length)` : Ajusta el valor del parámetro DP1. El formato de los datos en texto plano es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). Los datos de entrada del parámetro DP1 se copian en la representación interna. `buffer` es el buffer de entrada y `offset` da la posición del buffer a partir de la cual empieza el valor del parámetro. `length` es la longitud (en bytes) del parámetro. Si el objeto de clave implementa la interfaz `javacardx.crypto.KeyEncryption` y el objeto `Cipher` especificado a través de `setKeyCipher()` no es `null`, el valor del parámetro DP1 se decodifica usando el objeto `Cipher`.

`public void setDQ1(byte[] buffer, short offset, short length)` : Ajusta el valor del parámetro DP1. El formato de los datos en texto plano es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). Los datos de entrada del parámetro DQ1 se copian en la representación interna. `buffer` es el buffer de entrada y `offset` da la posición del buffer a partir de la cual empieza el valor del parámetro. `length` es la longitud (en bytes) del parámetro. Si el objeto de clave implementa la interfaz `javacardx.crypto.KeyEncryption` y el objeto `Cipher` especificado a través de `setKeyCipher()` no es `null`, el valor del parámetro DQ1 se decodifica usando el objeto `Cipher`.

`public void setP(byte[] buffer, short offset, short length)` : Ajusta el valor del parámetro P. El formato de los datos en texto plano es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). Los datos de entrada del parámetro P se copian en la representación interna. `buffer` es el buffer de entrada y `offset` da la posición del buffer a partir de la cual empieza el valor del parámetro. `length` es la longitud (en bytes) del parámetro. Si el objeto de clave implementa la interfaz `javacardx.crypto.KeyEncryption` y el objeto `Cipher` especificado a través de `setKeyCipher()` no es `null`, el valor del parámetro P se decodifica usando el objeto `Cipher`.

`public void setPQ(byte[] buffer, short offset, short length)` : Ajusta el valor del parámetro PQ. El formato de los datos en texto plano es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). Los datos de entrada del parámetro PQ se copian en la representación interna. `buffer` es el buffer de entrada y `offset` da la posición del buffer a partir de la cual empieza el valor del parámetro. `length` es la longitud (en bytes) del parámetro. Si el objeto de clave implementa la interfaz `javacardx.crypto.KeyEncryption` y el objeto `Cipher` especificado a través de `setKeyCipher()` no es `null`, el valor del parámetro PQ se decodifica usando el objeto `Cipher`.

`public void setQ(byte[] buffer, short offset, short length)` : Ajusta el valor del parámetro Q. El formato de los datos en texto plano es big-endian y

alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). Los datos de entrada del parámetro Q se copian en la representación interna. `buffer` es el buffer de entrada y `offset` da la posición del buffer a partir de la cual empieza el valor del parámetro. `length` es la longitud (en bytes) del parámetro. Si el objeto de clave implementa la interfaz `javacardx.crypto.KeyEncryption` y el objeto `Cipher` especificado a través de `setKeyCipher()` no es `null`, el valor del parámetro Q se decodifica usando el objeto `Cipher`.

12.1.9 INTERFAZ RSAPRIVATEKEY

`public interface RSAPrivateKey extends PrivateKey` : La interfaz `RSAPrivateKey` se usa para firmar datos usando el algoritmo RSA en la forma módulo/exponente. La clase `javacardx.crypto.Cipher` quizás también la use para codificar o decodificar mensajes. Cuando se ajustan el módulo y el exponente de la clave, la clave se inicializa y está lista para usarse.

12.1.9.1 Métodos

`public short getExponent(byte[] buffer, short offset)` : Devuelve el valor privado del exponente en texto plano. El formato de los datos es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). `buffer` es el buffer de salida y `offset` da la posición del buffer a partir de la cual empieza el valor del exponente. El método devuelve la longitud del valor privado del exponente devuelto.

`public short getModulus(byte[] buffer, short offset)` : Devuelve el valor del módulo en texto plano. El formato de los datos es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). `buffer` es el buffer de salida y `offset` da la posición del buffer a partir de la cual empieza el valor del módulo. El método devuelve la longitud del valor del módulo devuelto.

`public void setExponent(byte[] buffer, short offset, short length)` : Ajusta el valor privado del exponente de la clave. El formato de los datos en texto plano es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). Los datos de entrada del exponente se copian en la representación interna. `buffer` es el buffer de entrada y `offset` da la posición del buffer a partir de la cual empieza el valor del exponente. `length` es la longitud (en bytes) del parámetro. Si el objeto de clave implementa la interfaz `javacardx.crypto.KeyEncryption` y el objeto `Cipher` especificado a través de `setKeyCipher()` no es `null`, el valor del exponente se decodifica usando el objeto `Cipher`.

`public void setModulus(byte[] buffer, short offset, short length)` : Ajusta el valor del módulo de la clave. El formato de los datos en texto plano es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). Los datos de entrada del módulo se copian en la representación interna. `buffer` es el buffer de entrada y `offset` da la posición del buffer a partir de la cual empieza el valor del módulo. `length` es la longitud (en bytes)

del parámetro. Si el objeto de clave implementa la interfaz `javacardx.crypto.KeyEncryption` y el objeto `Cipher` especificado a través de `setKeyCipher()` no es `null`, el valor del módulo se decodifica usando el objeto `Cipher`.

12.1.10INTERFAZ RSAPUBLICKEY

`public interface RSAPublicKey extends PublicKey` : La interfaz `RSAPublicKey` se utiliza para verificar firmas en datos firmados usando el algoritmo RSA. La clase `javacardx.crypto.Cipher` quizás también la use para codificar o decodificar mensajes. Cuando se ajustan el módulo y el exponente de la clave, la clave se inicializa y está lista para usarse.

12.1.10.1 Métodos

`public short getExponent(byte[] buffer, short offset)` : Devuelve el valor público del exponente en texto plano. El formato de los datos es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). `buffer` es el buffer de salida y `offset` da la posición del buffer a partir de la cual empieza el valor del exponente. El método devuelve la longitud del valor público del exponente devuelto.

`public short getModulus(byte[] buffer, short offset)` : Devuelve el valor del módulo en texto plano. El formato de los datos es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). `buffer` es el buffer de salida y `offset` da la posición del buffer a partir de la cual empieza el valor del módulo. El método devuelve la longitud del valor del módulo devuelto.

`public void setExponent(byte[] buffer, short offset, short length)` : Ajusta el valor público del exponente de la clave. El formato de los datos en texto plano es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). Los datos de entrada del exponente se copian en la representación interna. `buffer` es el buffer de entrada y `offset` da la posición del buffer a partir de la cual empieza el valor del exponente. `length` es la longitud (en bytes) del parámetro. Si el objeto de clave implementa la interfaz `javacardx.crypto.KeyEncryption` y el objeto `Cipher` especificado a través de `setKeyCipher()` no es `null`, el valor del exponente se decodifica usando el objeto `Cipher`.

`public void setModulus(byte[] buffer, short offset, short length)` : Ajusta el valor del módulo de la clave. El formato de los datos en texto plano es big-endian y alineados a la derecha (el bit menos significativo es el bit menos significativo del último byte). Los datos de entrada del módulo se copian en la representación interna. `buffer` es el buffer de entrada y `offset` da la posición del buffer a partir de la cual empieza el valor del módulo. `length` es la longitud (en bytes) del parámetro. Si el objeto de clave implementa la interfaz `javacardx.crypto.KeyEncryption` y el objeto `Cipher` especificado a través de

`setKeyCipher()` no es `null`, el valor del módulo se decodifica usando el objeto `Cipher`.

12.1.11INTERFAZ SECRETKEY

`public interface SecretKey extends Key` : La clase `SecretKey` es la interfaz base para las claves usadas en algoritmos simétricos.

12.2 CLASES

12.2.1 CLASE KEYBUILDER

`public class KeyBuilder` : La clase `KeyBuilder` es una fábrica de objetos de claves.

12.2.1.1 Campos

`public static final short LENGTH_DES` : Longitud de la clave DES. `LENGTH_DES` = 64.

`public static final short LENGTH_DES3_2KEY` : Longitud de la clave DES. `LENGTH_DES3_2KEY` = 128.

`public static final short LENGTH_DES3_3KEY` : Longitud de la clave DES. `LENGTH_DES3_3KEY` = 192.

`public static final short LENGTH_DSA_1024` : Longitud de la clave DSA. `LENGTH_DSA_1024` = 1024.

`public static final short LENGTH_DSA_512` : Longitud de la clave DSA. `LENGTH_DSA_512` = 512.

`public static final short LENGTH_DSA_768` : Longitud de la clave DSA. `LENGTH_DSA_768` = 768.

`public static final short LENGTH_RSA_1024` : Longitud de la clave RSA. `LENGTH_RSA_1024` = 1024.

`public static final short LENGTH_RSA_2048` : Longitud de la clave RSA. `LENGTH_RSA_2048` = 2048.

`public static final short LENGTH_RSA_512` : Longitud de la clave RSA. `LENGTH_RSA_512` = 512.

`public static final short LENGTH_RSA_768` : Longitud de la clave RSA. `LENGTH_RSA_768` = 768.

`public static final byte TYPE_DES` : Objeto de clave que implementa el tipo de interfaz `DESKey` con datos de clave persistentes.

`public static final byte TYPE_DES_TRANSIENT_DESELECT` : Objeto de clave que implementa el tipo de interfaz `DESKey` con datos de clave transitorios `CLEAR_ON_DESELECT`. Este objeto de clave realiza implícitamente un `clearKey()` en el encendido, en un reset de la tarjeta y en la deselección del applet.

`public static final byte TYPE_DES_TRANSIENT_RESET` : Objeto de clave que implementa el tipo de interfaz `DESKey` con datos de clave transitorios `CLEAR_ON_RESET`. Este objeto de clave realiza implícitamente un `clearKey()` en el encendido o en un reset de la tarjeta.

`public static final byte TYPE_DSA_PRIVATE` : Objeto de clave que implementa el tipo de interfaz `DSAPrivateKey` para el algoritmo DSA.

`public static final byte TYPE_DSA_PUBLIC` : Objeto de clave que implementa el tipo de interfaz `DSAPublicKey` para el algoritmo DSA.

`public static final byte TYPE_RSA_CRT_PRIVATE` : Objeto de clave que implementa el tipo de interfaz `RSAPrivateCrtKey` que usa el Chinese Remainder Theorem.

`public static final byte TYPE_RSA_PRIVATE` : Objeto de clave que implementa el tipo de interfaz `RSAPrivateKey` que usa la forma módulo/exponente.

`public static final byte TYPE_RSA_PUBLIC` : Objeto de clave que implementa el tipo de interfaz `RSAPublicKey`.

12.2.1.2 Métodos

`public static Key buildKey(byte keyType, short keyLength, boolean keyEncryption)` : Crea claves criptográficas para los algoritmos de firma y de cifrado. Este método crea instancias que quizás sean los únicos objetos de claves usados para inicializar las instancias de `Signature` y `Cipher`. Advertir que al objeto devuelto se le debe hacer un cast al tipo apropiado que depende del tipo de interfaz de clave. `keyType` es el tipo de clave generada (los códigos válidos son las constantes que empiezan por `TYPE...`) y `keyLength` es el tamaño de clave en bits (las longitudes dependen del tipo de clave). si `keyEncryption` es true solicita la implementación de clave que implemente la interfaz `javacardx.cipher.KeyEncryption`. El método devuelve la instancia de la clave solicitada.

12.2.2 CLASE `MESSAGEDIGEST`

`public abstract class MessageDigest` : La clase `MessageDigest` es la clase base para los algoritmos de dispersión. Las implementaciones de los algoritmos deben extender esta clase e implementar todos los métodos abstractos.

12.2.2.1 Campos

`public static final byte ALG_MD5` : Algoritmo MD5 para asimilación de mensajes.

`public static final byte ALG_RIPEMD160` : Algoritmo RIPE MD-160 para asimilación de mensajes.

`public static final byte ALG_SHA` : Algoritmo SHA para asimilación de mensajes.

12.2.2.2 Constructores

`protected MessageDigest()` : Constructor protegido.

12.2.2.3 Métodos

`public abstract short doFinal(byte[] inBuff, short inOffset, short inLength, byte[] outBuff, short outOffset)` : Genera una dispersión para todos los datos de entrada. Completa y devuelve el cálculo de la dispersión después de realizar operaciones finales como el relleno. El objeto `MessageDigest` se resetea después de que se realice la llamada. Quizás los datos de entrada y de salida del buffer se solapen. `inBuff` es el buffer de datos a los que se le calcula la dispersión (sólo los `inLength` bytes del buffer) e `inOffset` es la posición del buffer de entrada a partir de la cual empieza la generación de la dispersión. El parámetro `outBuff` es el buffer de salida , que quizás sea el mismo que el buffer de entrada. `outOffset` es la posición del buffer de salida a partir de la cual se guarda el valor del resultado de la dispersión. El método devuelve el número de bytes originados (resultado de la dispersión) en el buffer de salida.

`public abstract byte getAlgorithm()` : Indica el código de algoritmo de asimilación de mensajes empleado.

`public static final MessageDigest getInstance(byte algorithm, boolean externalAccess)` : Crea la instancia de un objeto `MessageDigest` con el algoritmo especificado. El parámetro `algorithm` selecciona el algoritmo de asimilación de mensajes deseado (los códigos válidos son los que empiezan por `ALG_...`y que se encuentran en la sección Campos). Si `externalAccess` tiene el valor `true` indica que la instancia `MessageDigest` se compartirá entre múltiples instancias de applets y que se accederá (a través de una interfaz `Shareable`) a la instancia de `MessageDigest` cuando el propietario de la instancia no sea el applet actualmente seleccionado. El método devuelve la instancia del objeto `MessageDigest` del algoritmo solicitado.

`public abstrac byte getLength()` : Devuelve la longitud en bytes de la dispersión.

`public abstract void reset()` : Resetea el objeto `MessageDigest` al estado inicial para un uso futuro.

`public abstract void update(byte[] inBuff, short inOffset, short inLength)` : Acumula la dispersión de los datos de entrada dados en el buffer `inBuff` y a partir de la posición `inOffset`. Cuando se usa este método se solicita almacenamiento temporal para resultados intermedios. Este método solo se debería usar si todos los datos (cuya longitud la da `inLength`) de entrada solicitados por la

dispersión no están disponibles en un solo array de bytes. Se recomienda el método `doFinal()` siempre que sea posible.

12.2.3 CLASE `RANDOMDATA`

`public abstract class RandomData` : La clase abstracta `RandomData` es la clase base para la generación de números aleatorios. Las implementaciones de los algoritmos de `RandomData` deben extender esta clase e implementar todos los métodos abstractos.

12.2.3.1 Campos

`public static final byte ALG_PSEUDO_RANDOM` : Algoritmos de generación de números pseudoaleatorios.

`public static final byte ALG_SECURE_RANDOM` : Algoritmos de generación de números aleatorios criptográficamente seguros.

12.2.3.2 Constructores

`protected RandomData()` : Constructor protegido.

12.2.3.3 Métodos

`public abstract void generateData(byte[] buffer, short offset, short length)` : Genera datos aleatorios. `buffer` es el buffer de salida y `offset` es la posición del buffer a partir de la cual se encuentra el número aleatorio, cuya longitud se define mediante el parámetro `length`.

`public static final RandomData getInstance(byte algorithm)` : Crea una instancia de `RandomData` del algoritmo seleccionado. La semilla de la instancia de `RandomData` se inicializa a un valor interno por defecto. Con el parámetro `algorithm` se elige el algoritmo de generación de números aleatorios. Los códigos válidos son las constantes que empiezan por `ALG_...` y que se han expuesto en la sección Campos. El método devuelve la instancia del objeto `RandomData` del algoritmo solicitado.

`public abstract void setSeed(byte[] buffer, short offset, short length)` : Introduce una semilla en el generador de números aleatorios. `buffer` es el buffer de entrada y `offset` indica la posición del buffer a partir de la cual se encuentra la semilla. `length` da la longitud (en bytes) de la semilla.

12.2.4 CLASE `SIGNATURE`

`public abstract class Signature` : La clase `Signature` es la clase base para los algoritmos de firma. Las implementaciones de los algoritmos de firmado deben extender esta clase e implementar todos los métodos abstractos. El término de relleno se utiliza en los algoritmos de firma de clave pública para hacer referencia a

todas las operaciones especificadas en el esquema referenciado para transformar la asimilación del mensaje en el tamaño de bloque de codificación.

12.2.4.1 Campos

`public static final byte ALG_DES_MAC4_ISO9797_M1` : El algoritmo de firma `ALG_DES_MAC4_ISO9797_M1` genera una MAC (Message Authentication Code) de 4 bytes (los 4 bytes más significativos de un bloque de codificación) usando DES ó triple DES en el modo CBC. Este algoritmo usa un CBC externo para triple DES. Los datos de entrada se rellenan según el esquema ISO 9797 método1.

`public static final byte ALG_DES_MAC4_ISO9797_M2` : El algoritmo de firma `ALG_DES_MAC4_ISO9797_M2` genera una MAC (Message Authentication Code) de 4 bytes (los 4 bytes más significativos de un bloque de codificación) usando DES ó triple DES en el modo CBC. Este algoritmo usa un CBC externo para triple DES. Los datos de entrada se rellenan según el esquema ISO 9797 método2 (ISO 7816-4, EMV'96).

`public static final byte ALG_DES_MAC4_NOPAD` : El algoritmo de firma `ALG_DES_MAC4_NOPAD` genera una MAC (Message Authentication Code) de 4 bytes (los 4 bytes más significativos de un bloque de codificación) usando DES ó triple DES en el modo CBC. Este algoritmo usa un CBC externo para triple DES. Los datos de entrada no se rellenan. Si los datos de entrada no están alineados en un bloque de 8 bytes, se lanza una excepción `CryptoException` con el código de causa `ILLEGAL_USE`.

`public static final byte ALG_DES_MAC4_PKCS5` : El algoritmo de firma `ALG_DES_MAC4_PKCS5` genera una MAC (Message Authentication Code) de 4 bytes (los 4 bytes más significativos de un bloque de codificación) usando DES ó triple DES en el modo CBC. Este algoritmo usa un CBC externo para triple DES. Los datos de entrada se rellenan según el esquema PKCS#5.

`public static final byte ALG_DES_MAC8_ISO9797_M1` : El algoritmo de firma `ALG_DES_MAC8_ISO9797_M1` genera una MAC (Message Authentication Code) de 8 bytes usando DES ó triple DES en el modo CBC. Este algoritmo usa un CBC externo para triple DES. Los datos de entrada se rellenan según el esquema ISO 9797 método1.

`public static final byte ALG_DES_MAC8_ISO9797_M2` : El algoritmo de firma `ALG_DES_MAC8_ISO9797_M2` genera una MAC (Message Authentication Code) de 8 bytes usando DES ó triple DES en el modo CBC. Este algoritmo usa un CBC externo para triple DES. Los datos de entrada se rellenan según el esquema ISO 9797 método2 (ISO 7816-4, EMV'96).

`public static final byte ALG_DES_MAC8_NOPAD` : El algoritmo de firma `ALG_DES_MAC8_NOPAD` genera una MAC (Message Authentication Code) de 8 bytes usando DES ó triple DES en el modo CBC. Este algoritmo usa un CBC externo para triple DES. Los datos de entrada no se rellenan. Si los datos de entrada no están alineados en un bloque de 8 bytes, se lanza una excepción `CryptoException` con el código de causa `ILLEGAL_USE`.

`public static final byte ALG_DES_MAC8_PKCS5` : El algoritmo de firma `ALG_DES_MAC8_PKCS5` genera una MAC (Message Authentication Code) de 8 bytes usando DES ó triple DES en el modo CBC. Este algoritmo usa un CBC externo para triple DES. Los datos de entrada se rellenan según el esquema PKCS#5.

`public static final byte ALG_DSA_SHA` : El algoritmo de firma `ALG_DSA_SHA` firma/verifica la asimilación SHA de 20 bytes usando DSA.

`public static final byte ALG_RSA_MD5_PKCS1` : El algoritmo `ALG_RSA_MD5_PKCS1` de firma, codifica los 16 bytes de la asimilación MD5 usando RSA. La asimilación se rellena según el esquema PKCS#1 (v1.5).

`public static final byte ALG_RSA_RFC2409` : El algoritmo `ALG_RSA_RFC2409` de firma, codifica los 16 bytes de la asimilación MD5 usando RSA. La asimilación se rellena según el esquema RFC2409.

`public static final byte ALG_RSA_RIPEMD160_ISO9796` : El algoritmo `ALG_ RSA_RIPEMD160_ISO9796` de firma, codifica los 20 bytes de la asimilación RIPE MD-160 usando RSA. La asimilación se rellena según el esquema ISO 9796.

`public static final byte ALG_RSA_RIPEMD160_PKCS1` : El algoritmo `ALG_ RSA_RIPEMD160_PKCS1` de firma, codifica los 20 bytes de la asimilación RIPE MD-160 usando RSA. La asimilación se rellena según el esquema PKCS#1 (v1.5).

`public static final byte ALG_RSA_SHA_ISO9796` : El algoritmo `ALG_ RSA_SHA_ISO9796` de firma, codifica los 20 bytes de la asimilación SHA usando RSA. La asimilación se rellena según el esquema ISO 9796 (EMV'96).

`public static final byte ALG_RSA_SHA_PKCS1` : El algoritmo `ALG_RSA_SHA_PKCS1` de firma, codifica los 20 bytes de la asimilación SHA usando RSA. La asimilación se rellena según el esquema PKCS#1 (v1.5).

`public static final byte ALG_RSA_SHA_RFC2409` : El algoritmo `ALG_RSA_SHA_RFC2409` de firma, codifica los 20 bytes de la asimilación SHA usando RSA. La asimilación se rellena según el esquema RFC2409.

`public static final byte MODE_SIGN` : Se usa en el método `init()` para indicar el modo de firmar la firma.

`public static final byte MODE_VERIFY` : Se usa en el método `init()` para indicar el modo de verificar la firma.

12.2.4.2 Constructores

`protected signature()` : Constructor protegido.

12.2.4.3 Métodos

`public abstract byte getAlgorithm()` : Este método devuelve el código del algoritmo de firma.

`public static final Signature getInstance(byte algorithm, boolean externalAccess)` : Crea una instancia del objeto `Signature` especificándole el algoritmo deseado mediante el parámetro `algorithm`. Si `externalAccess` es `true`, indica que la instancia se compartirá entre varias instancias de applets y que también se accederá (a través de una interfaz `Shareable`) a la instancia `Signature` cuando el propietario de la instancia `Signature` no sea el applet actualmente seleccionado.

`public abstract short getLength()` : Devuelve la longitud (en bytes) de los datos de la firma.

`public abstract void init(Key theKey, byte theMode)` : Inicializa el objeto `Signature` con la clave apropiada. Los algoritmos que no necesitan parámetros de inicialización o usan valores de parámetros por defecto, deberían utilizar este método. Los algoritmos DES y triple DES en el modo CBC, usan 0 para el vector (IV) inicial si se utiliza este método. El parámetro `theKey` es el objeto clave que se usa para la firma o para la verificación. `theMode` puede ser `MODE_SIGN` ó `MODE_VERIFY`.

`public abstract void init(Key theKey, byte theMode, byte[] bArray, short bOff, short bLen)` : Inicializa el objeto `Signature` con la clave apropiada y los parámetros específicos del algoritmo. Los algoritmos DES y triple DES en el modo CBC externo esperan un parámetro con un valor de 8 bytes para el vector (IV) inicial que se encuentra en `bArray`. Los algoritmos RSA y DSA lanzan una `CryptoException.ILLEGAL_VALUE`. El parámetro `theKey` es el objeto clave que se usa para firmar. `theMode` puede tener los valores `MODE_SIGN` ó `MODE_VERIFY`. `bArray` es un array de bytes que contiene información (que empieza en la posición `bOff` del array) de inicialización específica que depende del algoritmo. `bLen` es la longitud (en bytes) de dicha información de inicialización.

`public abstract short sign(byte[] inBuff. short inOffset, short inLength, byte[] sigBuff, short sigOffset)` : Genera la firma de todos/los últimos datos de entrada. Una llamada a este método también resetea este objeto `Signature` al estado en que estaba cuando fue previamente inicializado a través de una llamada a `init()`. Es decir, el objeto está reseteado y disponible para firmar otro mensaje. Los datos del buffer de entrada y de salida se pueden solapar. El parámetro `inBuff` indica el buffer de entrada que contiene los datos que se van a firmar. `inOffset` es la posición de los datos de entrada en el buffer `inBuff`. `inLength` da la longitud (en bytes) de los datos de entrada que se encuentran en `inBuff`. `sigBuff` es el buffer de salida donde el método guarda la firma de los datos, a partir de la posición especificada por `sigOffset`. El método devuelve el número de bytes de la firma que se encuentra en `sigBuff`.

`public abstract void update(byte[] inBuff, short inOffset, short inLength)` : Acumula una firma de los datos de entrada. Cuando se usa este método se solicita almacenamiento temporal para resultados intermedios. Este método

solo se debería usar si los datos de entrada requeridos para la firma, no están disponibles en un solo array de bytes. Se recomienda el uso de los métodos `sign()` ó `verify()` siempre que sea posible.

`public abstract boolean verify(byte[] inBuff. short inOffset, short inLength, byte[] sigBuff, short sigOffset, short sigLength)` : Compara la firma de todos/los últimos datos de entrada con la que se encuentra en la firma. Una llamada a este método también resetea este objeto `Signature` al estado en que estaba cuando fue previamente inicializado a través de una llamada a `init()`. Es decir, el objeto está reseteado y disponible para verificar otro mensaje. El parámetro `inBuff` indica el buffer de entrada que contiene los datos que se van a verificar. `inOffset` es la posición de los datos de entrada en el buffer `inBuff`. `inLength` da la longitud (en bytes) de los datos de entrada que se encuentran en `inBuff`. `sigBuff` es el buffer de entrada donde se encuentran los datos de la firma, a partir de la posición especificada por `sigOffset`. El método devuelve el número de bytes de la firma que se encuentra en `sigBuff`. `sigLength` da la longitud (en bytes) de los datos de la firma que se encuentran en `sigBuff`. El método devuelve `true` si la firma es correcta y `false` en caso contrario.

12.3 EXCEPCIONES

12.3.1 EXCEPCIÓN `CRYPTOEXCEPTION`

`public class CryptoException extends CardRuntimeException` : `CryptoException` representa una excepción relacionada con la criptografía. Las clases de la API lanzan instancias de `SystemException` pertenecientes al JCRE.

12.3.1.1 Campos

`public static final short ILLEGAL_USE` : Este código de causa se usa para indicar que la firma o el algoritmo de cifrado no rellena el mensaje de entrada y el mensaje de entrada no está alineado en bloque.

`public static final short ILLEGAL_VALUE` : Este código de causa se utiliza para indicar que uno o más parámetros están fuera de los límites permitidos.

`public static final short INVALID_INI` : Este código de causa se usa para indicar que la forma o el objeto de cifrado no se ha inicializado correctamente para la operación solicitada.

`public static final short NO_SUCH_ALGORITHM` : Este código de causa se usa para indicar que el algoritmo solicitado o el tipo de clave no se soporta.

`public static final short UNINITIALIZED_KEY` : Este código de causa se utiliza para indicar que la clave no está inicializada.

12.3.1.2 Constructores

`public CryptoException(short reason)` : Construye una `CryptoException` con la causa especificada en el parámetro `reason`. Para ahorrar recursos utilice el método `throwIt()` para usar la instancia de esta clase perteneciente al JCRE.

12.3.1.3 Métodos

`public static void throwIt(short reason)` : Lanza una instancia de `CryptoException` con la causa especificada en `reason`. Esta instancia pertenece al JCRE.

13 APÉNDICE 4: CONTENIDO DEL PAQUETE JAVACARDX.CRYPTO

13.1 INTERFACES

13.1.1 INTERFAZ KEYENCRYPTION

`public interface KeyEncryption` : La interfaz `KeyEncryption` define métodos que se usan para permitir a la implementación de una clave, el acceso a los datos de clave codificados.

13.1.1.1 Métodos

`public Cipher getKeyCipher()` : Devuelve el objeto `Cipher` que se usa para decodificar los datos de clave de entrada y los parámetros de la clave definidos por los métodos de ajuste. Si el método devuelve `null`, indica que la decodificación no se ha realizado.

`public void setKeyCipher(Cipher keyCipher)` : Ajusta el objeto `Cipher` (dado por el parámetro `keyCipher`) que se usa para decodificar los datos de clave de entrada y los parámetros de la clave definidos por los métodos de ajuste. Si `keyCipher` es `null`, indica que no se solicita ninguna decodificación.

13.2 CLASES

13.2.1 CLASE CIPHER

`public abstract class Cipher` : La clase `Cipher` es la clase base abstracta para los algoritmos de cifrado. Las implementaciones de algoritmos de cifrado deben extender esta clase e implementar todos los métodos abstractos. El término relleno se utiliza en los algoritmos de cifrado de clave pública para hacer referencia a todas las operaciones especificadas en el esquema referenciado para transformar el bloque del mensaje en el tamaño de bloque de cifrado.

13.2.1.1 Campos

`public static final byte ALG_DES_CBC_ISO9797_M1` : El algoritmo de cifrado `ALG_DES_CBC_ISO9797_M1` provee un algoritmo de cifrado que usa DES en el modo CBC. Este algoritmo usa un CBC externo para triple DES. Los datos de entrada se rellenan según el esquema ISO 9797 método 1.

`public static final byte ALG_DES_CBC_ISO9797_M2` : El algoritmo de cifrado `ALG_DES_CBC_ISO9797_M2` provee un algoritmo de cifrado que usa DES en el modo CBC. Este algoritmo usa un CBC externo para triple DES. Los datos de entrada se rellenan según el esquema ISO 9797 método 2 (ISO 7816-4, EMV'96).

`public static final byte ALG_DES_CBC_NOPAD` : El algoritmo de cifrado `ALG_DES_CBC_NOPAD` provee un algoritmo de cifrado que usa DES en el modo CBC. Este algoritmo usa un CBC externo para triple DES. Este algoritmo no rellena los datos de entrada. Si los datos de entrada no están alineados en un bloque de 8 bytes, se lanza una excepción `CryptoException` con el código de causa `ILLEGAL_USE`.

`public static final byte ALG_DES_CBC_PKCS5` : El algoritmo de cifrado `ALG_DES_CBC_PKCS5` provee un algoritmo de cifrado que usa DES en el modo CBC. Este algoritmo usa un CBC externo para triple DES. Los datos de entrada se rellenan según el esquema PKCS#5.

`public static final byte ALG_DES_ECB_ISO9797_M1` : El algoritmo de cifrado `ALG_DES_ECB_ISO9797_M1` provee un algoritmo de cifrado que usa DES en el modo ECB. Los datos de entrada se rellenan según el esquema ISO 9797 método 1.

`public static final byte ALG_DES_ECB_ISO9797_M2` : El algoritmo de cifrado `ALG_DES_ECB_ISO9797_M2` provee un algoritmo de cifrado que usa DES en el modo ECB. Los datos de entrada se rellenan según el esquema ISO 9797 método 2 (ISO 7816-4, EMV'96).

`public static final byte ALG_DES_ECB_NOPAD` : El algoritmo de cifrado `ALG_DES_ECB_NOPAD` provee un algoritmo de cifrado que usa DES en el modo ECB. Este algoritmo no rellena los datos de entrada. Si los datos de entrada no están alineados en un bloque de 8 bytes, se lanza una excepción `CryptoException` con el código de causa `ILLEGAL_USE`.

`public static final byte ALG_DES_ECB_PKCS5` : El algoritmo de cifrado `ALG_DES_ECB_PKCS5` provee un algoritmo de cifrado que usa DES en el modo ECB. Los datos de entrada se rellenan según el esquema PKCS#5.

`public static final byte ALG_RSA_ISO14888` : El algoritmo de cifrado `ALG_RSA_ISO14888` provee un algoritmo de cifrado que usa RSA. Los datos de entrada se rellenan según el esquema ISO 14888.

`public static final byte ALG_RSA_ISO9796` : El algoritmo de cifrado `ALG_RSA_ISO9796` provee un algoritmo de cifrado que usa RSA. Los datos de entrada se rellenan según el esquema ISO 9796 (EMV'96). Este algoritmo es solo adecuado para mensajes de longitud limitada. El número total de bytes de entrada procesados deberían ser mayores que k/2, donde k es el tamaño del módulo de la clave RSA en bytes.

`public static final byte ALG_RSA_PKCS1` : El algoritmo de cifrado `ALG_RSA_PKCS1` provee un algoritmo de cifrado que usa RSA. Los datos de entrada se rellenan según el esquema PKCS#1 (v1.5). Este algoritmo es solo adecuado para mensajes de longitud limitada. El número total de bytes de entrada procesados no deberían ser mayores que k-11, donde k es el tamaño del módulo de la clave RSA en bytes.

`public static final byte MODE_DECRYPT` : Se usa en el método `init()` para indicar el modo de decodificación.

`public static final byte MODE_ENCRYPT` : Se usa en el método `init()` para indicar el modo de codificación.

13.2.1.2 Constructores

`protected() Cipher()` : Constructor protegido.

13.2.1.3 Métodos

`public abstract short doFinal(byte[] inBuff, short inOffset, short inLength, byte[] outBuff, short outOffset)` : Genera una salida codificada/decodificada para todos/los últimos datos de entrada. La llamada al método también resetea este objeto `Cypher` al estado en que estaba cuando se inicializó a través de una llamada al método `init()`. Es decir, el objeto se resetea y está disponible para codificar y decodificar más datos (dependiendo del modo de operación que se especificó en la llamada al método `init()`). Quizás los datos de entrada y de salida del buffer se solapen. `inBuff` es el buffer de datos a codificar o decodificar, e `inOffset` es la posición del buffer de entrada a partir de la cual empiezan dichos datos (de `inLength` bytes de longitud). El parámetro `outBuff` es el buffer de salida, que quizás sea el mismo que el buffer de entrada. `outOffset` es la posición del buffer de salida a partir de la cual se guardan los datos codificados/decodificados. El método devuelve el número de bytes originados (resultado de la codificación/decodificación) en el buffer de salida. En las operaciones de decodificación (excepto cuando se usa el relleno del método 1 de la ISO 9797) los bytes de relleno no se escriben en `outBuff`. En las operaciones de codificación, el número de bytes de salida (que se escriben en `outBuff`) quizás sea mayor que `inLength`.

`public abstract byte getAlgorithm()` : Consigue el código de los algoritmos de cifrado que se han presentado anteriormente.

`public static final Cipher getInstance(byte algorithm, boolean externalAccess)` : Este método devuelve la instancia de un objeto `Cipher`. El algoritmo de cifrado se especifica en el parámetro `algorithm`. Si `externalAccess` toma el valor `true`, indica que la instancia se compartirá entre múltiples instancias de applet y que se accederá (a través de una interfaz `Shareable`) a la instancia de `Cipher` cuando el propietario de dicha instancia no sea el applet actualmente seleccionado.

`public abstract void init(Key theKey, byte theMode)` : Inicializa el objeto `Cipher` con la clave apropiada. Los algoritmos que no necesitan parámetros de inicialización o usa valores de parámetros por defecto, deberían utilizar este método. Los algoritmos DES y triple DES en el modo CBC, usan 0 para el vector (IV) inicial si se utiliza este método. El parámetro `theKey` es el objeto clave que se usa para la firma o para la verificación. `theMode` puede ser `MODE_DECRYPT` ó `MODE_ENCRYPT`.

`public abstract void init(Key theKey, byte theMode, byte[] bArray, short bOff, short bLen)` : Inicializa el objeto `Cipher` con la clave apropiada y los parámetros específicos del algoritmo. Los algoritmos DES y triple DES en el modo CBC externo esperan un parámetro con un valor de 8 bytes para el vector (IV) inicial que se encuentra en `bArray`. Los algoritmos RSA y DSA lanzan una `CryptoException.ILLEGAL_VALUE`. El parámetro `theKey` es el objeto clave que se usa para codificar/decodificar. `theMode` puede tener los valores `MODE_DECRYPT` ó `MODE_ENCRYPT`. `bArray` es un array de bytes que contiene información (que empieza en la posición `bOff` del array) de inicialización específica que depende del algoritmo. `bLen` es la longitud (en bytes) de dicha información de inicialización.

`public abstract short update(byte[] inBuff, short inOffset, short inLength, byte[] outBuff, short outOffset)` : Genera una salida codificada/decodificada a partir de los datos de entrada. Cuando se usa este método se solicita almacenamiento temporal para resultados intermedios. Este método solo se debería usar si todos datos de entrada solicitados para el codificador, no están disponibles en un solo array de bytes. Se recomienda el uso del método `doFinal()` siempre que sea posible. Los datos de entrada y de salida del buffer quizás se solapen. El parámetro `inBuff` representa al buffer donde se van a codificar o decodificar los datos. `inOffset` da la posición de `inBuff` a partir de la cual empiezan los `inLength` bytes datos que se van a codificar o decodificar. `outBuff` es el buffer de salida, que quizás sea el mismo que el buffer de entrada. `outOffset` da la posición de `outBuff` a partir de la cual empiezan los datos codificados o decodificados. El método devuelve el número de bytes de datos que se han introducido en `outBuff`.

En las operaciones de decodificación (excepto cuando se usa el relleno del método 1 de la ISO 9797) los bytes de relleno no se escriben en `outBuff`. En las operaciones de codificación, el número de bytes de salida (que se escriben en `outBuff`) quizás sea mayor que `inLength`. En las operaciones de codificación y decodificación (excepto cuando se usa el relleno del método 1 de la ISO 9797), las consideraciones de la alineación de bloque quizás requieran que el número de bytes de salida de `outBuff` sean menores que `inLength` o igual a 0.

14 APÉNDICE 5: CONTENIDO DEL PAQUETE SIM.ACCESS

Este paquete provee la manera en la que los applets pueden acceder a los datos GSM y el sistema de ficheros de la aplicación GSM definida en la especificación GSM 11.11.

14.1 INTERFACES

14.1.1 INTERFAZ SIMVIEW

Esta interfaz extiende `javacard.framework.Shareable`. La interfaz `SIMView` es la interfaz entre el sistema de servicios GSM y cualquier applet (SIM Toolkit u otro). Ofrece métodos para comunicarse con el sistema de servicios de GSM sin comprometer la integridad del sistema de ficheros GSM. Todos los métodos están basados en los comandos de la especificación GSM 11.11. Esta interfaz debe ser implementada por un objeto perteneciente al JCRE. Esta interfaz no está pensada para la activación de un applet toolkit.

14.1.1.1 Campos

`static short FID_DF_ACES` : Identificador del fichero DF ACeS = 0x5F33 (cuelga del DF GSM).

`static short FID_DF_CTS` : Identificador del fichero DF CTS = 0x5F60 (cuelga del DF GSM).

`static short FID_DF_DCS_1800` : Identificador del fichero DF DCS-1800 = 0x7F21.

`static short FID_DF_FP_CTS` : Identificador del fichero DF FP-CTS = 0x7F23.

`static short FID_DF_GLOBALSTAR` : Identificador del fichero DF Globalstar = 0x5F31 (cuelga del DF GSM).

`static short FID_DF_Graphics` : Se sustituye por `FID_DF_GRAPHICS` a partir de la versión 7.4.0.

`static short FID_DF_GRAPHICS` : Identificador del fichero DF Graphics = 0x5F50 (cuelga del DF TELECOM).

`static short FID_DF_GSM` : Identificador del fichero DF GSM = 0x7F20.

`static short FID_DF_ICO` : Identificador del fichero DF ICO = 0x5F32 (cuelga del DF GSM).

`static short FID_DF_IRIDIUM` : Identificador del fichero DF IRIDIUM = 0x5F30 (cuelga del DF GSM).

`static short FID_DF_IS_41` : Identificador del fichero DF IS-41 = 0x7F22.

`static short FID_DF_MEXE` : Identificador del fichero DF MExE = 0x5F3C (cuelga del DF GSM).

`static short FID_DF_PCS_1900` : Identificador del fichero DF PCS-1900 = 0x5F40 (cuelga del DF GSM).

`static short FID_DF_PDC` : Identificador del fichero DF PDC = 0x7F80

`static short FID_DF_SOLSA` : Identificador del fichero DF SoLSA = 0x5F70 (cuelga del DF GSM).

`static short FID_DF_TELECOM` : Identificador del fichero DF TELECOM = 0x7F10.

`static short FID_DF_TETRA` : Identificador del fichero DF TETRA = 0x7F90.

`static short FID_DF_TIA_EIA_136` : Identificador del fichero DF TIA-EIA-136 = 0x7F24.

`static short FID_DF_TIA_EIA_553` : Identificador del fichero DF TIA-EIA-553= 0x5F40 (cuelga del DF GSM).

`static short FID_DF_TIA_EIA_95` : Identificador del fichero DF TIA-EIA-95 = 0x7F25.

`static short FID_EF_AAEM` : Identificador del fichero EF AAeM = 0x6FB6 (cuelga del DF GSM).

`static short FID_EF_ACC` : Identificador del fichero EF ACC = 0x6F78 (cuelga del DF GSM).

`static short FID_EF_ACCOLC` : Identificador del fichero EF ACCOLC = 0x4F89 (cuelga del DF TIA-EIA-553).

`static short FID_EF_ACM` : Identificador del fichero EF ACM = 0x6F39 (cuelga del DF GSM).

`static short FID_EF_ACMMAX` : Identificador del fichero EF ACMmax = 0x6F37 (cuelga del DF GSM).

`static short FID_EF_AD` : Identificador del fichero EF AD = 0x6FAD (cuelga del DF GSM).

`static short FID_EF_ADN` : Identificador del fichero EF ADN = 0x6F3A (cuelga del DF TELECOM).

`static short FID_EF_AMPS_2_GSM` : Identificador del fichero EF AMPS_2_GSM = 0x4F91 (cuelga del DF TIA-EIA-553).

`static short FID_EF_AMPS_UI` : Identificador del fichero EF AMPS_UI = 0x4F93 (cuelga del DF TIA-EIA-553).

`static short FID_EF_ARPK` : Identificador del fichero EF ARPK= 0x4F42 (cuelga del DF MExE).

`static short FID_EF_BCCH` : Identificador del fichero EF BCCH = 0x6F74 (cuelga del DF GSM).

`static short FID_EF_BDN` : Identificador del fichero EF BDN = 0x6F4D (cuelga del DF TELECOM).

`static short FID_EF_CBMI` : Identificador del fichero EF CBMI = 0x6F45 (cuelga del DF GSM).

`static short FID_EF_CBMID` : Identificador del fichero EF CBMID = 0x6F48 (cuelga del DF GSM).

`static short FID_EF_CBMIR` : Identificador del fichero EF CBMIR = 0x6F50 (cuelga del DF GSM).

`static short FID_EF_CCCH` : Identificador del fichero EF CCCH = 0x4F8E (cuelga del DF TIA-EIA-553).

`static short FID_EF_CCP` : Identificador del fichero EF CCP = 0x6F3D (cuelga del DF TELECOM).

`static short FID_EF_CMI` : Identificador del fichero EF CMI = 0x6F58 (cuelga del DF TELECOM).

`static short FID_EF_CNL` : Identificador del fichero EF CNL = 0x6F32 (cuelga del DF GSM).

`static short FID_EF_COUNT` : Identificador del fichero EF COUNT = 0x4F83 (cuelga del DF TIA-EIA-553).

`static short FID_EF_CPBCCH` : Identificador del fichero EF CPBCCH= 0x6F63(cuelga del DF GSM).

`static short FID_EF_CSID` : Identificador del fichero EF CSID= 0x4F8C (cuelga del DF TIA-EIA-553).

`static short FID_EF_DCK` : Identificador del fichero EF DCK = 0x6F2C (cuelga del DF GSM).

`static short FID_EF_ECC` : Identificador del fichero EF ECC = 0x6FB7 (cuelga del DF GSM).

`static short FID_EF_ECCP` : Identificador del fichero EF ECCP = 0x6F4F (cuelga del DF TELECOM).

`static short FID_EF_ELP` : Identificador del fichero EF ELP = 0x2F05 (cuelga del MF).

`static short FID_EF_EMLPP` : Identificador del fichero EF eMLPP = 0x6FB5 (cuelga del DF GSM).

`static short FID_EF_EXT1` : Identificador del fichero EF EXT1 = 0x6F4A (cuelga del DF TELECOM).

`static short FID_EF_EXT2` : Identificador del fichero EF EXT2 = 0x6F4B (cuelga del DF TELECOM).

`static short FID_EF_EXT3` : Identificador del fichero EF EXT3 = 0x6F4C (cuelga del DF TELECOM).

`static short FID_EF_EXT4` : Identificador del fichero EF EXT4 = 0x6F4E (cuelga del DF TELECOM).

`static short FID_EF_FC1` : Identificador del fichero EF FC1= 0x4F8A (cuelga del DF TIA-EIA-553).

`static short FID_EF_FDN` : Identificador del fichero EF FDN = 0x6F3B (cuelga del DF TELECOM).

`static short FID_EF_FPLMN` : Identificador del fichero EF FPLMN = 0x6F7B (cuelga del DF GSM).

`static short FID_EF_GID1` : Identificador del fichero EF GID1 = 0x6F3E (cuelga del DF GSM).

`static short FID_EF_GID2` : Identificador del fichero EF GID2 = 0x6F3F (cuelga del DF GSM).

`static short FID_EF_GPI` : Identificador del fichero EF GPI = 0x4F81 (cuelga del DF TIA-EIA-553).

`static short FID_EF_GSM_RECON` : Identificador del fichero EF GSM_RECON = 0x4F90 (cuelga del DF TIA-EIA-553).

`static short FID_EF_HPLMN` : Identificador del fichero EF HPLMN = 0x6F31 (cuelga del DF GSM).

`static short FID_EF_HPLMNWACT` : Identificador del fichero EF HPLMNwAcT= 0x6F62 (cuelga del DF GSM).

`static short FID_EF_ICCID` : Identificador del fichero EF ICCID = 0x2FE2 (cuelga del MF).

`static short FID_EF_IMG` : Identificador del fichero EF IMG = 0x4F20 (cuelga del DF Graphics).

`static short FID_EF_IMSI` : Identificador del fichero EF IMSI = 0x6F07 (cuelga del DF GSM).

`static short FID_EF_INVSCAN` : Identificador del fichero EF InvScan = 0x6F64 (cuelga del DF GSM).

`static short FID_EF_IPC` : Identificador del fichero EF IPC = 0x4F82 (cuelga del DF TIA-EIA-553).

`static short FID_EF_KC` : Identificador del fichero EF Kc = 0x6F20 (cuelga del DF GSM).

`static short FID_EF_KCGPRS` : Identificador del fichero EF KcGPRS = 0x6F52 (cuelga del DF GSM).

`static short FID_EF_LDCC` : Identificador del fichero EF LDCC = 0x4F8F (cuelga del DF TIA-EIA-553).

`static short FID_EF_LND` : Identificador del fichero EF LND = 0x6F44 (cuelga del DF TELECOM).

`static short FID_EF_LOCI` : Identificador del fichero EF LOCI = 0x6F7E (cuelga del DF GSM).

`static short FID_EF_LOCIGPRS` : Identificador del fichero EF LOCIGPRS = 0x6F53 (cuelga del DF GSM).

`static short FID_EF_LP` : Identificador del fichero EF LP = 0x6F05 (cuelga del DF GSM).

`static short FID_EF_MEXE_ST` : Identificador del fichero EF MExE_ST= 0x4F40 (cuelga del DF MExE).

`static short FID_EF_MIN` : Identificador del fichero EF MIN = 0x4F88 (cuelga del DF TIA-EIA-553).

`static short FID_EF_MSISDN` : Identificador del fichero EF MSISDN = 0x6F40 (cuelga del DF TELECOM).

`static short FID_EF_NETSEL` : Identificador del fichero EF NETSEL = 0x4F86 (cuelga del DF TIA-EIA-553).

`static short FID_EF_NIA` : Identificador del fichero EF NIA = 0x6F51 (cuelga del DF GSM).

`static short FID_EF_NSID` : Identificador del fichero EF NSID = 0x4F84 (cuelga del DF TIA-EIA-553).

`static short FID_EF_OPLMNWACT` : Identificador del fichero EF OPLMNwAcT= 0x6F61 (cuelga del DF GSM).

`static short FID_EF_ORPK` : Identificador del fichero EF ORPK= 0x4F41 (cuelga del DF MExE).

`static short FID_EF_PHASE` : Identificador del fichero EF Phase = 0x6FAE (cuelga del DF GSM).

`static short FID_EF_PLMNSEL` : Identificador del fichero EF PLMNsel = 0x6F30 (cuelga del DF GSM).

`static short FID_EF_PLMNWACT` : Identificador del fichero EF PLMNwAcT= 0x6F60 (cuelga del DF GSM).

`static short FID_EF_PSID` : Identificador del fichero EF PSID = 0x4F85 (cuelga del DF TIA-EIA-553).

`static short FID_EF_PUCT` : Identificador del fichero EF PUCT = 0x6F41 (cuelga del DF GSM).

`static short FID_EF_REG_THRESH` : Identificador del fichero EF REG_THRESH = 0x4F8D (cuelga del DF TIA-EIA-553).

`static short FID_EF_S_ESN` : Identificador del fichero EF S_ESN = 0x4F8B (cuelga del DF TIA-EIA-553).

`static short FID_EF_SAI` : Identificador del fichero EF SAI = 0x4F30 (cuelga del DF SoLSA).

`static short FID_EF_SDN` : Identificador del fichero EF SDN = 0x6F49 (cuelga del DF TELECOM).

`static short FID_EF_SID` : Identificador del fichero EF SID = 0x4F80 (cuelga del DF TIA-EIA-553).

`static short FID_EF_SLL` : Identificador del fichero EF SLL = 0x4F31 (cuelga del DF SoLSA).

`static short FID_EF_SMS` : Identificador del fichero EF SMS = 0x6F3C (cuelga del DF TELECOM).

`static short FID_EF_SMSP` : Identificador del fichero EF SMSP = 0x6F42 (cuelga del DF TELECOM).

`static short FID_EF_SMSR` : Identificador del fichero EF SMSR = 0x6F47 (cuelga del DF TELECOM).

`static short FID_EF_SMSS` : Identificador del fichero EF SMSS = 0x6F43 (cuelga del DF TELECOM).

`static short FID_EF_SPL` : Identificador del fichero EF SPL = 0x4F87 (cuelga del DF TIA-EIA-553).

`static short FID_EF_SPN` : Identificador del fichero EF SPN = 0x6F46 (cuelga del DF GSM).

`static short FID_EF_SST` : Identificador del fichero EF SST = 0x6F38 (cuelga del DF GSM).

`static short FID_EF_SUME` : Identificador del fichero EF SUME = 0x6F54 (cuelga del DF GSM).

`static short FID_EF_TPRPK` : Identificador del fichero EF TPRPK = 0x4F43 (cuelga del DF MExE).

`static short FID_EF_VBS` : Identificador del fichero EF VBS = 0x6FB3 (cuelga del DF GSM).

`static short FID_EF_VBSS` : Identificador del fichero EF VBSS = 0x6FB4 (cuelga del DF GSM).

`static short FID_EF_VGCS` : Identificador del fichero EF VGCS = 0x6FB1 (cuelga del DF GSM).

`static short FID_EF_VGCSS` : Identificador del fichero EF VGCSS = 0x6FB2 (cuelga del DF GSM).

`static short FID_MF` : Identificador del fichero MF = 0x3F00.

`static byte REC_ACC_MODE_ABSOLUTE_CURRENT` : Registro de modo de acceso del registro absoluto/actual = 0x04.

`static byte REC_ACC_MODE_NEXT` : Registro de modo de acceso del próximo registro = 0x02.

`static byte REC_ACC_MODE_PREVIOUS` : Registro de modo de acceso del registro anterior = 0x03.

`static byte SEEK_FROM_BEGINNING_FORWARD` : Modo de búsqueda desde el principio hacia delante = 0x00.

`static byte SEEK_FROM_END_BACKWARD` : Modo de búsqueda desde el final hacia atrás = 0x01.

`static byte SEEK_FROM_NEXT_FORWARD` : Modo de búsqueda desde el próximo hacia delante = 0x02.

`static byte SEEK_FROM_PREVIOUS_BACKWARD` : Modo de búsqueda desde el anterior hacia atrás = 0x03.

14.1.1.2 Métodos

`public short select(short fid, byte[] fci, short fciOffset, short fciLength)` : Este método representa al comando SELECT definido en la norma GSM 11.11. Por defecto, el MF se selecciona al comienzo de la activación de cada applet. Este método selecciona un fichero del sistema de ficheros GSM. La búsqueda del fichero empieza en el DF actual según el método de búsqueda de ficheros descrito en la especificación GSM 11.11. El DF actual o el EF actual y el puntero del registro actual pueden ser cambiados después de una ejecución exitosa del applet. El parámetro `fid` es el identificador del fichero que va a ser seleccionado. `fci` es un array de `fciLength` (el array debe ser de tamaño suficiente) bytes donde el método almacena el FCI (File Control Information), codificado de acuerdo con la norma GSM 11.11. `fciOffset` indica la posición a partir de la cual se va a almacenar el FCI en el array (`fciOffset+fciLength` debe ser menor que `fci.length`). El método devuelve la longitud de los datos que se han escrito en el buffer `fci`. El método podrá lanzar una `SIMViewException` con los siguientes códigos de causa: `FILE_NOT_FOUND`, `MEMORY_PROBLEM` e `INTERNAL_ERROR`.

`public void select(short fid)` : Representa al comando SELECT definido en la norma GSM 11.11. Este método permite actualizar el fichero actual sin tener que manejar la respuesta de anterior método `select`. Por defecto, el MF está seleccionado al comienzo de la activación de cada applet. Este método selecciona un fichero del sistema de ficheros GSM. La búsqueda del ficheros empieza en el DF actual tal y como se describe en la norma GSM 11.11. El DF actual o el EF actual y el puntero del registro actual pueden ser cambiados después de una ejecución exitosa del applet. El parámetro `fid` es el identificador del fichero que va a ser seleccionado. El método puede lanzar una `SIMViewException` con los siguientes códigos de causa: `FILE_NOT_FOUND`, `MEMORY_PROBLEM` e `INTERNAL_ERROR`.

`public short status(byte[] fci, short fciOffset, short fciLength)` : El método `status` representa al comando STATUS definido en el estándar GSM 11.11. Este método devuelve el FCI (File Control Information) del DF actual (también del MF) del applet que lo llama. `fci` es un array de `fciLength` (el array debe ser de tamaño suficiente) bytes donde el método almacena el FCI (File Control Information) del DF (o MF) actual, codificado de acuerdo con la norma GSM 11.11. `fciOffset` indica la posición a partir de la cual se va a almacenar el FCI en el array (`fciOffset+fciLength` debe ser menor que `fci.length`). El método devuelve la longitud de los datos que se han escrito en el buffer `fci`. El método podrá lanzar una `SIMViewException` con los siguientes códigos de causa: `MEMORY_PROBLEM` e `INTERNAL_ERROR`.

`public short readBinary(short fileOffset, byte[] resp, short respOffset, short respLength)` : Representa al comando READ BINARY definido en el estándar GSM 11.11. Este método lee los bytes de datos del EF transparente actual del applet llamante. El parámetro `fileOffset` indica el offset a partir del cual se empezará a leer en el fichero transparente actual. `resp` es el buffer de bytes donde se guardan (a partir de la posición `respOffset`) los `respLength` bytes de datos a leer. El método devuelve `respOffset+respLength`, cantidad que debe ser menor que `resp.length`. El método podrá lanzar una `SIMViewException` con los siguientes códigos de causa: `NO_EF_SELECTED`, `FILE_INCONSISTENT`,

AC_NOT_FULFILLED, INVALIDATION_STATUS_CONTRADICTION, OUT_OF_FILE_BOUNDARIES, MEMORY_PROBLEM e INTERNAL_ERROR.

`public void updateBinary(short fileOffset, byte[] data, short dataOffset, short dataLength)` : Este método representa al comando UPDATE BINARY definido en la norma GSM 11.11. Actualiza los bytes de datos del EF transparente actual del applet llamante. El parámetro `fileOffset` indica el offset a partir del cual se empezará a escribir en el fichero transparente actual. `data` es el buffer de bytes donde se encuentran (a partir de la posición `dataOffset`) los `dataLength` bytes de datos a escribir en el fichero de destino. El método podrá lanzar una `SIMViewException` con el código de causa: `NO_EF_SELECTED`, `FILE_INCONSISTENT`, `AC_NOT_FULFILLED`, `INVALIDATION_STATUS_CONTRADICTION`, `OUT_OF_FILE_BOUNDARIES`, `MEMORY_PROBLEM` e `INTERNAL_ERROR`.

`public short readRecord(short recNumber, byte mode, short recOffset, byte[] resp, short respOffset, short respLength)` : Este método implementa el comando READ RECORD definido en el estándar GSM 11.11. Lee los bytes de datos del EF lineal fijo / cíclico actual del applet llamante. El puntero del registro actual puede ser cambiado según el modo elegido. El parámetro `recNumber` es el número de registro. `mode` indica el modo de lectura de registros según la norma GSM 11.11. Si el modo es `REC_ACC_MODE_NEXT` y el puntero de registro apunta al último registro, se lanza una `SIMViewException` con el código de causa `RECORD_NUMBER_NOT_AVAILABLE`. Si el modo es `REC_ACC_MODE_PREVIOUS` y el puntero de registro apunta al primer registro se lanza una `SIMViewException` con el código de causa `RECORD_NUMBER_NOT_AVAILABLE`. `resp` es el buffer de bytes donde se guardan (a partir de la posición `respOffset`) los `respLength` bytes de datos a leer a partir de la posición `recOffset` del registro. El método devuelve `respOffset+respLength`, que no debe ser mayor que `resp.length`. Los otros códigos de causa que puede lanzar la `SIMViewException` son: `NO_EF_SELECTED`, `FILE_INCONSISTENT`, `AC_NOT_FULFILLED`, `INVALIDATION_STATUS_CONTRADICTION`, `OUT_OF_RECORD_BOUNDARIES`, `INVALID_MODE`, `MEMORY_PROBLEM` e `INTERNAL_ERROR`.

`public void updateRecord(short recNumber, byte mode, short recOffset, byte[] data, short dataOffset, short dataLength)` : Este método implementa al comando UPDATE RECORD definido en el estándar GSM 11.11. Actualiza los bytes de datos del EF lineal fijo / cíclico actual del applet llamante. El puntero de registro actual se puede cambiar según el modo elegido. El parámetro `recNumber` es el número de registro. `mode` indica el modo de lectura de registros según la norma GSM 11.11. Si el modo es `REC_ACC_MODE_NEXT` y el puntero de registro apunta al último registro, se lanza una `SIMViewException` con el código de causa `RECORD_NUMBER_NOT_AVAILABLE`. Si el modo es `REC_ACC_MODE_PREVIOUS` y el puntero de registro apunta al primer registro se lanza una `SIMViewException` con el código de causa `RECORD_NUMBER_NOT_AVAILABLE`. `data` es el buffer de bytes donde se encuentran (a partir de la posición `dataOffset`) los `dataLength` bytes de datos a escribir en el fichero de destino (`dataOffset+dataLength` debe ser menor que `data.length`). Los otros códigos de causa que puede lanzar la `SIMViewException` son: `NO_EF_SELECTED`, `FILE_INCONSISTENT`,

AC_NOT_FULFILLED, INVALIDATION_STATUS_CONTRADICTION, OUT_OF_RECORD_BOUNDARIES, INVALID_MODE, MEMORY_PROBLEM e INTERNAL_ERROR.

`public short seek(byte mode, byte[] patt, short pattOffset, short pattLength)` : El método `seek` representa al comando SEEK definido en el estándar GSM 11.11. Este método busca un patrón en el EF lineal fijo / cíclico actual del applet llamante. El parámetro `mode` indica el modo de búsqueda según la norma GSM 11.11. Si el modo es `SEEK_FROM_NEXT_FORWARD` y el puntero de registro apunta al último registro, se lanzará una `SIMViewException` con el código de causa `PATTERN_NOT_FOUND`. Si el modo es `SEEK_FROM_PREVIOUS_BACKWARD` y el puntero de registro apunta al primer registro, se lanzará una `SIMViewException` con el código de causa `PATTERN_NOT_FOUND`. `patt` es el buffer de bytes que contiene el patrón (que se encuentra en `pattOffset`) a buscar. `pattLength` da la longitud del patrón a buscar (`pattOffset+pattLength` no debe ser mayor que `patt.length`). `pattLength` no debe ser mayor que el tamaño del registro (sino se lanzaría una `SIMViewException` con código de causa `OUT_OF_RECORD_BOUNDARIES`). `pattLength` tampoco debe ser cero (sino provocaría el lanzamiento de una `SIMViewException` con código de causa `PATTERN_NOT_FOUND`). El método devuelve el número del registro en el que se encontró el patrón. Los otros códigos de causa que puede lanzar la `SIMViewException` son: `NO_EF_SELECTED`, `FILE_INCONSISTENT`, `INVALIDATION_STATUS_CONTRADICTION`, `INVALID_MODE`, `MEMORY_PROBLEM` e `INTERNAL_ERROR`.

`public short increase(byte[] incr, short incrOffset, byte[] resp, short respOffset)` : Este método representa al comando INCREASE definido en el estándar GSM 11.11. Incrementa el valor del último registro (de un EF cíclico) que ha sido actualizado o incrementado y guarda el resultado en el registro más antiguo. El parámetro `incr` es el array de bytes que contiene el valor (que debe tener un tamaño de 3 bytes) a añadir y que se encuentra a partir de la posición dada por `incrOffset`. El buffer de respuesta `resp` contiene el valor del registro incrementado a partir de la posición `respOffset` de este buffer. El buffer `resp` debe ser mayor que el tamaño del registro y se llena con el valor del registro alineado a la izquierda. El método devuelve la longitud (que no podrá ser mayor que el tamaño del registro) de los datos válidos en el buffer `resp`. El método podrá lanzar una `SIMViewException` con el código de causa: `NO_EF_SELECTED`, `FILE_INCONSISTENT`, `AC_NOT_FULFILLED`, `INVALIDATION_STATUS_CONTRADICTION`, `MAX_VALUE_REACHED`, `MEMORY_PROBLEM` e `INTERNAL_ERROR`.

`public void invalidate()` : Este método representa al comando INVALIDATE definido en el estándar GSM 11.11. Invalida el EF actualmente seleccionado del applet llamante. El método podrá lanzar una `SIMViewException` con el código de causa: `NO_EF_SELECTED`, `AC_NOT_FULFILLED`, `INVALIDATION_STATUS_CONTRADICTION`, `MEMORY_PROBLEM` e `INTERNAL_ERROR`.

`public void rehabilitate()` : Representa al comando REHABILITATE definido en la norma GSM 11.11. Este método rehabilita el EF actualmente seleccionado del applet llamante. El método podrá lanzar una `SIMViewException` con

el código de causa: `NO_EF_SELECTED`, `AC_NOT_FULFILLED`, `INVALIDATION_STATUS_CONTRADICTION`, `MEMORY_PROBLEM` e `INTERNAL_ERROR`.

14.2 CLASES

14.2.1 CLASE `SIMSYSTEM`

`public class SIMSystem extends java.lang.Object` : La clase `SIMSystem` provee una manera de conseguir una vista del sistema de ficheros de GSM. En cualquier caso, el applet cliente (ya sea SIM Toolkit u otro) solo podrá acceder a los métodos de la interfaz `SIMView`. No hace falta ninguna instancia de esta clase.

14.2.1.1 Métodos

`public static SIMView getTheSIMView()` : Devuelve una referencia de la interfaz GSM.

14.3 EXCEPCIONES

14.3.1 EXCEPCIÓN `SIMVIEWEXCEPTION`

`public class SIMViewException extends javacard.framework. CardRuntimeException` : La clase `SIMViewException` encapsula excepciones específicas que los métodos de la interfaz `SIMView` pueden lanzar en caso de error.

14.3.1.1 Campos

`static short AC_NOT_FULFILLED` : Este código de causa (= 3) se usa para indicar que el applet que ha llamado al método no cumple la condición de acceso.

`static short FILE_INCONSISTENT` : Este código de causa (= 2) se usa para indicar que el tipo de fichero actual no concuerda con el método llamado.

`static short FILE_NOT_FOUND` : Este código de causa (= 4) se usa para indicar al applet llamante que el fichero no se encuentra en el directorio actual.

`static short INTERNAL_ERROR` : Este código de causa (= 5) se usa para indicar al applet llamante que ha ocurrido un error interno durante la llamada al método.

`static short INVALID_MODE` : Este código de causa (= 10) se usa para indicar que el método llamado no soporta el modo de búsqueda o de acceso solicitado.

`static short INVALIDATION_STATUS_CONTRADICTION` : Este código de causa (= 6) se usa para indicar que el método llamado está en contradicción con el estado de invalidación del fichero actual.

`static short MAX_VALUE_REACHED` : Este código de causa (= 12) se usa para indicar que el método no se puede llevar a cabo cuando se alcanza el valor máximo de un registro.

`static short MEMORY_PROBLEM` : Este código de causa (= 13) se usa para indicar que ha ocurrido un problema de memoria.

`static short NO_EF_SELECTED` : Este código de causa (= 1) se usa para indicar que no hay un EF seleccionado para concluir el método llamado.

`static short OUT_OF_FILE_BOUNDARIES` : Este código de causa (= 7) se usa para indicar que la longitud o el offset del fichero o ambos, pasados como parámetros al método llamado, están fuera de los límites del fichero transparente actual.

`static short OUT_OF_RECORD_BOUNDARIES` : Este código de causa (= 8) se usa para indicar que la longitud o el offset del fichero o ambos, pasados como parámetros al método llamado, están fuera de los límites del registro actual.

`static short PATTERN_NOT_FOUND` : Este código de causa (= 11) se usa para indicar que el patrón solicitado no se ha pasado al método llamado.

`static short RECORD_NUMBER_NOT_AVAILABLE` : Este código de causa (= 9) se usa para indicar que el número de registro indicado no está disponible en el fichero actual.

14.3.1.2 Constructores

`public SIMViewException(short reason)` : Construye una `SIMViewException` con el código de causa especificado en el parámetro `reason`. Para ahorrar recursos de la tarjeta, use el método `throwIt()` para utilizar la instancia del JCRE de esta clase.

14.3.1.3 Métodos

`public static void throwIt(short reason)` : Lanza la instancia del JCRE de `SIMViewException` con el código de causa especificada en el parámetro `reason`.

15 APÉNDICE 6: CONTENIDO DEL PAQUETE SIM.TOOLKIT

15.1 INTERFACES

15.1.1 INTERFAZ TOOLKITCONSTANTS

La interfaz `ToolkitConstants` encapsula constantes relacionadas con los applets toolkit.

15.1.1.1 Campos

`static byte BTAG_CALL_CONTROL` : En objetos BER-TLV representa la etiqueta de control de llamadas = 0xD4.

`static byte BTAG_CELL_BROADCAST_DOWNLOAD` : En objetos BER-TLV : representa la etiqueta de descarga a través de Cell Broadcast = 0xD2.

`static byte BTAG_EVENT_DOWNLOAD` : En objetos BER-TLV representa la etiqueta de evento de descarga = 0xD6.

`static byte BTAG_MENU_SELECTION` : En objetos BER-TLV representa la etiqueta de selección de menú = 0xD3.

`static byte BTAG_MO_SHORT_MESSAGE_CONTROL` : En objetos BER-TLV representa la etiqueta de control de mensajes cortos originados en el ME = 0xD5.

`static byte BTAG_PROACTIVE_SIM_COMMAND` : En objetos BER-TLV representa la etiqueta de comando SIM proactivo = 0xD0.

`static byte BTAG_SMS_PP_DOWNLOAD` : En objetos BER-TLV representa la etiqueta de descarga a través de SMS-PP = 0xD1.

`static byte BTAG_SMS_PP_DOWNWLOAD` : En desuso. Se sustituye por `BTAG_SMS_PP_DOWNLOAD` a partir de la versión 7.4.0.

`static byte BTAG_TIMER_EXPIRATION` : En objetos BER-TLV representa la etiqueta de expiración del contador = 0xD7.

`static byte DCS_8_BIT_DATA` : Representa al esquema de codificación de datos GSM 8 bit Data = 0x04.

`static byte DCS_DEFAULT_ALPHABET` : Representa al esquema de codificación de datos GSM Default Aphabet = 0x00.

`static byte DCS_UCS2` : Representa al esquema de codificación de datos UCS2 = 0x08.

`static byte DEV_ID_ADDITIONAL_CARD_READER_0` : Identificador de dispositivo. Lector adicional 0 de tarjetas = 0x10.

`static byte DEV_ID_ADDITIONAL_CARD_READER_1` : Identificador de dispositivo. Lector adicional 1 de tarjetas = 0x11.

`static byte DEV_ID_ADDITIONAL_CARD_READER_2` : Identificador de dispositivo. Lector adicional 2 de tarjetas = 0x12.

`static byte DEV_ID_ADDITIONAL_CARD_READER_3` : Identificador de dispositivo. Lector adicional 3 de tarjetas = 0x13.

`static byte DEV_ID_ADDITIONAL_CARD_READER_4` : Identificador de dispositivo. Lector adicional 4 de tarjetas = 0x14.

`static byte DEV_ID_ADDITIONAL_CARD_READER_5` : Identificador de dispositivo. Lector adicional 5 de tarjetas = 0x15.

`static byte DEV_ID_ADDITIONAL_CARD_READER_6` : Identificador de dispositivo. Lector adicional 6 de tarjetas = 0x16

`static byte DEV_ID_ADDITIONAL_CARD_READER_7` : Identificador de dispositivo. Lector adicional 7 de tarjetas = 0x17.

`static byte DEV_ID_DISPLAY` : Identificador de dispositivo. Pantalla = 0x02

`static byte DEV_ID_EARPIECE` : Identificador de dispositivo. Auricular = 0x03.

`static byte DEV_ID_KEYPAD` : Identificador de dispositivo. Teclado numérico = 0x01.

`static byte DEV_ID_ME` : Identificador de dispositivo. ME = 0x82

`static byte DEV_ID_NETWORK` : Identificador de dispositivo. Red = 0x83

`static byte DEV_ID_SIM` : Identificador de dispositivo. SIM = 0x81

`static byte EVENT_CALL_CONTROL_BY_SIM` : Representa al evento de control de llamadas a través de la SIM = 9.

`static byte EVENT_EVENT_DOWNLOAD_BROWSER_TERMINATION` : Representa al evento de terminación del navegador = 21.

`static byte EVENT_EVENT_DOWNLOAD_CALL_CONNECTED` : Representa al evento de tipo de llamada conectada = 13.

`static byte EVENT_EVENT_DOWNLOAD_CALL_DISCONNECTED` : Representa al evento de tipo de llamada desconectada = 14.

`static byte EVENT_EVENT_DOWNLOAD_CARD_READER_STATUS` : Representa al evento de estado del lector de tarjetas = 18.

`static byte EVENT_EVENT_DOWNLOAD_IDLE_SCREEN_AVAILABLE` : Representa al evento de tipo de pantalla disponible en espera = 17.

`static byte EVENT_EVENT_DOWNLOAD_LANGUAGE_SELECTION` : Representa al evento de selección de lengua = 20.

`static byte EVENT_EVENT_DOWNLOAD_LOCATION_STATUS` : Representa al evento de tipo de estado de la localización = 15.

`static byte EVENT_EVENT_DOWNLOAD_MT_CALL` : Representa al evento de tipo de llamada MT = 12.

`static byte EVENT_EVENT_DOWNLOAD_USER_ACTIVITY` : Representa al evento de tipo de actividad del usuario = 16.

`static byte EVENT_FORMATTED_SMS_CB` : Evento de descarga de datos con el formato Cell Broadcast = 24.

`static byte EVENT_FORMATTED_SMS_PP_ENV` : Evento de descarga de datos con el formato (03.48) Envelope SMS-PP = 2.

`static byte EVENT_FORMATTED_SMS_PP_UPD` : Evento de APDU (con formato 03.48) de actualización de un registro del EF SMS = 3

`static byte EVENT_MENU_SELECTION` : Evento de selección de menú = 7.

`static byte EVENT_MENU_SELECTION_HELP_REQUEST` : Evento de selección del menú de solicitud de ayuda = 8.

`static byte EVENT_MO_SHORT_MESSAGE_CONTROL_BY_SIM` : Evento de control de mensaje corto (originado en el ME) por la SIM = 10.

`static byte EVENT_PROFILE_DOWNLOAD` : Evento de descarga del perfil del ME = 1.

`static byte EVENT_STATUS_COMMAND` : Evento de estado de un comando APDU = 19.

`static byte EVENT_TIMER_EXPIRATION` : Evento de expiración de contador = 11.

`static byte EVENT_UNFORMATTED_SMS_CB` : Evento de descarga de datos Cell Broadcast = 6.

`static byte EVENT_UNFORMATTED_SMS_PP_ENV` : Evento Envelope SMS-PP de descarga de datos a través de SMS sin formato = 4

`static byte EVENT_UNFORMATTED_SMS_PP_UPD` : Evento de recepción de una APDU de actualización de un registro de EF SMS en un SMS sin formato = 5

`static byte EVENT_UNRECOGNIZED_ENVELOPE` : Evento Unrecognized Envelope = -1.

`static byte POLL_NO_DURATION` : Solicitud de desregistro del evento asociado al intervalo de monitorización (Poll Interval) = 0.

`static byte POLL_SYSTEM_DURATION` : Solicitud de la duración del intervalo de monitorización establecido en el sistema = -1.

`static byte PRO_CMD_DISPLAY_TEXT` : Comando proactivo DISPLAY TEXT = 0x21.

`static byte PRO_CMD_GET_INKEY` : Comando proactivo GET INKEY = 0x22.

`static byte PRO_CMD_GET_INPUT` : Comando proactivo GET INPUT = 0x23.

`static byte PRO_CMD_GET_READER_STATUS` : Comando proactivo GET READER STATUS = 0x33.

`static byte PRO_CMD_LANGUAGE_NOTIFICATION` : Comando proactivo LANGUAGE NOTIFICATION = 0x35.

`static byte PRO_CMD_LAUNCH_BROWSER` : Comando proactivo LAUNCH BROWSER = 0x15.

`static byte PRO_CMD_MORE_TIME` : Comando proactivo MORE TIME = 0x02.

`static byte PRO_CMD_PERFORM_CARD_APDU` : Comando proactivo PERFORM CARD APDU = 0x30.

`static byte PRO_CMD_PLAY_TONE` : Comando proactivo PLAY TONE = 0x20.

`static byte PRO_CMD_POWER_OFF_CARD` : Comando proactivo POWER OFF CARD = 0x32.

`static byte PRO_CMD_POWER_ON_CARD` : Comando proactivo POWER ON CARD = 0x31.

`static byte PRO_CMD_PROVIDE_LOCAL_INFORMATION` : Comando proactivo PROVIDE LOCAL INFORMATION = 0x26.

`static byte PRO_CMD_REFRESH` : Comando proactivo REFRESH = 0x01.

`static byte PRO_CMD_RUN_AT_COMMAND` : Comando proactivo RUN AT COMMAND = 0x34.

`static byte PRO_CMD_SELECT_ITEM` : Comando proactivo SELECT ITEM = 0x24.

`static byte PRO_CMD_SEND_DTMF` : Comando proactivo SEND DTMF = 0x14.

`static byte PRO_CMD_SEND_SHORT_MESSAGE` : Comando proactivo SEND SHORT MESSAGE = 0x13.

`static byte PRO_CMD_SEND_SS` : Comando proactivo SEND SS = 0x11.

`static byte PRO_CMD_SEND_USSD` : Comando proactivo SEND USSD = 0x12.

`static byte PRO_CMD_SET_UP_CALL` : Comando proactivo SET UP CALL = 0x10.

`static byte PRO_CMD_SET_UP_IDLE_MODE_TEXT` : Comando proactivo SET UP IDLE MODE TEXT = 0x28.

`static byte PRO_CMD_TIMER_MANAGEMENT` : Comando proactivo TIMER MANAGEMENT = 0x27.

`static byte RES_CMD_PERF` : Resultado general del comando proactivo, llevado a cabo con éxito = 0x00

`static byte RES_CMD_PERF_BACKWARD_MOVE_REQ` : Resultado general del comando proactivo de solicitud de ir hacia atrás en la sesión de la aplicación proactiva = 0x11.

`static byte RES_CMD_PERF_HELP_INFO_REQ` : Resultado general del comando proactivo de solicitud de ayuda, llevado a cabo con éxito = 0x13.

`static byte RES_CMD_PERF_LIMITED_SERVICE` : Resultado general del comando proactivo, llevado a cabo de forma limitada = 0x06.

`static byte RES_CMD_PERF_MISSING_INFO` : Resultado general del comando proactivo, llevado a cabo con pérdida de información = 0x02.

`static byte RES_CMD_PERF_MODIF_CC_SIM` : Resultado general del comando proactivo llevado a cabo con modificación realizada por el control de llamada = 0x05.

`static byte RES_CMD_PERF_NO_RESP_FROM_USER` : Resultado general del comando proactivo llevado a cabo sin respuesta del usuario = 0x12.

`static byte RES_CMD_PERF_PARTIAL_COMPR` : Resultado general del comando proactivo llevado a cabo con comprensión parcial = 0x01.

`static byte RES_CMD_PERF_REFRESH_ADD_EF_READ` : Resultado general del comando proactivo REFRESH llevado a cabo con la lectura adicional de varios EF's = 0x03.

`static byte RES_CMD_PERF_REQ_ICON_NOT_DISP` : El resultado general del comando proactivo es que el icono solicitado no se ha mostrado en pantalla = 0x04.

`static byte RES_CMD_PERF_SESSION_TERM_USER` : El resultado general de la sesión del comando proactivo es que ha sido finalizada por el usuario = 0x10.

`static byte RES_CMD_PERF_USSD_TRANSAC_TERM` : El resultado general del comando proactivo es que la transacción USSD ha terminado = 0x14.

`static byte RES_CMD_PERF_WITH_MODIFICATION` : El resultado general del comando proactivo es que ha sido llevado a cabo con modificaciones = 0x07.

`static byte RES_ERROR_CMD_BEYOND_ME_CAPAB` : El resultado general es que el comando proactivo va más allá de la capacidad del ME = 0x30.

`static byte RES_ERROR_CMD_DATA_NOT_UNDERSTOOD` : El resultado general del comando proactivo es que no ha sido entendido por el ME = 0x32.

`static byte RES_ERROR_CMD_NUMBER_NOT_KNOWN` : El resultado general del comando proactivo es que el ME no conoce el número de comando = 0x33.

`static byte RES_ERROR_CMD_TYP_NOT_UNDERSTOOD` : El resultado general del comando proactivo es que el ME no entiende el tipo de comando = 0x31.

`static byte RES_ERROR_INTERACT_CC_SMSMO_BY_SIM` : El resultado general del comando proactivo es una interacción con una CC o un SMS MO controlado por la SIM = 0x39.

`static byte RES_ERROR_MULTIPLE_CARD_ERROR` : El resultado general de un comando proactivo que implica a múltiples tarjetas es un error = 0x38.

`static byte RES_ERROR_REQ_VALUES_MISS` : El resultado general del comando proactivo es que los valores requeridos no se encuentran = 0x36.

`static byte RES_ERROR_SMS_RP_ERROR` : El resultado general del comando proactivo es un SMS RP-ERROR = 0x35

`static byte RES_ERROR_SS_RETURN_ERROR` : El resultado general del comando proactivo es un SS return error = 0x34

`static byte RES_ERROR_USSD_RETURN_ERROR` : El resultado general del comando proactivo es un USSD return error = 0x37.

`static byte RES_TEMP_PB_IN_CONTR_TIMER_STATE` : El resultado general del comando proactivo es que la acción entra en contradicción con el estado del contador = 0x24.

`static byte RES_TEMP_PB_INTERACT_CC_BY_SIM` : El resultado general del comando proactivo es que interacciona con el control de llamadas de la SIM = 0x25.

`static byte RES_TEMP_PB_LAUNCH_BROWSER` : El resultado general del comando proactivo es un error genérico en el lanzamiento del navegador = 0x26.

`static byte RES_TEMP_PB_ME_UNABLE_PROC` : El resultado general del comando proactivo es que actualmente el ME no es capaz de procesar comandos = 0x20.

`static byte RES_TEMP_PB_SESSION_TERM_USER` : El resultado general del comando proactivo es que actualmente la red no es capaz de procesar el comando = 0x21.

`static byte RES_TEMP_PB_USER_CLEAR_CALL` : El resultado general del comando proactivo es que el usuario cortó la llamada antes de la conexión = 0x23.

`static byte RES_TEMP_PB_USER_REJECT_CALL_REQ` : El resultado general del comando proactivo es que el usuario no aceptó la solicitud de establecimiento de llamada = 0x22.

`static byte SW1_RP_ACK` : Inclusión, en la palabra de estado 1, del mensaje RP_ACK para que el ME lo transmita a la red = 0x9F.

`static byte SW1_RP_ERROR` : Inclusión, en la palabra de estado 1, del mensaje RP_ERROR para que el ME lo transmita a la red = 0x9E.

`static byte TAG_ADDRESS` : En objetos Simple-TLV representa la etiqueta de dirección = 0x06.

`static byte TAG_ALPHA_IDENTIFIER` : En objetos Simple-TLV representa la etiqueta Alpha Identifier = 0x05.

`static byte TAG_AT_COMMAND` : En objetos Simple-TLV representa la etiqueta de comando AT = 0x28.

`static byte TAG_AT_RESPONSE` : En objetos Simple-TLV representa la etiqueta de respuesta AT = 0x29.

`static byte TAG_BC_REPEAT_INDICATOR` : En objetos Simple-TLV representa la etiqueta del indicador de repetición BC = 0x2A.

`static byte TAG_BCCH_CHANNEL_LIST` : En objetos Simple-TLV representa la etiqueta de lista de canales BCCH = 0x1D.

`static byte TAG_BEARER` : En objetos Simple-TLV representa la etiqueta de Bearer = 0x32.

`static byte TAG_BROWSER_IDENTITY` : En objetos Simple-TLV representa el identificador del navegador = 0x30.

`static byte TAG_BROWSER_TERMINATION_CAUSE` : En objetos Simple-TLV representa el motivo de la terminación del navegador = 0x34.

`static byte TAG_C_APDU` : En objetos Simple-TLV representa la etiqueta C-APDU = 0x22

`static byte TAG_CALL_CONTROL_REQUESTED_ACTION` : En objetos Simple-TLV representa la acción solicitada de control de llamadas = 0x27.

`static byte TAG_CALLED_PARTY_SUBADDRESS` : En objetos Simple-TLV representa la etiqueta de subdireccionamiento de la parte llamada = 0x08.

`static byte TAG_CAPABILITY_CONFIGURATION_PARAMETERS` : En objetos Simple-TLV representa la etiqueta de parámetros de configuración de capacidad = 0x07.

`static byte TAG_CARD_ATR` : En objetos Simple-TLV representa la etiqueta Card ATR = 0x21.

`static byte TAG_CARD_READER_IDENTIFIER` : En objetos Simple-TLV representa la etiqueta del identificador de lector de tarjetas = 0x3A.

`static byte TAG_CARD_READER_STATUS` : En objetos Simple-TLV representa la etiqueta de estado del lector de tarjetas = 0x20.

`static byte TAG_CAUSE` : En objetos Simple-TLV representa la etiqueta de causa = 0x1A.

`static byte TAG_CELL_BROADCAST_PAGE` : En objetos Simple-TLV representa la etiqueta de la página Cell Broadcast = 0x0C.

`static byte TAG_COMMAND_DETAILS` : En objetos Simple-TLV representa la etiqueta de detalles del comando = 0x01.

`static byte TAG_DATE_TIME_AND_TIME_ZONE` : En objetos Simple-TLV representa la etiqueta de fecha-hora y zona horaria = 0x26.

`static byte TAG_DEFAULT_TEXT` : En objetos Simple-TLV representa la etiqueta de texto por defecto = 0x17.

`static byte TAG_DEVICE_IDENTITIES` : En objetos Simple-TLV representa la etiqueta de identificadores del dispositivo = 0x02.

`static byte TAG_DTMF_STRING` : En objetos Simple-TLV representa la etiqueta de cadena DTMF = 0x2C.

`static byte TAG_DURATION` : En objetos Simple-TLV representa la etiqueta de duración = 0x04.

`static byte TAG_EVENT_LIST` : En objetos Simple-TLV representa la etiqueta de lista de eventos = 0x19.

`static byte TAG_FILE_LIST` : En objetos Simple-TLV representa la etiqueta de lista de ficheros = 0x12.

`static byte TAG_HELP_REQUEST` : En objetos Simple-TLV representa la etiqueta de solicitud de ayuda = 0x15.

`static byte TAG_ICON_IDENTIFIER` : En objetos Simple-TLV representa la etiqueta de identificador de icono = 0x1E.

`static byte TAG_IMEI` : En objetos Simple-TLV representa la etiqueta de IMEI = 0x14.

`static byte TAG_IMMEDIATE_RESPONSE` : En objetos Simple-TLV representa la etiqueta de respuesta inmediata = 0x2B.

`static byte TAG_ITEM` : En objetos Simple-TLV representa la etiqueta de ítem = 0x0F.

`static byte TAG_ITEM_ICON_IDENTIFIER_LIST` : En objetos Simple-TLV representa la etiqueta de lista de identificadores de iconos de un ítem = 0x1F.

`static byte TAG_ITEM_IDENTIFIER` : En objetos Simple-TLV representa la etiqueta de identificador de un ítem = 0x10.

`static byte TAG_ITEMS_NEXT_ACTION_INDICATOR` : En objetos Simple-TLV representa la etiqueta de indicador de los ítems implicados en la próxima acción = 0x18.

`static byte TAG_LANGUAGE` : En objetos Simple-TLV representa la etiqueta de lengua = 0x2D.

`static byte TAG_LOCATION_INFORMATION` : En objetos Simple-TLV representa la etiqueta de información de localización = 0x13.

`static byte TAG_LOCATION_STATUS` : En objetos Simple-TLV representa la etiqueta de estado de la localización = 0x1B.

`static byte TAG_NETWORK_MEASUREMENT_RESULTS` : En objetos Simple-TLV representa la etiqueta de resultados de las medidas de las red = 0x16.

`static byte TAG_PROVISIONING_REFERENCE_FILE` : En objetos Simple-TLV representa la etiqueta de aprovisionamiento de la referencia de un archivo = 0x33.

`static byte TAG_R_APDU` : En objetos Simple-TLV representa la etiqueta de R-APDU = 0x23.

`static byte TAG_RESPONSE_LENGTH` : En objetos Simple-TLV representa la etiqueta de longitud de la respuesta = 0x11.

`static byte TAG_RESULT` : En objetos Simple-TLV representa la etiqueta de resultado = 0x03.

`static byte TAG_SET_CR` : En objetos Simple-TLV representa la máscara para ajustar la bandera CR de cualquier etiqueta de un Simple-TLV = 0x80.

`static byte TAG_SET_NO_CR` : En objetos Simple-TLV representa la máscara para resetear la bandera CR de cualquier etiqueta de un Simple-TLV = 0x7F.

`static byte TAG_SMS_TPDU` : En objetos Simple-TLV representa la etiqueta SMS TPDU = 0x0B.

`static byte TAG_SS_STRING` : En objetos Simple-TLV representa la etiqueta SS String = 0x09.

`static byte TAG_TEXT_STRING` : En objetos Simple-TLV representa la etiqueta Text String = 0x0D.

`static byte TAG_TIMER_IDENTIFIER` : En objetos Simple-TLV representa la etiqueta del identificador del contador = 0x24.

`static byte TAG_TIMER_VALUE` : En objetos Simple-TLV representa la etiqueta del valor del contador = 0x25.

`static byte TAG_TIMING_ADVANCE` : En objetos Simple-TLV representa la etiqueta del avance temporal = 0x2E.

`static byte TAG_TONE` : En objetos Simple-TLV representa la etiqueta de zona = 0x0E.

`static byte TAG_TRANSACTION_IDENTIFIER` : En objetos Simple-TLV representa la etiqueta de identificador de transacción = 0x1C.

`static byte TAG_URL` : En objetos Simple-TLV representa la etiqueta de URL = 0x31.

`static byte TAG_USSD_STRING` : En objetos Simple-TLV representa la etiqueta de USSD String = 0x0A

`static byte TLV_FOUND_CR_NOT_SET` : Es un posible resultado del método `findTLV` si se encuentra el TLV con la bandera CR no ajustada = 0x02.

`static byte TLV_FOUND_CR_SET` : Es un posible resultado del método `findTLV` si se encuentra el TLV con la bandera CR ajustada = 0x01.

`static byte TLV_LENGTH_CODED_2BYTES` : Valor del primer byte de un campo de longitud TLV con un valor de campo mayor de 127 bytes = 0x81.

`static byte TLV_NOT_FOUND` : Es un posible resultado del método `findTLV` si no se encuentra el TLV en el manejador = 0x00.

15.1.2 INTERFAZ `TOOLKITINTERFACE`

Esta interfaz debe ser implementada por un applet toolkit (que extenderá la clase `javacard.framework.Applet`) para que el Toolkit Handler pueda dispararlo de acuerdo con la información de registro. El applet toolkit tendrá que implementar el

método compartido `processToolkit` para que se le pueda notificar los eventos explicados en la página 170.

15.1.2.1 Métodos

`public void processToolkit(byte event)` : Este método es el método estándar para manejar eventos toolkit de un applet toolkit. El Toolkit Handler lo llamará para procesar el evento toolkit actual. Este método es invocado para notificar los eventos registrados. El parámetro `event` indica el tipo de evento que será procesado. Este método podrá lanzar una `ToolkitException`.

15.2 CLASES

15.2.1 CLASE `EDITHANDLER`

`public abstract class EditHandler extends ViewHandler` : Esta clase es la clase base para la construcción de una lista de elementos TLV simples. Esta clase es capaz de manejar un Simple-TLV con un valor de campo no mayor de 255 bytes.

15.2.1.1 Métodos

`public void appendArray(byte[] buffer, short offset, short length)` : Este método añade un buffer al final de un buffer que se encuentra dentro de `EditHandler`. Si esta operación tiene éxito, no se modifica el TLV seleccionado. Si después se van a usar los métodos de manipulación de TLV's con el manejador, el applet debería mantener la estructura de la lista TLV del manejador en el array que ha sido añadido (por ejemplo, la longitud del elemento TLV debería codificarse de acuerdo con la ISO 7816-6). El parámetro `buffer` contiene el buffer que contienen los `length` bytes de datos (a partir de la posición dada por `offset`) que se van a copiar. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `HANDLER_OVERFLOW` (si el buffer de `EditHandler` es demasiado pequeño como para añadir los datos solicitados) y `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado).

`public void appendTLV(byte tag, byte value)` : Este método añade un elemento TLV a la lista TLV actual. Este método es práctico para añadir elementos formados por un solo byte (como el identificador de un ítem o un tono). Si esta operación tiene éxito, no se modifica el TLV seleccionado. El parámetro `tag` es la etiqueta del TLV a añadir, incluyendo la bandera CR (Comprehension Required flag). `value` representa el valor de un byte del elemento TLV. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `HANDLER_OVERFLOW` (si el buffer de `EditHandler` es demasiado pequeño como para añadir los datos solicitados) y `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado).

`public void appendTLV(byte tag, byte[] value, short valueOffset, short valueLength)` : Añade un elemento TLV a la lista TLV actual (en formato de array de bytes).). Si esta operación tiene éxito, no se modifica el TLV seleccionado. El parámetro `tag` es la etiqueta del TLV a añadir, incluyendo la

bandera CR (Comprehension Required flag). `value` es el buffer que contiene el valor (a partir de la posición `valueOffset`) del elemento TLV. `valueLength` da la longitud del valor del elemento TLV a añadir. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `HANDLER_OVERFLOW` (si el buffer de `EditHandler` es demasiado pequeño como para añadir los datos solicitados), `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado) y `BAD_INPUT_PARAMETER` (si `valueLength` es mayor que 255).

`public void appendTLV(byte tag, byte value1, byte value2)` : Añade un elemento TLV a la lista TLV actual (elemento de 2 bytes). Este método es práctico para añadir elementos formados por dos bytes como los identificadores de dispositivos, longitud de la respuesta o de la duración. Si esta operación tiene éxito, no se modifica el TLV seleccionado. El parámetro `tag` indica la etiqueta del TLV a añadir, incluyendo la bandera CR (Comprehension Required flag). `value1` es el primer byte (msb) del valor TLV. `value2` es el segundo byte del valor TLV. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `HANDLER_OVERFLOW` (si el buffer de `EditHandler` es demasiado pequeño como para añadir los datos solicitados) y `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado).

`public void appendTLV(byte tag, byte value1, byte[] value2, short value2Offset, short value2Length)` : Añade un elemento TLV a la lista TLV actual (con formato de un byte y de un array de bytes). Si esta operación tiene éxito, no se modifica el TLV seleccionado. El parámetro `tag` indica la etiqueta del TLV a añadir, incluyendo la bandera CR (Comprehension Required flag). `value1` representa al primer byte del campo. `value2` es el buffer que contiene el resto (a partir de la posición `value2Offset`) del campo TLV. `value2Length` es el valor de la longitud del resto del campo TLV a añadir. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `HANDLER_OVERFLOW` (si el buffer de `EditHandler` es demasiado pequeño como para añadir los datos solicitados), `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado) y `BAD_INPUT_PARAMETER` (si `value2Length` es mayor que 254).

`public void clear()` : Limpia una lista TLV de un `EditHandler` y resetea el TLV actualmente seleccionado. El método podrá lanzar una `ToolkitException` con el código de causa `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado).

15.2.2 CLASE `ENVELOPEHANDLER`

`public final class EnvelopeHandler extends ViewHandler` : La clase `EnvelopeHandler` contiene los métodos básicos para manejar el campo de datos del comando ENVELOPE (enviado por el ME). El applet toolkit usará esta clase para tener acceso a la información del comando ENVELOPE actual. No hay un constructor disponible para el applet toolkit. La clase `EnvelopeHandler` es un objeto temporal de punto de entrada al JCRE. La única manera de obtener una referencia del `EnvelopeHandler` es a través del método estático `getTheHandler()`.

15.2.2.1 Métodos

`public byte getEnvelopeTag()` : Devuelve la etiqueta BER-TLV de ENVELOPE.

`public byte getItemIdentifier()` : Devuelve el identificador (de un byte) del ítem a partir del primer TLV identificador del ítem que se encuentra en el campo de datos del ENVELOPE actual. Si el elemento está disponible será el TLV seleccionado. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `UNAVAILABLE_ELEMENT` (en el caso de que no esté disponible un elemento TLV) y `OUT_OF_TLV_BOUNDARIES` (si el identificador del ítem se ha perdido en el Simple TLV identificador del ítem).

`public short getSecuredDataLength()` : Busca la longitud de los datos seguros, que vienen en el paquete del comando, del primer Simple TLV (contenido en el `EnvelopeHandler`) de una SMS TPDU o de una página Cell Broadcast. Esto se puede usar en el caso de que se den los eventos: `EVENT_FORMATTED_SMS_PP_ENV`, `EVENT_FORMATTED_SMS_PP_UPD`, si el campo TP-UD del SMS tiene un formato que cumple con el GSM 03.48 Single Short Message; `EVENT_FORMATTED_SMS_CB`, si la página Cell Broadcast Page tiene un formato que cumple con el GSM 03.48. Si el elemento está disponible será el TLV seleccionado. El método devuelve la longitud de los datos seguros contenidos en el primer elemento TLV (sin bytes de relleno) de una SMS TPDU o de una página Cell Broadcast. Si la longitud de los datos seguros es cero, no se lanzará ninguna excepción. El método podrá lanzar una `ToolkitException` con el código de causa `UNAVAILABLE_ELEMENT` (en el caso de que no esté disponible un elemento TLV de una SMS TPDU o de una página Cell Broadcast o porque los datos tengan un formato incorrecto).

`public short getSecuredDataOffset()` : Busca los datos seguros, que vienen en el paquete del comando, del primer Simple TLV (contenido en el `EnvelopeHandler`) de una SMS TPDU o de una página Cell Broadcast. Esto se puede usar en el caso de que se den los eventos: `EVENT_FORMATTED_SMS_PP_ENV`, `EVENT_FORMATTED_SMS_PP_UPD`, si el campo TP-UD del SMS tiene un formato que cumple con el GSM 03.48 Single Short Message; `EVENT_FORMATTED_SMS_CB`, si la página Cell Broadcast Page tiene un formato que cumple con el GSM 03.48. Si el elemento está disponible será el TLV seleccionado. El método devuelve el offset del primer byte de los datos seguros contenidos en el primer elemento TLV de una SMS TPDU o de una página Cell Broadcast. El método podrá lanzar una `ToolkitException` con el código de causa `UNAVAILABLE_ELEMENT` (en el caso de que no esté disponible un elemento TLV de una SMS TPDU o de una página Cell Broadcast o porque los datos tengan un formato incorrecto).

`public static EnvelopeHandler getTheHandler()` : Devuelve la referencia de la única instancia de la clase `EnvelopeHandler` del sistema. El applet conseguirá la referencia del manejador en el instante en que se produzca su disparo, al comienzo del método `processToolkit`. El método podrá lanzar una `ToolkitException` con el código de causa `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado).

`public short getTPUDLOffset()` : Busca el campo TP-UDL en el primer elemento TPDU TLV en el campo de datos de ENVELOPE. Este método se puede usar en el caso de que se den los eventos: `EVENT_FORMATTED_SMS_PP_ENV`, `EVENT_FORMATTED_SMS_PP_UPD`, `EVENT_UNFORMATTED_SMS_PP_ENV` y `EVENT_UNFORMATTED_SMS_PP_UPD`. . Si el elemento está disponible será el TLV seleccionado. El método devuelve el offset TPUDL del primer elemento TPDU TLV si el TPUDL existe. La longitud del TPUD se puede recuperar mediante el método `getValueByte` de la clase `Handler`. El método podrá lanzar una `ToolkitException` con el código de causa `UNAVAILABLE_ELEMENT` (en el caso de que no esté disponible un elemento TPDU TLV o si no existe el campo TPUDL).

15.2.3 CLASE `ENVELOPERESPONSEHANDLER`

`public final class EnvelopeResponseHandler extends EditHandler` : La clase `EnvelopeResponseHandler` contiene los métodos básicos para manejar los campos de datos de la respuesta a ENVELOPE. El applet toolkit usará esta clase para editar la respuesta al comando ENVELOPE actual. No hay ningún constructor disponible para el applet toolkit. La clase `EnvelopeResponseHandler` es un objeto temporal de punto de entrada al JCRE. La única manera de conseguir una referencia de `EnvelopeResponseHandler` es mediante el método estático `getTheHandler()`.

15.2.3.1 Métodos

`public static EnvelopeResponseHandler getTheHandler()` : Devuelve la única referencia de la instancia de la clase `EnvelopeResponseHandler` del sistema. El applet conseguirá la referencia del manejador en el instante en que se produzca su disparo, al comienzo del método `processToolkit`. El método podrá lanzar una `ToolkitException` con el código de causa `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado).

`public void post(byte statusType)` : Prepara la respuesta al comando ENVELOPE. Este método se debería usar en el caso de un comando ENVELOPE que incluya un SMS-PP para descarga de datos. El parámetro `statusType` indica el estado a enviar al ME (`SW1_RP_ACK` ó `SW1_RP_ERROR`). El método podrá lanzar una `ToolkitException` con el código de causa `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado).

`public void postAsBERTLV(byte statusType, byte tag)` : Prepara la respuesta al comando ENVELOPE con una estructura BER TLV. Este método se debería usar en el caso de un comando ENVELOPE que incluya control de llamadas ejercido por la SIM o control de mensajes cortos originados en el ME ejercido por la SIM. El valor de la etiqueta se usa para ajustar el resultado del control de llamadas y mensajes originados es en ME ejercidos por la SIM. El parámetro `statusType` indica el estado a enviar al ME (`SW1_RP_ACK` ó `SW1_RP_ERROR`). `tag` es la etiqueta BER a usar al comienzo de la lista Simple-TLV. El método podrá lanzar una `ToolkitException` con el código de causa `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado).

15.2.4 CLASE `MEPROFILE`

`public final class MEProfile extends java.lang.Object` : La clase `MEProfile` contiene métodos para preguntar el perfil del terminal móvil, en cuanto al SIM Application Toolkit y en el caso en que ese perfil ya haya sido ajustado por la APDU de comando TERMINAL PROFILE. La siguiente tabla da el valor del índice según la facilidad a comprobar. Esta clase solo contiene métodos estáticos, no es necesario crear ninguna instancia de esta clase.

Tabla 24: Facilidades a comprobar

Facilidad	Índice
Profile download	0
SMS-PP data download	1
Cell Broadcast data download	2
Menu selection	3
'9E XX' response code for SIM data download error	4
Timer Expiration	5
USSD string data object supported in Call Control	6
Envelope Call Control sent during auto redial	7
Command result	8
Call Control by SIM	9
Cell identity included in Call Control by SIM	10
MO short message control by SIM11	11
Handling of the alpha identifier, user indication	12
UCS2 Entry supported	13
UCS2 Display supported	14
Display of the extension Text	15
Proactive SIM: Display Text	16
Proactive SIM: Get Inkey	17
Proactive SIM: Get Input	18
Proactive SIM: More Time	19
Proactive SIM: Play Tone	20
Proactive SIM: Poll Interval	21
Proactive SIM: Polling Off	22
Proactive SIM: Refresh	23
Proactive SIM: Select Item	24
Proactive SIM: Send Short Message	25
Proactive SIM: Send SS	26
Proactive SIM: Send USSD	27
Proactive SIM: Set Up Call	28
Proactive SIM: Set Up Menu	29
Proactive SIM: Provide Local Information	30
Proactive SIM: Provide Local Information (NMR)	31
Proactive SIM: Set Up Event List	32
Event: MT call	33
Event: Call connected	34
Event: Call disconnected	35
Event: Location status	36
Event: User activity	37
Event: Idle screen available	38
Event: Card reader status	39
Event: Language selection	40
Event: Browser termination	41
RFU (Reserved for Future Use)	42
RFU	43
RFU	44

RFU	45
RFU	46
RFU	47
Proactive SIM: Power ON Card	48
Proactive SIM: Power OFF Card	49
Proactive SIM: Perform Card APDU	50
Proactive SIM: Get Reader Status (reader status)	51
Proactive SIM: Get Reader Status (reader ident.)	52
RFU	53
RFU	54
RFU	55
Proactive SIM: Timer Management (start, stop)	56
Proactive SIM: Timer Management (get cur. value)	57
Proactive SIM: Provide Local Info (date,time...)	58
Binary choice in Get Inkey	59
Set Up Idle Mode Text	60
Run AT Command	61
2nd Alpha Id in Set Up Call	62
2nd Capability configuration par. in Call Control	63
Sustained Display Text	64
Send DTMF	65
Proactive SIM: Provide Local Info. (BCCH)	66
Proactive SIM: Provide Local Info. (language)	67
Proactive SIM: Provide Local Info. (Timing Adv.)	68
Proactive SIM: Language Notification	69
Proactive SIM: Launch Browser	70
RFU	71
Soft keys support for Select Item	72
Soft keys support for Set Up Menu	73
RFU	74
RFU	75
RFU	76
RFU	77
RFU	78
RFU	79
Maximum number of softkeys available (b0)	80
Maximum number of softkeys available (b1)	81
Maximum number of softkeys available (b2)	82
Maximum number of softkeys available (b3)	83
Maximum number of softkeys available (b4)	84
Maximum number of softkeys available (b5)	85
Maximum number of softkeys available (b6)	86
Maximum number of softkeys available (b7)	87
RFU	88
RFU	89
RFU	90
RFU	91
RFU	92
RFU	93
RFU	94
RFU	95
RFU	96
RFU	97
RFU	98
RFU	99
RFU	100
RFU	101
RFU	102

RFU	103
Nb of characters down ME display (b0)	104
Nb of characters down ME display (b1)	105
Nb of characters down ME display (b2)	106
Nb of characters down ME display (b3)	107
Nb of characters down ME display (b4)	108
RFU	109
RFU	110
Screen Sizing parameters supported	111
Nb of characters accross ME display (b0)	112
Nb of characters accross ME display (b1)	113
Nb of characters accross ME display (b2)	114
Nb of characters accross ME display (b3)	115
Nb of characters accross ME display (b4)	116
Nb of characters accross ME display (b5)	117
Nb of characters accross ME display (b6)	118
Variable size fonts supported	119
Display can be resized	120
Text Wrapping supported	121
Text Scrolling supported	122
RFU	123
RFU	124
Width reduction when in a menu (b0)	125
Width reduction when in a menu (b1)	126
Width reduction when in a menu (b2)	127
RFU	128
RFU	129
RFU	130
RFU	131
RFU	132
RFU	133

15.2.4.1 Métodos

`public static boolean check(byte index)` : Comprueba una facilidad en el perfil del terminal móvil. El parámetro `index` es el número de facilidad a comprobar, según la tabla presentada arriba. El método devuelve `true` si el ME soporta la facilidad, `false` en caso contrario. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `ME_PROFILE_NOT_AVAILABLE` (si los datos del perfil del terminal no están disponibles) y `BAD_INPUT_PARAMETER` (si `index` tiene un valor negativo).

`public static boolean check(byte[] mask, short offset, short length)` : Comprueba un conjunto de facilidades en el perfil del terminal. El método comprueba todas las facilidades correspondientes a los bits puestos a 1 en el buffer de máscara. El parámetro `mask` es un array de bytes que contiene la máscara (que empieza a partir de la posición dada por `offset`) a comparar con el perfil. `length` es la longitud de la máscara (por lo menos 1). El método devuelve `true` si el AND entre el perfil del ME y la máscara, es igual a la máscara. Si `length` es igual a 0, el método devuelve true. El método podrá lanzar una `ToolkitException` con el código de causa `ME_PROFILE_NOT_AVAILABLE` (si los datos del perfil del terminal no están disponibles).

`public static boolean check(short index)` : Comprueba una facilidad en el perfil del terminal móvil. El parámetro `index` es el número de facilidad a comparar, según la tabla presentada arriba. El método devuelve `true` si el ME soporta la facilidad, `false` en caso contrario. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `ME_PROFILE_NOT_AVAILABLE` (si los datos del perfil del terminal no están disponibles) y `BAD_INPUT_PARAMETER` (si `index` tiene un valor negativo).

`public static short copy(short startOffset, byte[] dstBuffer, short dstOffset, short dstLength)` : Copia parte del perfil del terminal móvil en un buffer. Los valores que se encuentren fuera de los datos disponibles en el perfil del ME se consideran puestos a 0. El parámetro `startOffset` indica la posición del primer byte del perfil del ME que será copiado. `dstBuffer` es el array de bytes de destino donde se copiarán los `dstLength` bytes a partir de la posición `dstOffset`. El método devuelve `stOffset + dstLength`. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `ME_PROFILE_NOT_AVAILABLE` (si los datos del perfil del terminal no están disponibles) y `BAD_INPUT_PARAMETER` (si `index` tiene un valor negativo).

`public static short getValue(short indexMSB, short indexLSB)` : Devuelve el valor binario de un parámetro, delimitado por dos índices, a partir del perfil del terminal móvil. El parámetro `indexMSB` es el índice del bit más significativo del perfil del ME. `indexLSB` es el índice del bit menos significativo del perfil del ME. El método devuelve un valor binario del campo de datos indicado y que procede del perfil de ME. El bit `indexLSB` de los datos del perfil del ME es el bit menos significativo en el valor tipo short devuelto. Si hace falta relleno, el valor devuelto se rellena con ceros hacia la izquierda. Los valores que están fuera de los datos disponibles del perfil del ME se consideran puestos a 0. Un ejemplo de valor devuelto es el siguiente: si `indexMSB`=108 e `indexLSB`=104, el valor devuelto es el número de caracteres que caben hacia abajo en la pantalla del ME. Otro ejemplo: Si `indexMSB`=31 e `indexLSB`=16, el valor devuelto es un tipo short construido a partir del cuarto y tercer byte del perfil del ME con el cuarto byte como el byte más significativo. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `ME_PROFILE_NOT_AVAILABLE` (si los datos del perfil del terminal no están disponibles) y `BAD_INPUT_PARAMETER` (si `indexMSB>=indexLSB` ó `indexMSB<indexLSB` ó `indexMSB<0` ór `indexLSB<0`).

15.2.5 CLASE PROACTIVEHANDLER

`public final class ProactiveHandler extends EditHandler` : Esta clase es la clase base para la definición de los comandos proactivos. Los métodos de bajo nivel, como `init()`, `appendTLV()`... se usaran para tratar comandos proactivos genéricos (estándares o propuestos en futuras definiciones). La clase `ProactiveHandler` es un objeto temporal de punto de entrada al JCRE. Los applets toolkit, que necesiten enviar comandos proactivos, llamarán al método estático `getTheHandler()` para conseguir la referencia de la instancia del sistema.

15.2.5.1 Métodos

`public static ProactiveHandler getTheHandler()` : Devuelve la referencia a la instancia del sistema de la clase `ProactiveHandler`. El applet conseguirá la referencia del manejador en el instante en que se produzca su disparo, al comienzo del método `processToolkit`. El método podrá lanzar una `ToolkitException` con el código de causa `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado).

`public void init(byte type, byte qualifier, byte dstDevice)` : Inicializa el próximo comando proactivo con el TLV que especifica los detalles del comando y los identificadores de los dispositivos. El dispositivo de origen siempre es la tarjeta SIM. El método genera el número del comando. Las banderas CR (Comprehension Required) se ajustan. Después de la invocación del método no está seleccionado el TLV. El parámetro `type` indica el tipo de comando y `qualifier` el calificador del comando. `dstDevice` indica el dispositivo de destino.

`public void initDisplayText(byte qualifier, byte dcs, byte[] buffer, short offset, short length)` : Construye un comando proactivo de muestra de texto por pantalla sin enviar el comando. Todas las banderas CR (Comprehension Required) están ajustadas a 1. Después de la invocación del método no está seleccionado el TLV. El parámetro `qualifier` es el calificador del comando DISPLAY TEXT. `dcs` da el esquema de codificación de los datos. `buffer` es la referencia del buffer donde se encuentra la cadena de texto a mostrar por pantalla (a partir de la posición `offset` del buffer). `length` es la longitud de la cadena de texto que se encuentra en `buffer`. El método podrá lanzar una `ToolkitException` con el código de causa `HANDLER_OVERFLOW` (si el buffer del `ProactiveHandler` es pequeño para poner los datos solicitados).

`public void initGetInkey(byte qualifier, byte dcs, byte[] buffer, short offset, short length)` : Construye un comando proactivo GET INKEY sin enviar el comando. Todas las banderas CR (Comprehension Required) están puestas a 1. Después de la invocación del método no está seleccionado el TLV. El parámetro `qualifier` es el calificador del comando GET INKEY. `dcs` da el esquema de codificación de los datos. `buffer` es la referencia del buffer donde se encuentra la cadena de texto a mostrar en pantalla (a partir de la posición `offset` del buffer). `length` es la longitud de la cadena de texto que se encuentra en `buffer`. El método podrá lanzar una `ToolkitException` con el código de causa `HANDLER_OVERFLOW` (si el buffer del `ProactiveHandler` es pequeño para poner los datos solicitados). El método podrá lanzar una `ToolkitException` con el código de causa `HANDLER_OVERFLOW` (si el buffer del `ProactiveHandler` es pequeño para poner los datos solicitados).

`public void initGetInput(byte qualifier, byte dcs, byte[] buffer, short offset, short length, short minRespLength, short maxRespLength)` : Inicializa la construcción de un comando proactivo GET INPUT. Todas las banderas CR (Comprehension Required) están ajustadas a 1. Los siguientes parámetros del comando quizás se añadan antes de enviarlo. Después de la invocación del método no está seleccionado el TLV. El parámetro `qualifier` es el calificador del comando GET INPUT. `dcs` da el esquema de codificación de los datos. `buffer` es la

referencia del buffer donde se encuentra la cadena de texto a mostrar en pantalla (a partir de la posición `offset` del buffer). `length` es la longitud de la cadena de texto que se encuentra en `buffer`. `minRespLength` es la longitud mínima de la cadena de texto de respuesta. `maxRespLength` es la longitud máxima de la cadena de texto de respuesta. El método podrá lanzar una `ToolkitException` con el código de causa `HANDLER_OVERFLOW` (si el buffer del `ProactiveHandler` es pequeño para poner los datos solicitados). El método podrá lanzar una `ToolkitException` con el código de causa `HANDLER_OVERFLOW` (si el buffer del `ProactiveHandler` es pequeño para poner los datos solicitados).

`public byte send()` : Envía el comando proactivo actual. Devuelve el resultado general del comando (el primer byte del TLV de resultado en un TERMINAL RESPONSE). El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `UNAVAILABLE_ELEMENT` (si el TLV de resultado se pierde) y `OUT_OF_TLV_BOUNDARIES` (si el byte de resultado general del TLV de resultado se pierde).

15.2.6 CLASE `PROACTIVERESPONSEHANDLER`

`public final class ProactiveResponseHandler extends ViewHandler` : La clase `ProactiveResponseHandler` contiene métodos básicos para tratar el campo de datos de un TERMINAL RESPONSE. El applet toolkit usará esta clase para conseguir la respuesta de los comandos proactivos. No hay un constructor disponible para los applets toolkit. La clase `ProactiveResponseHandler` es un objeto temporal de punto de entrada al JCRE. La única manera de obtener una referencia del `ProactiveResponseHandler` es a través del método estático `getTheHandler()`.

15.2.6.1 Métodos

`public short copyAdditionalInformation(byte[] dstBuffer, short dstOffset, short dstLength)` : Copia una parte del campo adicional de información que se encuentra a partir del primer elemento TLV de resultado que pertenece al campo de datos de respuesta actual. Si el elemento está disponible será el TLV seleccionado. El parámetro `dstBuffer` da una referencia al buffer de destino. `dstOffset` indica la posición donde empezar a copiar los `dstLength` bytes de datos en el buffer de destino. El método devuelve `dstOffset + longitud del valor copiado`. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `UNAVAILABLE_ELEMENT` (si el TLV de resultado no está disponible) y `OUT_OF_TLV_BOUNDARIES` (si `dstLength` es mayor que el valor del campo del TLV disponible).

`public short copyTextString(byte[] dstBuffer, short dstOffset)` : Este método copia el valor de una cadena de texto a partir del primer elemento TLV de una cadena de texto que pertenece al campo de datos de respuesta actual. El esquema de codificación de los datos no se copia. Si el elemento está disponible será el TLV seleccionado. El parámetro `dstBuffer` da una referencia al buffer de destino. `dstOffset` indica la posición donde empezar a copiar los `dstLength` bytes de datos en el buffer de destino. El método devuelve `dstOffset +`

`longitud del valor copiado`. El método podrá lanzar una `ToolkitException` con el código de causa `UNAVAILABLE_ELEMENT` (si el elemento TLV de cadena de texto no está disponible).

`public short getAdditionalInformationLength()` : Devuelve la longitud del campo de información adicional a partir del primer elemento TLV de resultado que pertenece al campo de datos de respuesta actual. Si el elemento está disponible será el TLV seleccionado. El método podrá lanzar una `ToolkitException` con el código de causa `UNAVAILABLE_ELEMENT` (si el elemento TLV de resultado no está disponible).

`public byte getGeneralResult()` : Devuelve el byte de resultado general (primer byte del TLV de resultado del TERMINAL RESPONSE) del comando proactivo. Si el elemento está disponible será el TLV seleccionado. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `UNAVAILABLE_ELEMENT` (si el TLV de resultado no está disponible) y `OUT_OF_TLV_BOUNDARIES` (si el byte del resultado general se pierde en el TLV de resultado).

`public byte getItemIdentifier()` : Devuelve el byte identificador del ítem a partir del primer elemento TLV de identificación de ítems y que pertenece al campo de datos de respuesta actual. Si el elemento está disponible será el TLV seleccionado. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `UNAVAILABLE_ELEMENT` (si el elemento TLV de identificación de ítems no está disponible) y `OUT_OF_TLV_BOUNDARIES` (si el byte del identificador del ítem se pierde en el TLV de identificación de ítems).

`public byte getTextStringCodingScheme()` : Devuelve el byte de esquema de codificación a partir del primer elemento TLV de cadena de texto y que pertenece al campo de datos de respuesta actual. Si el elemento está disponible será el TLV seleccionado. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `UNAVAILABLE_ELEMENT` (si el elemento TLV de cadena de texto no está disponible) y `OUT_OF_TLV_BOUNDARIES` (si el TLV de cadena de texto con una longitud de 0, es decir, no hay byte DCS).

`public short getTextStringLength()` : Devuelve el valor de la longitud de la cadena de texto a partir del primer elemento TLV de cadena de texto y que pertenece al campo de datos de respuesta actual. El byte de esquema de codificación de datos no se tiene en cuenta. Si el elemento está disponible será el TLV seleccionado. El método podrá lanzar una `ToolkitException` con el código de causa `UNAVAILABLE_ELEMENT` (si el elemento TLV de cadena de texto no está disponible).

`public static ProactiveResponseHandler getTheHandler()` : Devuelve la referencia de la instancia única de la clase `ProactiveResponseHandler`. El applet conseguirá la referencia del manejador en el instante en que se produzca su disparo, al comienzo del método `processToolkit`. El método podrá lanzar una `ToolkitException` con el código de causa `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado).

15.2.7 CLASE `TOOLKITREGISTRY`

`public final class ToolkitRegistry extends java.lang.Object` : La clase `ToolkitRegistry` ofrece servicios básicos y métodos que permiten que cualquier applet toolkit registre su configuración (eventos soportados) durante la fase de instalación y la posibilidad de cambiarla durante todo el tiempo de vida del applet. Cada applet toolkit conseguirá una referencia a su entrada en el registro con el método estático `getEntry`. El estado inicial de todos los eventos se borran. Las constantes relacionadas con los eventos están definidas en la interfaz `ToolkitConstants`.

15.2.7.1 Métodos

`public byte allocateTimer()` : Le pide al Toolkit Framework que le asigne un contador que el applet pueda gestionar. Llamando a este método, el applet se registra en el evento EVENT_TIMER_EXPIRATION del contador asignado. El contador se asigna al applet hasta que el applet lo libera explícitamente. Entonces, puede enviar comandos proactivos de gestión del contador del applet para iniciar la cuenta, pararla u obtener su valor. El método devuelve, al applet, el identificador del contador asignado. El método podrá lanzar una `ToolkitException` con el código de causa `NO_TIMER_AVAILABLE` (si todos los contadores están asignados o se ha alcanzado el número máximo de contadores que puede tener asignados un applet).

`public void changeMenuEntry(byte id, byte[] menuEntry, short offset, short length, byte nextAction, boolean helpSupported, byte iconQualifier, short iconIdentifier)` : Cambia el valor de la entrada de un menú. El estado por defecto de la entrada de un menú que se ha cambiado está 'disponible'. El valor del parámetro booleano `helpSupported` define el estado de registro del evento `EVENT_MENU_SELECTION_HELP_REQUEST`. El identificador de icono provisto, se añadirá a la lista de identificadores de iconos que se encuentra en la lista de ítems, si todos los applets registrados al EVENT_MENU_SELECTION lo proveen. El calificador de la lista de iconos transmitido al ME será el de icono no autoexplicativo, si uno de los applets registrados prefiere ese calificador. Después de la invocación de este método, durante la sesión actual de la tarjeta, el SIM Toolkit Framework actualizará dinámicamente el menú almacenado en el ME. El parámetro `id` es el identificador de entrada al menú suministrado por el método `initMenuEntry()`. `menuEntry` es la referencia a un array de bytes que contiene la cadena (de longitud `length`) de entrada al menú, a partir de la posición `offset` del array. `nextAction` es un byte que codifica la siguiente acción a realizar para esa entrada del menú. `helpSupported` vale `true` si hay ayuda para la entrada al menú. `iconQualifier` es el valor preferido para el calificador de la lista de iconos. `iconIdentifier` es el identificador del icono para la entrada al menú (0 significa que no hay icono). El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `MENU_ENTRY_NOT_FOUND` (si la entrada al menú no existe para ese applet) y `ALLOWED_LENGTH_EXCEEDED` (si la cadena de texto de la entrada al menú es mayor que el espacio asignado para ella).

`public void clearEvent(byte event)` : Borra un evento de la entrada del applet en el Toolkit Registry. El parámetro `event` es el valor del evento que quiere quitar del registro (entre -128 y 127). El método podrá lanzar una `ToolkitException`

con el código de causa `EVENT_NOT_ALLOWED` (si el evento es `EVENT_MENU_SELECTION`, `EVENT_MENU_SELECTION_HELP_REQUEST`, `EVENT_TIMER_EXPIRATION` ó `EVENT_STATUS_COMMAND`).

`public void disableMenuEntry(byte id)` : Deshabilita una entrada al menú. Este método no modifica el estado de registro en los eventos `EVENT_MENU_SELECTION` y `EVENT_MENU_SELECTION_HELP_REQUEST`. Después de la invocación a este método, durante la sesión actual de la tarjeta, el SIM Toolkit Framework actualizará dinámicamente el menú almacenado en el ME. El parámetro `id` es el identificador de la entrada al menú suministrado por el método `initMenuEntry()`. El método podrá lanzar una `ToolkitException` con el código de causa `MENU_ENTRY_NOT_FOUND` (si no existe la entrada al menú para este applet).

`public void enableMenuEntry(byte id)` : Habilita una entrada al menú. Este método no modifica el estado de registro en los eventos `EVENT_MENU_SELECTION` y `EVENT_MENU_SELECTION_HELP_REQUEST`. Después de la invocación de este método, durante la sesión actual de tarjeta, el SIM Toolkit Framework actualizará dinámicamente el menú almacenado en el ME. El parámetro `id` es el identificador de entrada al menú suministrado por el método `initMenuEntry()`. El método podrá lanzar una `ToolkitException` con el código de causa `MENU_ENTRY_NOT_FOUND` (si no existe la entrada al menú para este applet).

`public static ToolkitRegistry getEntry()` : El applet toolkit usa este método para obtener una referencia a su entrada en el Toolkit Registry, para que pueda manejar su estado de registro en los eventos toolkit. El método devuelve una referencia al objeto `ToolkitRegistry`. El método podrá lanzar una `ToolkitException` con el código de causa `REGISTRY_ERROR` (en el caso de que se produzca un error en el registro).

`public short getPollInterval()` : Devuelve el número de segundos de la duración ajustada por el applet toolkit llamante. Si la duración devuelta es igual a `POLL_NO_DURATION`, el applet toolkit no está registrado en el evento `EVENT_STATUS_COMMAND`. La duración devuelta quizás: sea un múltiplo del verdadero intervalo de tiempo de polling en el ME, sea diferente de la duración solicitada debido a las solicitudes de otros applets toolkit o debido a las limitaciones del ME actual, sea cambiado durante la sesión de tarjeta debido a solicitudes de otros applets, esté mal debido a generación directa de los comandos proactivos `POLL INTERVAL` ó `POLLING OFF`, no corresponda con el tiempo transcurrido debido a los comandos STATUS adicionales enviados por el ME. El método devuelve el número de segundos de la duración ajustada por el applet.

`public byte initMenuEntry(byte[] menuEntry, short offset, short length, byte nextAction, boolean helpSupported, byte iconQualifier, short iconIdentifier)` : Inicializa la siguiente entrada al menú asignada en el momento de la carga. El estado por defecto de la entrada al menú es ‘disponible'. El valor del parámetro booleano `helpSupported` define el estado del registro del applet en el evento `EVENT_MENU_SELECTION_HELP_REQUEST`. El applet está registrado en el `EVENT_MENU_SELECTION`. El identificador de icono provisto, se añadirá a la lista de identificadores de iconos, si todos los applets registrados al evento `EVENT_MENU_SELECTION` lo proveen. El calificador de la lista de iconos transmitida al

ME será el de icono no autoexplicativo si uno de los applets registrados prefiere ese calificador. Este método será llamado por el applet en el mismo orden de los parámetros de los ítems, definidos en el instante de la descarga del applet, si el applet tiene varias entradas en el menú. El applet inicializará todas sus entradas del menú durante su instalación. El parámetro `menuEntry` es una referencia de un array de bytes que contiene la cadena de texto (a partir de la posición `offset` y con longitud `length`) que define la entrada al menú. `nextAction` es un byte que codifica la próxima acción para esa entrada al menú (ó 0). `helpSupported` es igual a `true` si hay ayuda disponible para esa entrada al menú. `iconQualifier` es el valor preferido para el calificador de la lista de iconos. `iconIdentifier` es el identificador del icono para la entrada al menú (0 significa que no hay icono). El método devuelve el identificador unido a la entrada al menú inicializado. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `REGISTRY_ERROR` (si la entrada al menú no se puede inicializar) y `ALLOWED_LENGTH_EXCEEDED` (si la cadena de texto de la entrada al menú es mayor que el espacio asignado para ella).

`public boolean isEventSet(byte event)` : Permite conocer si un evento está ajustado en la entrada del applet en el Toolkit Registry. El parámetro `event` indica el valor del evento (entre -128 y 127). El método devuelve `true` si el evento está ajustado en la entrada del applet en el Toolkit Registry, `false` en caso contrario.

`public void releaseTimer(byte timerIdentifier)` : Libera un contador que ha sido asignado al applet llamado. El applet quita del registro al evento `EVENT_TIMER_EXPIRATION` con el identificador de contador indicado dado por `timerIdentifier`. El método podrá lanzar una `ToolkitException` con el código de causa `INVALID_TIMER_ID` (si el identificador del contador no está asignado a este applet).

`public void requestPollInterval(short duration)` : Solicita la duración del evento `EVENT_STATUS_COMMAND` del applet toolkit registrado. Debido a las duraciones diferentes solicitadas por otros applets toolkit o debido a limitaciones del ME, el SIM Toolkit Framework quizás ajuste otra duración. Este método se puede usar en cada instante, para solicitar una duración nueva. El parámetro `duration` especifica el número de segundos solicitados para el polling proactivo. El máximo valor de la duración es 15300 (255 minutos). Si la duración es igual a `POLL_NO_DURATION`, el applet llamante se quita del `EVENT_STATUS_COMMAND`, y el SIM Toolkit Framework quizás suministre el comando proactivo POLLING OFF. Si la duración es igual a `POLL_SYSTEM_DURATION`, el applet llamante se registra al evento `EVENT_STATUS_COMMAND` y deja que el SIM Toolkit Framework defina la duración. El método podrá lanzar una `ToolkitException` con el código de causa `REGISTRY_ERROR` (si la duración es mayor que el valor máximo).

`public void setEvent(byte event)` : Ajusta el evento en la entrada del applet en el Toolkit Registry. No se lanzará ninguna excepción si el applet se registra más de una vez en el mismo evento. El parámetro `event` indica el valor del nuevo evento a registrar (entre -128 y 127). El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `EVENT_NOT_SUPPORTED` (si no se soporta el evento) y `EVENT_ALREADY_REGISTERED` (si el evento ya se ha registrado, para eventos limitados como el control de llamadas), `EVENT_NOT_ALLOWED` (si el evento es:

EVENT_MENU_SELECTION, EVENT_MENU_SELECTION_HELP_REQUEST, EVENT_ TIMER_EXPIRATION y EVENT_STATUS_COMMAND).

`public void setEventList(byte[] eventList, short offset, short length)` : Ajusta una lista de eventos en la entrada del applet en el Toolkit Registry. En el caso de que se produzca cualquier excepción, el estado del registro se encuentra en estado indefinido. El applet toolkit tiene que incluir esta llamada en una transacción, si es necesario. El parámetro `eventList` es el buffer que contiene la lista (a partir de la posición `offset` y con una longitud de `length` bytes) de los nuevos eventos a registrar. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `EVENT_NOT_SUPPORTED` (si alguno de los eventos no están soportados) y `EVENT_ALREADY_REGISTERED` (si el evento ya se ha registrado, para eventos limitados como el control de llamadas), `EVENT_NOT_ALLOWED` (si la lista de eventos contiene eventos como: `EVENT_MENU_SELECTION`, `EVENT_MENU_SELECTION_HELP_REQUEST`, `EVENT_ TIMER_EXPIRATION` y `EVENT_STATUS_COMMAND`).

15.2.8 CLASE `VIEWHANDLER`

`public abstract class ViewHandler extends java.lang.Object` : La clase `ViewHandler` ofrece servicios básicos y métodos básicos para manejar una lista de Simple-TLV's, que se pueden encontrar en el campo de datos de TERMINAL RESPONSE o en un elemento BER-TLV (del campo de datos de ENVELOPEo un comando proactivo). El byte de la posición de un manejador es la etiqueta del primer TLV simple.

15.2.8.1 Métodos

`public byte compareValue(short valueOffset, byte[] compareBuffer, short compareOffset, short compareLength)` : Compara el último elemento TLV encontrado, con un buffer. El parámetro `valueOffset` indica la posición del primer byte del elemento TLV a comparar. `compareBuffer` es una referencia del buffer a comparar. `compareOffset` da la posición en el buffer de comparación. `compareLength` da el número de bytes a comparar. El método devuelve el resultado de la comparación: 0 si son idénticos, -1 si el primer byte a comparar de la lista de Simple-TLV's es menor que el de `compareBuffer` y 1 si el primer byte a comparar de la lista de Simple-TLV's es mayor que el de `compareBuffer`. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado), `UNAVAILABLE_ELEMENT` (en el caso en que no haya disponible un elemento TLV) y `OUT_OF_TLV_BOUNDARIES` (si el parámetro `valueOffset` es negativo ó `valueOffset+compareLength` es mayor que la longitud del TLV actual).

`public short copy(byte[] dstBuffer, short dstOffset, short dstLength)` : Copia la lista de Simple-TLV's (contenida en el manejador) al array de bytes de destino. El parámetro `dstBuffer` es la referencia al buffer de destino. Los `dstLength` bytes de datos empezarán a copiarse a partir de la posición `dstOffset` del buffer. El método devuelve `dstOffset+dstLength`. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `HANDLER_NOT_AVAILABLE`

(si el manejador está ocupado) y `OUT_OF_TLV_BOUNDARIES` (si el parámetro `dstLength` es mayor que la longitud de la lista de Simple-TLV's).

`public short copyValue(short valueOffset, byte[] dstBuffer, short dstOffset, short dstLength)` : Copia una parte del último elemento TLV que ha sido encontrado, en un buffer de destino El parámetro `valueOffset` indica la posición del primer byte del elemento TLV que se va a copiar. `dstBuffer` es la referencia al buffer de destino donde se copiarán los `dstLength` bytes de datos a partir de la posición dada por `dstOffset`. El método devuelve `dstOffset+dstLength`. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado), `UNAVAILABLE_ELEMENT` (en el caso en que no haya disponible un elemento TLV) y `OUT_OF_TLV_BOUNDARIES` (si el parámetro `valueOffset` es negativo ó `valueOffset+dstLength` es mayor que la longitud del TLV actual).

`public byte findAndCompareValue(byte tag, byte[] compareBuffer, short compareOffset)` : Busca la primera ocurrencia de un elemento TLV desde el principio de una lista de TLV's y compara su valor con un buffer. Si no se encuentra ningún elemento TLV, se lanzará una excepción con el código de causa `UNAVAILABLE_ELEMENT`. Si el método tiene éxito, entonces el TLV correspondiente se convierte en el actual, sino no se selecciona ningún TLV. Este método de búsqueda no depende de la bandera CR (Comprehension Required). El parámetro `tag` es la etiqueta del elemento TLV a buscar. `compareBuffer` es una referencia al buffer de comparación. `compareOffset` da la posición en el buffer de comparación. El método puede devolver lo siguiente: 0 si son idénticos, -1 si el primer byte a comparar en el elemento TLV es menor que el de `compareBuffer` y 1 si el primer byte a comparar en el elemento TLV es mayor que el de `compareBuffer`. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado) y `UNAVAILABLE_ELEMENT` (en el caso en que no haya disponible un elemento TLV).

`public byte findAndCompareValue(byte tag, byte occurence, short valueOffset, byte[] compareBuffer, short compareOffset, short compareLength)` : Busca la ocurrencia de un elemento TLV desde el principio de una lista TLV y compara su valor con un buffer. Si no se encuentra ningún elemento TLV, se lanzará una excepción con el código de causa `UNAVAILABLE_ELEMENT`. Si el método tiene éxito, entonces el TLV correspondiente se convierte en el actual, sino no se selecciona ningún TLV. Este método de búsqueda no depende de la bandera CR (Comprehension Required). El parámetro `tag` es la etiqueta del elemento TLV a buscar. `valueOffset` da la posición del primer byte del elemento TLV de origen. `compareBuffer` es una referencia al buffer de comparación. `compareOffset` da la posición en el buffer de comparación. `compareLength` es la longitud de los datos que se van a comparar. El método puede devolver lo siguiente: 0 si son idénticos, -1 si el primer byte a comparar en el elemento TLV es menor que el de `compareBuffer` y 1 si el primer byte a comparar en el elemento TLV es mayor que el de `compareBuffer`. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado), `UNAVAILABLE_ELEMENT` (en el caso en que no haya disponible un elemento TLV), `OUT_OF_TLV_BOUNDARIES` (si el parámetro `valueOffset` es negativo ó

`valueOffset+compareLength` es mayor que la longitud del TLV actual) y `BAD_INPUT_PARAMETER` (si un parámetro de entrada no es válido, por ejemplo `occurence`=0).

`public short findAndCopyValue(byte tag, byte[] dstBuffer, short dstOffset)` : Busca la primera ocurrencia de un elemento TLV a partir del comienzo de la lista de TLV's y copia su valor en un buffer de destino. Si no se encuentra ningún elemento TLV, se lanza una excepción con el código de causa `UNAVAILABLE_ELEMENT`. Si el método tiene éxito, entonces el TLV correspondiente se convierte en el actual, sino no se selecciona ningún TLV. Este método de búsqueda no depende de la bandera CR (Comprehension Required). El parámetro `tag` indica la etiqueta del elemento TLV a buscar. `dstBuffer` es una referencia al buffer de destino. `dstOffset` indica la posición en el buffer de destino a partir de la cual se empezarán a copiar los datos. El método devuelve `dstOffset + la longitud de los datos copiados`. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado) y `UNAVAILABLE_ELEMENT` (en el caso en que no haya disponible un elemento TLV).

`public short findAndCopyValue(byte tag, byte occurence, short valueOffset, byte[] dstBuffer, short dstOffset, short dstLength)` : Busca la ocurrencia de un elemento TLV a partir del comienzo de la lista de TLV's y copia su valor en un buffer de destino. Si no se encuentra ningún elemento TLV, se lanza una excepción con el código de causa `UNAVAILABLE_ELEMENT`. Si el método tiene éxito, entonces el TLV correspondiente se convierte en el actual, sino no se selecciona ningún TLV. Este método de búsqueda no depende de la bandera CR (Comprehension Required). El parámetro `tag` es la etiqueta del elemento TLV a buscar. `valueOffset` es la posición del primer byte del elemento TLV. `dstBuffer` es la referencia al buffer de destino. `dstOffset` da la posición en el buffer de destino, a partir de la cual se empezarán a copiar los datos. `dstLength` da la longitud de los datos a copiar. El método devuelve `dstOffset+dstLength`. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado), `UNAVAILABLE_ELEMENT` (en el caso en que no haya disponible un elemento TLV), `OUT_OF_TLV_BOUNDARIES` (si el parámetro `valueOffset` es negativo ó `valueOffset+dstLength` es mayor que la longitud del TLV actual) y `BAD_INPUT_PARAMETER` (si un parámetro de entrada no es válido, por ejemplo `occurence`=0).

`public byte findTLV(byte tag, byte occurrence)` : Busca la ocurrencia indicada de un elemento TLV a partir del comienzo de la lista de TLV's (que se encuentra en el buffer del manejador). Si el método tiene éxito, entonces el TLV correspondiente se convierte en el actual, sino no se selecciona ningún TLV. Este método de búsqueda no depende de la bandera CR (Comprehension Required). El parámetro `tag` indica la etiqueta del elemento TLV a buscar. `occurrence` da el número de ocurrencias del elemento TLV (1 para la primera, 2 para la segunda...). El método puede devolver lo siguiente: `TLV_NOT_FOUND` (si la ocurrencia requerida del elemento TLV no existe), `TLV_FOUND_CR_SET` (si la ocurrencia requerida existe y la bandera Comprehension Required flag está ajustada) y `TLV_FOUND_CR_NOT_SET` (si la ocurrencia existe y la bandera Comprehension Required no está ajustada). El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa:

`HANDLER_NOT_AVAILABLE` (si el manejador está ocupado), `UNAVAILABLE_ELEMENT` (en el caso en que no haya disponible un elemento TLV) y `BAD_INPUT_PARAMETER` (si un parámetro de entrada no es válido, por ejemplo `occurence`=0).

`public short getLength()` : Devuelve la longitud de la lista de TLV's. El método devuelve la longitud en bytes. . El método podrá lanzar una `ToolkitException` con el código de causa `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado).

`public byte getValueByte(short valueOffset)` : Consigue un byte del último elemento TLV que se ha encontrado en el manejador. El parámetro `valueOffset` indica la posición del byte (que devuelve el método) en el elemento TLV. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado), `UNAVAILABLE_ELEMENT` (en el caso en que no haya disponible un elemento TLV) y `OUT_OF_TLV_BOUNDARIES` (si el parámetro `valueOffset` está fuera del elemento TLV actual).

`public short getValueLength()` : Consigue la longitud binaria del campo de valor del último elemento TLV que se ha encontrado en el manejador. El método devuelve la longitud del campo de valor. El método podrá lanzar una `ToolkitException` con los siguientes códigos de causa: `HANDLER_NOT_AVAILABLE` (si el manejador está ocupado) y `UNAVAILABLE_ELEMENT` (en el caso en que no haya disponible un elemento TLV).

15.3 EXCEPCIONES

15.3.1 EXCEPCIÓN `TOOLKITEXCEPTION`

`public class ToolkitException extends javacard.framework .CardRuntimeException` : Esta clase extiende la clase `Throwable` y permite que las clases de este paquete puedan lanzar excepciones específicas en el caso de que se produzcan problemas.

15.3.1.1 Campos

`static short ALLOWED_LENGTH_EXCEEDED` : Este código de causa (=10) se usa para indicar que la cadena (de texto) de la entrada al menú provista, es mayor que el espacio asignado.

`static short BAD_INPUT_PARAMETER` : Este código de causa (=14) se usa para indicar que un parámetro de entrada del método no es válido.

`static short EVENT_ALREADY_REGISTERED` : Este código de causa (= 7) se usa para indicar que el número máximo de applets registrados para este evento, ya se ha alcanzado (por ejemplo, el control de llamadas).

`static short EVENT_NOT_ALLOWED` : Este código de causa (=13) se usa para indicar que el método llamado no puede cambiar el registro del evento indicado.

`static short EVENT_NOT_SUPPORTED` : Este código de causa (= 6) se usa para indicar que el código del evento no está soportado por el Toolkit Framework.

`static short HANDLER_NOT_AVAILABLE` : Este código de causa (= 2) se usa para indicar que el manejador no está disponible (por ejemplo ocupado).

`static short HANDLER_OVERFLOW` : Este código de causa (= 1) se usa para indicar que la cantidad de datos son mayores que el espacio disponible en el manejador.

`static short INVALID_TIMER_ID` : Este código de causa (=12) se usa para indicar que el identificador de contador indicado, no está asignado a este applet.

`static short ME_PROFILE_NOT_AVAILABLE` : Este código de causa (= 9) se usa para indicar que los datos del perfil del terminal no están disponibles.

`static short MENU_ENTRY_NOT_FOUND` : Este código de causa (= 4) se usa para indicar que la entrada al menú solicitada no está definida para el applet correspondiente.

`static short NO_TIMER_AVAILABLE` : Este código de causa (=11) se usa para indicar que los contadores disponibles o el número máximo de contadores ya han sido asignados al applet.

`static short OUT_OF_TLV_BOUNDARIES` : Este código de causa (= 8) se usa para indicar que el offset, la longitud o ambos, están fuera de los límites del TLV actual.

`static short REGISTRY_ERROR` : Este código de causa (= 5) se usa para indicar un error en el registro del applet.

`static short UNAVAILABLE_ELEMENT` : Este código de causa (= 3) se usa para indicar que el elemento no está disponible en el buffer del manejador.

15.3.1.2 Constructores

`public ToolkitException(short reason)` : Construye una instancia de `ToolkitException` con la causa especificada en el parámetro `reason`. Para ahorrar recursos, use el método `throwIt()` para reutilizar la instancia de esta clase perteneciente al JCRE.

15.3.1.3 Métodos

`public static void throwIt(short reason)` : Lanza la instancia de la clase `ToolkitException` perteneciente al JCRE con la causa especificada en el parámetro `reason`.

16 Bibliografía y Referencias

Zhiqun Chen. *Java Card Technology for Smart Cards: architecture and programmer's guide*. Addison-Wesley, 2000.

Jose María Hernando Rábanos y Rafael Ayuso. *Comunicaciones móviles GSM*. Fundación Airtel, 1999.

GSM and UMTS: The creation of Global Mobile Communication. Editado por Friedhelm Hillebrand. John Wiley, 2002.

Alexander Joseph Huber y Josef Franz Huber. *UMTS and Mobile Computing*. Artech House, 2002.

José María Hernando Rábanos y Cayetano Lluch Mesquida. *Comunicaciones móviles de tercera generación, volumen 2*. Telefónica Móviles España, 2000.

Ed Ort. *Developing a Java Card Applet*. Sun Microsystems, `http://wireless.java.sun.com/javacard/articles/applet/`.

Rinaldo Di Giorgio. *Smart cards: A primer*. Java World, `http://www.javaworld.com/javaworld/jw-12-1997/jw-12-javadev_p.html`.

Zhiqun Chen y Rinaldo Di Giorgio. *Understanding Java Card 2.0. Learn the inner workings of the Java Card architecture, API, and runtime environment*. Java World, `http://www.javaworld.com/javaworld/jw-03-1998/jw-03-javadev_p.html`.

Jose María Hernando Rábanos. *Nuevos servicios y red UTMS*. Cátedra Amena, `http://catedra-amena.grpss.ssr.upm.es/`.

Descargas de las versiones de Java Card Kit, `http://java.sun.com/products/javacard/`.

Descargas de los entornos de desarrollo de Java, `http://java.sun.com/j2se/`.

Aplicación Sm@rtCafé. Giesecke & Devrient, `http://www.gi-de.com/eng/main/special/index.php4?special_id=86`.

Aplicación AspectsDeveloper. Aspects Software, `http://www.aspectssoftware.com/devtools/index.html`.

Javacomm. Sun Microsystems, `http://java.sun.com/products/ javacomm`.

Smart Card Applications. Java Card Special Interest Group, `http://www.javacard.org/others/sc_applications.htm`.

SIM Browsing, Universidad de Cantabria, `http://193.144.208.54/OTTIUC/es/Amena.html`.

Especificaciones de la ETSI y 3GPP:

- GSM 02.19: Subscriber Identity Module Application Programming Interface (SIM API).
- GSM 03.19 ó TS 43.019: Subscriber Identity Module Application Programming Interface (SIM API) for Java Card.
- GSM 03.48 ó TS 23.048: Security Mechanisms for the (U)SIM Application Toolkit.
- GSM 11.11: Specification of the Subscriber Identity Module - Mobile Equipment (SIM - ME) interface.
- GSM 11.13: Test specification for SIM API for Java Card.
- GSM 11.14: Specification of the SIM Application Toolkit for the Subscriber Identity Module - Mobile Equipment (SIM - ME) interface.
- TS 23038: Alphabets and language-specific information.
- TS 23.040: Technical realization of the Short Message Service.
- TS 31.101: UICC-terminal interface; Physical and logical characteristics.
- TS 31.102: Characteristics of the USIM application.
- TS 31.111: USIM Application Toolkit (USAT).
- TS 102.221: UICC-Terminal interface.
- TS 102.222: Integrated Circuit Cards (ICC); Administrative Commands for telecommunications applications.
- TS 102.223: Card Application Toolkit (CAT).
- TS 102.241: UICC Application Programming Interface (UICC API).

que se pueden descargar en las direcciones `http://www.3gpp.org/ftp /Specs` y `http://webapp.etsi.org/key/key.asp?full_list=y`.

A. F. Gómez, G. Martínez, J. García, J. F. Hidalgo, T. Jiménez. *EcoMobile Sistema instantáneo de acomodación de desplazamientos sobre teléfonos móviles mediante Java Card y SIM Application Toolkit*. Universidad de Murcia, `http://www.rediris.es/rediris/boletin/54-55/ponencia9.html`.

What is SIM Toolkit?, `http://www.cellular.co.za/sim_toolkit.htm`.

Technical Zone, `http://www.mobilesimtoolkit.com/`.

SIM Alliance, `http://www.simalliance.org/simalliance /default.asp`. Regístrese gratuitamente y podrá acceder a la zona de descargas. En dicha zona podrá encontrar presentaciones explicativas, ejemplos, aplicaciones y especificaciones.

Smart cards, Java Card. CIRCuS Work, `http://ants.dif.um.es/circus/work/smartcards/`.

G. Richter. *A study of the SIM Application Toolkit*. University of Pretoria.

Páginas web con enlaces a otras:

- Enlaces a herramientas que se usan en el mundo Java, http://wireless.java.sun.com/allsoftware/.
- Links relative to Smart Cards and Electronic Payment Systems, http://www.dice.ucl.ac.be/crypto/card.html.
- Prasad's Electronic Commerce & Smart Cards Page, http://home.att.net/~s-prasad/ecsc.htm.
- Welcome to Links2Mobile.com, http://www.links2mobile.com/.