OTA Manager V5.1

Developing Client Applications

gemalto
security to be free

GEMALTO, B.P. 100, 13881 GEMENOS CEDEX, FRANCE.

Tel: +33 (0)4.42.36.50.00 Fax: +33 (0)4.42.36.50.90

Printed in France.

Document Reference: DOC117205C

Product version: 5.1.0

November 20, 2009

Contents

Preface vii

Additional Product Modules vii
For More Information vii
Contact Our Hotline viii

Chapter 1 Using the Core API 1

Compiling Client Applications 1
Compiling the Sample Applications 2
Running Client Applications 2
Running the Sample Client Applications 3
Shutting Down the Core API 4

Chapter 2 Connecting to the Platform 5

PlatformCheckPoint Objects 5
Identifying a User 6
Authenticating a User 6
The PlatformAccessPoint View 7

Chapter 3 Common Administration Modules 9

User Account Management 9
- Subscriber Account Management 11
  - Accessing a Subscriber Account 11
  - Creating a Subscriber Account 12
  - Updating a Subscriber Account 12
  - Deleting a Subscriber Account 13
  - Getting the Number of Subscribers for each Subscriber Profile 14
- Account Management 14
  - Accessing an Account 15
  - Creating an Account 15
  - Updating an Account 15
  - Deleting an Account 16

User Profile Management 17
- Accessing a Profile 19
- Creating a Profile 20
- Updating a Profile 20
- Managing Profile Access Rights 21
  - Checking whether a Profile Has a Particular Access Right 21
  - Granting or Revoking an Access Right To a Profile 21
  - Setting a Quality of Service On a Profile 21
  - Setting Allowed Services and Allowed Protocol for Service Execution 23
- Deleting a Profile 24

System Facilities 24
- Product Life Cycle Facility Management 25
  - Starting a Product 26
- System Configuration Facility Management 27
  - Retrieving Product Parameter Values 28
  - Setting a Product Parameter Value 29

Exporting and Importing a Product Parameters File 29
System Time Facility Management 30
Getting the Date 30
Logging Facility Management 31
Retrieving Logging Trace Domains 32
Setting the Logging Trace Level 33
Billing Facility Management 34
Activating and Deactivating Billing 34
Audit Trail Facility Management 35
SNMP Facility Management 36
Activating the SNMP Agent 36
Getting an SNMP Object Value 37
Events 38
Card Events 40
Subscribing to any MO Event 40
Subscribing to any Binary MO Event 40
Receiving Events 41
Unsubscribing an Event 42
CRPC Events 42
Invocation Events 42
Subscribing To Any Invocation Event 43
Subscribing To Any Failed Service Execution Event 43
Unsubscribing to Invocation Events by Removing the Listening Chain 44
Implementing an Invocation Filter 44

Chapter 4 Business APIs 47

Access 47
Invoking a Service 47
Step One: Building the Invocation Chain 48
Card 48
Card Holders 48
Bearer 49
Step Two: Build Invocation Parameters 50
Step Three: Request Service Execution 52
SmartCard 53
SmartCard.post() 53
SmartCard.apply() 53
MobileEquipment 54
MobileEquipment.postSendText() 54
MobileEquipment.sendText() 54
SIMCard, SIMCardViewer 55
Application Interface and Sub-Interfaces 55
ServiceParameter 55
GenericServiceParameter 56
Service Result 60
Invocation 61
Code Examples 62
Performing a Service Invocation (ActivateADN) in Synchronous Mode 62
Performing a Service Invocation (ActivateADN) in Asynchronous Mode . . . 62
Performing a Service Invocation (UpdateADN) in Synchronous Mode 63
Setting Invocation Parameters and SMSC Options 63
Performing a CAT-TP Service Invocation (Activate ADN) in Synchronous Mode 65
Performing the sendText Service on the Mobile 66
Performing a Service Invocation Using Generic Call 66

Monitoring a Request 67
Searching for Invocations 67
Obtaining Invocation Processing Details 69
Acknowledging an Invocation 69
Provisioning and Data Management 70
Viewer 71
Manager 71
Loader 71
Code Examples 72
Creating a Card Definition 72
Updating a Card Definition 73
Deleting a Card Definition 73
Batchloading a Card Definition 74
SIM Card Provisioning 74
Creating a SIM Card 75
Activating an Existing SIM Card 76
Deleting a SIM Card 76
Card Content Representation and Access 77
File System Representation 78
Java Content Representation 80
Card Security Information 82
Searching for Card Security Information 83
Creating a New Synchronization Counter In An Existing Security Bundle 84
Updating a Synchronization Counter In An Existing Security Bundle 85
Deleting a Synchronization Counter in an Existing Security Bundle 86
The Delegated Management Facility 86

Chapter 5 Invocation Submission Flow Control 89

Busy Status Exception Handling 89
Determining when the Platform Lifts Busy Status 89

Chapter 6 Runtime Exceptions 91

Terminology 93

Abbreviations 93
Glossary 94

List of Figures

Figure 1 - Connecting to the Platform 5
Figure 2 - The PlatformAccess View 7
Figure 3 - Managing User Accounts 9
Figure 4 - PlatformSubscriberAccountManager interface 11
Figure 5 - The PlatformAccountManager Interface 14
Figure 6 - Profile Management 17
Figure 7 - The ProfileManager Interface 19
Figure 8 - PlatformSystemFacility 25
Figure 9 - ProductLifeCycleManager 25
Figure 10 - The Product Life Cycle 26
Figure 11 - The SystemConfigManager Interface 27
Figure 12 - The SystemTimeManager Interface 30
Figure 13 - The Logging Facility 31
Figure 14 - Billing Facility Management 34
Figure 15 - The AuditTrailManager Interface 35
Figure 16 - The Audit Trail Facility 35
Figure 17 - SNMP Facility Management 36
Figure 18 - Events 38
Figure 19 - Event Management 39
Figure 20 - Invocation Events 42
Figure 21 - Managing Card Holders and CAT-TP Objects 49
Figure 22 - The Bearer Interface 50
Figure 23 - The InvocationFacility Interface 51
Figure 24 - Building the InvocationParameter 51
Figure 25 - The InvocationParameter Interface 52
Figure 26 - The SmartCard Interface 53
Figure 27 - The Deprecated SIMCard Interface 55
Figure 28 - The ServiceParameter Interface 56
Figure 29 - The GenericServiceParameter Interface 56
Figure 30 - The Service Result 61
Figure 31 - The Invocation Interface 61
Figure 32 - Monitoring a Request 67
Figure 33 - Data Management 71
Figure 34 - The SmartCardImage Interface 77
Figure 35 - File System Representation 78
Figure 36 - File Interpreters 80
Figure 37 - Java Application Representation 81
Figure 38 - Card Security Information 82
Figure 39 - The Delegated Management Facility Interface 87

Gemalto's LinqUs OTA Manager V5.1 is part of LinqUs Over-The-Air Suite, a fully-integrated software solution providing you with a robust, reliable and scalable infrastructure with which to manage your (U)SIM installed base. LinqUs Over-The-Air Suite allows you to:

- Deliver new applications directly to your subscribers
- Easily adapt your service portal for diverse subscriber types
- Control your subscribers' mobile environment.

This book is for developers intending to develop new products to be integrated into or connect to the OTA Manager V5.1 platform.

Additional Product Modules

The following additional modules are not delivered with the standard product, but can be ordered separately for OTA Manager V5.1:

- Xxpress Campaign Technology™ (XCT™) campaign manager
- Interoperable Remote File Management (RFM) services for 3G Java cards
- OTA over IP CAT-TP module
- Wired Internet Reader (WIR) for point of sale use

For More Information

For more information on OTA Manager V5.1, see the following documents:

Document	Description
Third-Party Software Installation Guide	Describes how to install third-party software components, including Oracle 10g and Weblogic.
Third-Party Software Prerequisites	Lists the versions and licensing requirements of all third-party software packages that are necessary to install OTA Manager.
Installation Guide	Describes the procedure for installing the OTA Manager platform and channel drivers.
Administration Guide	Describes how to perform management tasks for the OTA Manager platform.
Getting Started	Provides an overview of OTA Manager functions, helps you understand the product and what you can do with it, and provides step-by-step tasks for using the product.
Customization Guide	Describes how to program a new service and add it to OTA Manager, and how to internationalize the interface.
Developing Client Applications	Gives information on expanding OTA Manager functionality by developing new products.
Script Programming Guide	Provides detailed information on how to write scenario scripts to perform remote file management tasks using the ExecScript service of OTA Manager.

Document	Description
Provisioning Guide	Provides the essentials for provisioning OTA Manager.
Online Help	Task Help (accessible from the left-hand menu bar) explains how to perform OTA Manager tasks using the graphical user interface. Field Help provides a contextual description of the fields on each window.
Master Index	A compilation of all OTA Manager index entries which enables you to find information in OTA Manager manuals quickly.

For information on managing applets, refer to the LinqUs Application Repository Manager (ARM) documentation.

Contact Our Hotline

If you do not find the information you need in this manual, or if you find errors, contact the Gemalto hotline at http://support.gemalto.com/.

Please note the document reference number, your job function, and the name of your company. (You will find the document reference number at the bottom of the legal notice on the inside front cover.)

1

Using the Core API

The Core API is a Java-based application programming interface (API) that provides a client application with a single point of access to the Framework.

The Core API is an “active” (that is, containing threads) client/server component that allows you to transparently and remotely access all the features of OTA Manager V5.1. The Core API provides platform user entry points (log in), administration of common modules (that is, the Framework administrative API), and also provides links to other products in the LinqUs Over-The-Air Suite product range with a business-specific API, also called the “Core API extensions”.

Although based on CORBA communications (using the VisiBroker ORB), the Core API user has NO visibility of this communications protocol and only sees a set of Java classes. Technically, the Core API is dependent on the Framework version, in particular the VisiBroker version used in the Framework.

OTA Manager V5.1 is built using JDK 1.5.0 and must function with the JDK 1.5.0. It uses VisiBroker as the remote communications layer. It is recommended that Core API client applications run under the same JDK version as that used for the OTA Manager server part, and use the appropriate files:

- `vbjorb.jar`
- `lm.jar`
- `vbsec.jar`
- `sanct6.jar`
- `sanctuary.jar` for VisiBroker 8.0 SP1.

Compiling Client Applications

The CLASSPATH to be used by a Framework client for compiling a Core API client only needs to reference the Framework JAR, `basesystem.jar`, and a common utility JAR, `common.jar`.

The following command can be used to compile any of the Core API client code examples described in this manual:

```
javac -classpath basesystem.jar; common.jar;
     [<CoreAPI_extention_JARs>;]
     [<Rest_of_the_classpath>]
<Class_to_compile>
```

Compiling the Sample Applications

Use the following command line to compile the sample Core API applications:

```
javac -verbose -g
      -d $SAMPLES_TARGET
     -classpath $JUNIT381\junit.jar;$PLATFORM_HOME\FRWK\lib\
reuse.jar;$PLATFORM_HOME\FRWK\lib\common.jar;$PLATFORM_HOME\FRWK\lib\
basesystem.jar $SAMPLES_SRC\*.java
```

Where:

- `$SAMPLES_TARGET` is the requested output location
- `$JUNIT381` is the path to the JUnit libraries
- `$PLATFORM_HOME` is the root installation directory of the platform.

Running Client Applications

To run Core API client applications, the following JAR files must be added to `basesystem.jar` and `common.jar`:

- The VisiBroker JAR. VisiBroker Version 8.0 SP1 is used for the OTA Manager V5.1 Framework. The `vbjorb.jar`, `lm.jar`, `vbesc.jar`, `sanct6.jar and sanctuary.jar` files must be included in the CLASSPATH. The BES_LIC_DEFAULT_DIR and BES_LIC_DIR environment variables must be set to `$VBROKER_HOME/license`.
- `reuse.jar`, containing reusable components.
- `container.jar`, containing low-level communication layers.
- `Rca-compat.jar` (only if compatibility with the deprecated API for platform connections is required).

Moreover, to be able to carry out the initialization of the Core API, and more generally for it to function correctly, certain Java system properties must be specified to the Core API. The set of properties to be used for the Framework Core API is:

- **vbroker.agent.port**. An integer specifying the port number of the VisiBroker OSAGENT. This is mandatory information that has no default value.
- **vbroker.agent.addrFile**. A string specifying the name of the file containing a list of the addresses (host name or IP address) of all machines hosting VisiBroker OSAGENTs. The file is automatically generated by the OSAGENT wrapper script. The fixed value of this property is `$PLATFORM_HOME/FRWK/bin/osagent.agentaddr.file`. This information is required because each OSAGENT starts in a restrictive “local” mode, meaning that the previous broadcast discovery technique no longer works.

In a complex multi-host environment using virtual IP addresses and multiple OSAGENT processes, a number of additional properties must be passed to the Java client products to control their behaviour:

- **-Dcontainer.poll.intervall=30**. The interval in seconds between two scans of all available CORBA servers, in order to store the information locally.
- **-Dvbroker.agent.keepAliveTimeout=30**. The period in seconds after which the associated OSAGENT is discarded, and a new one taken from the OSAGENTs list
- **-Dvbroker.ce.iiop.host=***product_host*. Use the *product_host* IP address when initiating client CORBA calls instead of the default host name (required when the product is associated with a virtual IP address).

- **-Dvbroker.se.iiop_tp.host=***product_host*. Binds the CORBA server listening socket to the *product_host* IP address (recommended when the product is associated with a virtual IP address).

Finally, when the client product accesses one or more database instances, it is required to identify the location of the `tnsnames.ora` file, which by default is not stored in the default location:

- **-Doracle.net.tns_admin=***product_home*`/SOLUTION/config`. Name of the directory containing the `tnsnames.ora` file.

If the Card Manager is deployed and its use requested by the client application, the following Java system properties information must be specified to the Core API:

- **gxs.product.name**: A string specifying the Card Manager name as declared in the hosted products list. This is mandatory information that has no default value.
- **gxs.invocationmanager.name**: A string specifying the Card Manager name as declared in the hosted products list. This is mandatory information that has no default value.

The following command line therefore allows you to launch any of the Core API code examples described in this manual:

```
java
   -classpath
     basesystem.jar;
     common.jar;
     reuse.jar;
     [<CoreAPI_extention_JARs>;]
     vbjorb.jar;lm.jar;vbsec.jar;
     Tools.jar;TopLink.jar;TopLinkX.jar;
     [<Remainder_of_the_classpath>]
   -Dvbroker.agent.port=<OSAgent_port>
   -Dvbroker.agent.addrFile=$PLATFORM_HOME/FRWK/bin/osagentaddr.file
   -Dgxs.product.name =<CardManager name>
   -Dgxs.invocationmanager.name =<CardManager name>
<Main_to_execute>
```

Running the Sample Client Applications

Use the following command line to execute the sample client applications provided:

```
java
   -Dtestproperties.mode=ARGS
   -Dvbroker.agent.port=value
   -Dvbroker.agent.addrFile=$PLATFORM_HOME/FRWK/bin/
osagent.agentaddr.file
   -Dgxs.invocationmanager.name=value
   -Dgxs.product.name=value
   -Ddefault_user.name=value
   -Ddefault_user.pwd=value
   -classpath $SAMPLE_TARGET;$PLATFORM_HOME\FRWK\lib\
reuse.jar;$PLATFORM_HOME\FRWK\lib\common.jar;$PLATFORM_HOME\FRWK\lib\
basesystem.jar;$PLATFORM_HOME\lib\ext\xercesImpl.jar;$PLATFORM_HOME\
lib\ext\xmlParserAPIs.jar;$VBROKER\lm.jar;$VBROKER\
vbjorb.jar;$VBROKER\vbsec.jar;$VBROKER\sanct4.jar;$VBROKER\
sanctuary.jar;$TOPLINK\Tools.jar;$TOPLINK\TOPLink.jar;$TOPLINK\
TOPLinkX.jar;$JUNIT381\junit.jar;
org.apache.tools.ant.taskdefs.optional.junit.JUnitTestRunner
```

```
gcota.example.AllTests
filtertrace=false
haltOnFailure=false
showoutput=true
formatter=org.apache.tools.ant.taskdefs.optional.junit.
SummaryJUnitResultFormatter
formatter=org.apache.tools.ant.
taskdefs.optional.junit.XMLJUnitResultFormatter,d:\tmp\
gcotaexample.xml
```

where:

- `$SAMPLE_TARGET` is the location of the compiled sample application
- `$JUNIT381` is the path to the Junit libraries
- `$PLATFORM_HOME` is the root installation directory of the platform
- `$XERCES` is the path to Xerces libraries
- `$LOG4J` is the path to Log4j libraries
- `$VBROKER` is the path to VisiBroker libraries
- `$TOPLINK` is the path to Toplink libraries
- `$ANT` is the path to the Ant libraries
- The `vbroker.agent.port` property *value* specifies the ORB agent port of an installed platform, for example, “16831”
- The `vbroker.agent.addrFile` property specifies the path of the `osagent.agentaddr.file` file.
- The `gxs.invocationmanager.name` property *value* specifies the Card Manager product name of an installed platform, for example, “RCA”
- The `gxs.product.name` property *value* specifies the Card Manager product name of an installed platform, for example “RCA”
- The `default_user.name` property *value* specifies the user account from which to run the samples, for example “admgemalto”
- The `default_user.pwd` property *value* specifies the corresponding user account password, for example, “gemalto40”

Shutting Down the Core API

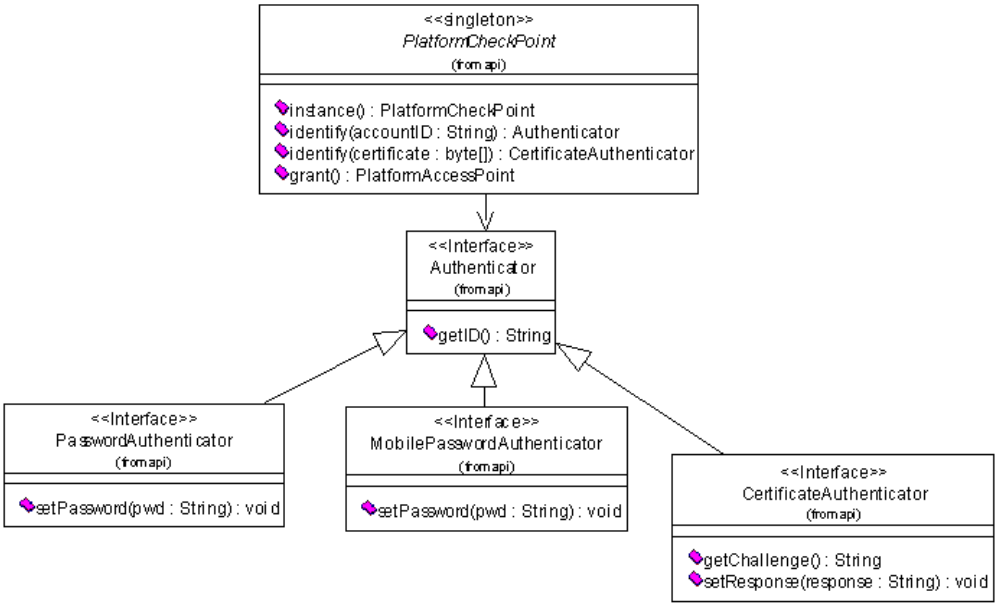
It is not possible to correctly control the stopping of the “active” elements of the Core API at the end of a Core API client applications’s main procedure. Consequently, the Core API client must exit using the **System.exit()** method, otherwise a “normal” exit cannot be ensured.

2

Connecting to the Platform

PlatformCheckPoint Objects

Figure 1 - Connecting to the Platform



The **PlatformCheckPoint** class is the Core API entry point. It allows a platform user to be identified and authenticated.

Access control is performed as follows:

- Identification of the user using **PlatformCheckPoint.identify()**: the platform is passed the user identifier. The method returns an **Authenticator** object that represents the authentication mode associated with the user:
  - password, certificate (reserved for future use),
  - “Mobile Challenge” (the password is generated by the platform and sent to the user’s handset—in OTA Manager V5.1, this mode is reserved for subscribers).
- Authentication of the user: the identity of the user is verified. The necessary proofs of identity are set on the **Authenticator** object, then the **PlatformCheckPoint.grant()** method is called.

If the user’s identity is unknown, or authentication fails, a **DeniedAccessException** is thrown.

Once connected, the user is provided with an instance of the **PlatformAccessPoint** object that establishes the connection.

Identifying a User

The following example shows how to identify a user:

```
// How to identify (using the PlatformCheckPoint)?
PlatformCheckPoint ptCheckPoint;
Authenticator authenticator;

//Get the PlatformCheckPoint instance
ptCheckPoint = PlatformCheckPoint.instance();

try {
   //identifies using the default account
   authenticator = ptCheckPoint.identify("admgemalto");

   //Can verify that the used authentication method is with password
   if (!(authenticator instanceof PasswordAuthenticator)) {
      System.out.println(
         "Connection to platform KO, PasswordAuthenticator
            was expected instead of " + authenticator.getClass());
      throw new DeniedAccessException();
   }
} catch (DeniedAccessException e1) {
   System.out.println("Connection to platform KO, Access denied: "
      + e1.getMessage());
   throw e1;
} catch (ExpiredAccountException e2) {
   System.out.println("Connection to platform KO, Account expired");
   throw e2;
}
```

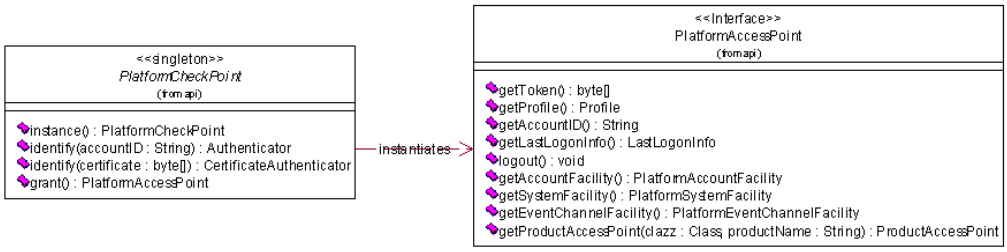
Authenticating a User

The **authenticator** object in the following example is the one obtained previously in “Identifying a User”.

```
// How to authenticate (using the PlatformCheckPoint)?
PlatformAccessPoint platform = null;

// Authentification phase (using the default account)
((PasswordAuthenticator) authenticator).setPassword("gemalto50");

try {
   platform = ptCheckPoint.grant(authenticator);
} catch (DeniedAccessException e1) {
   System.out.println("Connection to platform KO, Invalid password");
   System.exit(1);
} catch (TimeoutException e2) {
   System.out.println("Connection to platform KO, Timeout...");
   System.exit(1);
}
```

The PlatformAccessPoint View

Figure 2 - The PlatformAccess View



The **PlatformAccessPoint** object provides access to:

- User account management (**PlatformAccountFacility**)
- Global platform configuration (**PlatformSystemFacility**)
- Platform event management (**PlatformEventChannelFacility**)

The **PlatformAccessPoint** instance also retains information on the connected user: the user identifier, user profile, and last login.

In general, access control is dependent upon obtaining the corresponding manager when starting from **PlatformAccessPoint**. If access is not authorized, a **DeniedAccessException** is raised.

The facilities that are directly accessible relate to Framework administration, and are consequently independent of any particular business, in particular to card management.

The set of facilities applicable to a particular business is called the “Core API extension” of the product. The **PlatformAccessPoint** object enables access to be given to the Core API extension of all products that have implemented the Product Access Point concept. This is done using the **getProductAccessPoint** entry point.

The business to retrieve is specified by the relevant **ProductAccessPoint** class, and the name of the **Product** to connect to.

The following example shows how to retrieve the Card Manager access point, giving the targeted Card Manager product name.

```
//How to access the CardManagerAccessPoint?
CardManagerAccessPoint point = null;
try {
   //Use the product name declared
   point = (CardManagerAccessPoint) platform.getProductAccessPoint(
           CardManagerAccessPoint.class, productName);
} catch (ProductNotFoundException exp) {
   System.out.println("Unable to access 'CardManagerAccessPoint'");
   return;
}

System.out.println("CardManagerAccessPoint name" +
point.getProductName());
```

3

Common Administration Modules

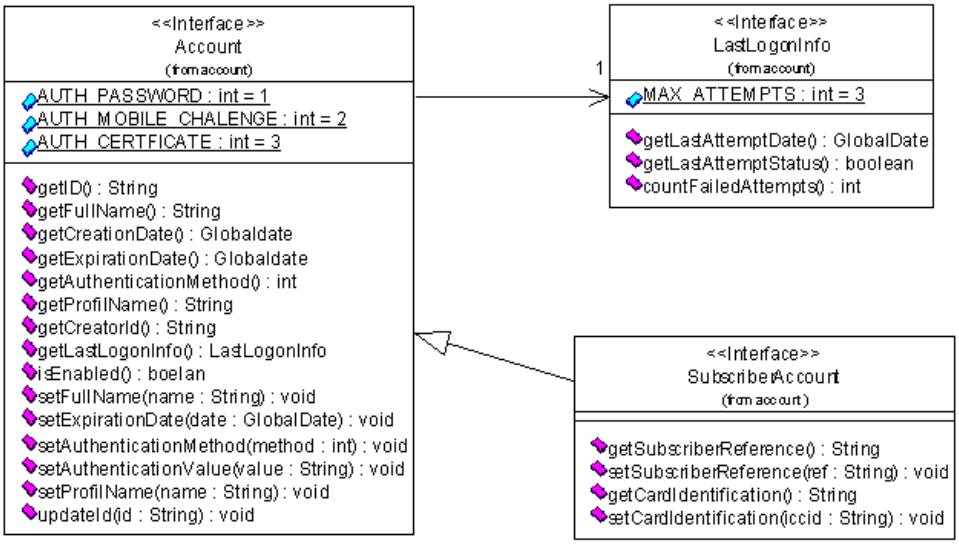
The API described in this chapter corresponds to what is termed the “administrative API” of the Framework.

The internal usage of the Framework facilities made within the hosted product’s source code is done through a different API, the “operational API”. This chapter only covers the administrative API.

User Account Management

“Figure 3” shows the interfaces available to perform user account management.

Figure 3 - Managing User Accounts



Accounts are represented by two interfaces:

- **Account** for all types of user accounts
- **SubscriberAccount**, specifically for accounts representing a subscriber.

An **Account** has:

- An identifier, accessible though the **getID** method. This identifier can subsequently be modified using the **updateId** method.

- Free text for specifying the full name of the individual user corresponding to the account, accessible though the **getFullName** method.
- The identifier of its creator and the creation date, automatically set at the account creation time and accessible though the **getCreatorId** and the **getCreationDate** methods.
- An account expiration date, after which the account is no longer valid. This information is accessible though the **getExpirationDate** method, and can be **null** to specify that the account never expires.
- An authentication method, accessible though the **getAuthenticationMethod** method. Valid values are the constants AUTH_PASSWORD, AUTH_MOBILE_CHALLENGE, and AUTH_CERTIFICATE.
- An authentication value, that is, data that is required in order to achieve authentication of the account, following the authentication method. For a “password” authentication, for example, the authentication value is the password value. The authentication value can be set (and modified) at any time, but cannot be accessed.
- The name of the profile that defines the access right of the account. The profile name is accessible though the **getProfileName** method.

SubscriberAccount provides additional information:

- A subscriber reference (free text to be used to make an external link with the operator’s subscriber database), accessible through the **getSubscriberReference** method.
- An “owned card” identifier (to establish a link with the Card Manager database), accessible through the **getCardIdentification** method.

Each account is associated with one and only one **Profile**.

Each account keeps information concerning the last connection attempt (the **getLastLogonInfo** method).

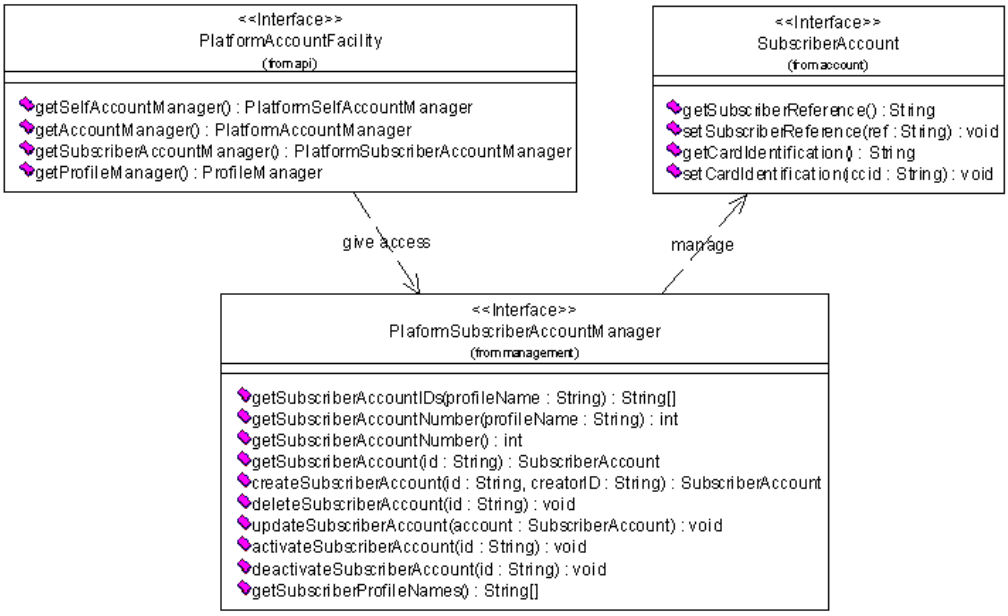
By default, when an account is created, it is:

- Inactive, requiring explicit activation
- In password authentication mode
- Without an expiry date
- Without a profile.

To be valid, therefore, an account must be associated with a profile and assigned a valid authentication mode.

Subscriber Account Management

Figure 4 - PlatformSubscriberAccountManager interface



The **PlatformSubscriberAccountManager** interface allows subscriber management. This functionality is available for administrators and customer care agents only.

It enables you to create, update, activate, deactivate, and delete subscriber accounts (using the **createSubscriberAccount**, **updateSubscriberAccount**, **activateSubscriberAccount**, **deactivateSubscriberAccount** and **deleteSubscriberAccount** methods, respectively), and also to retrieve accounts using either the subscriber identifier (**getSubscriberAccount** method), or a profile name to retrieve the identifiers of all the subscriber accounts associated with the profile (**getSubscriberAccountIDs** method).

The number of currently existing subscribers, either globally or associated with a particular profile, are accessible through the **getSubscriberAccountNumber()** method.

Accessing a Subscriber Account

The following example shows how to get access to a subscriber account:

```
// Standard Framework Subscriber Account Manager
PlatformSubscriberAccountManager manager = null;

try {
   manager =
platform.getAccountFacility().getSubscriberAccountManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

SubscriberAccount account =
manager.getSubscriberAccount("_SubscriberMsisdn_");
```

```
if (account==null) {
   System.out.println("Account not found");
} else {
   System.out.println("Account found: " + account.getID());
}
```

Creating a Subscriber Account

The following example shows how to create a subscriber account:

```
// Standard Framework Subscriber Account Manager
PlatformSubscriberAccountManager manager = null;

try {
   manager =
platform.getAccountFacility().getSubscriberAccountManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

try {
   manager.createSubscriberAccount("_SubscriberMsisdn_",
platform.getAccountID());
} catch (AlreadyExistsException e) {
   // The account may exist
   System.out.println(e.getMessage());
}
```

When created, the subscriber account is deactivated, as it is not valid until it is assigned to a valid profile. It must be updated and completed to be valid (refer to “Updating a Subscriber Account”).

Updating a Subscriber Account

The following example shows how to update a subscriber account:

```
// Standard Framework Subscriber Account Manager
PlatformSubscriberAccountManager manager = null;

try {
   manager =
platform.getAccountFacility().getSubscriberAccountManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

// Access to the account
SubscriberAccount account =
manager.getSubscriberAccount("_SubscriberMsisdn_");

account.setProfileName("_SubscriberProfileName_");
account.setFullName("Albert Dupont");
account.setSubscriberReference("A012");
```

```
account.setCardIdentification("123");
account.setAuthenticationValue("password00");

// Update the account
try {
   manager.updateSubscriberAccount(account);
} catch (NoMatchingObjectException e) {
   // If the account has not been previously declared
   System.out.println(e.getMessage());
   return;
} catch (IllegalArgumentExceptione) {
   // If, for example, the profile doesn't exist
   System.out.println(e.getMessage());
   return;
} catch (PlatformException e) {
   // If the connection between core api and card manager failed
   System.out.println(e.getMessage());
   return;
} catch (AlreadyExistsException e) {
   // If the card ID is already used with an other susbscriber
   System.out.println(e.getMessage());
   return;
} catch (InvalidStateException e) {
   // If the card manager is not active
   System.out.println(e.getMessage());
   return;
}

// Activate the account
try {
   manager.activateSubscriberAccount("_SubscriberMsisdn_");
} catch (NoMatchingObjectException e) {
   // If the account has not been previously declared
   System.out.println(e.getMessage());
}
```

Deleting a Subscriber Account

The following example shows how to delete a subscriber account:

```
// Standard Framework Subscriber Account Manager
PlatformSubscriberAccountManager manager = null;

try {
   manager =
platform.getAccountFacility().getSubscriberAccountManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

try {
   manager.deleteSubscriberAccount("_SubscriberMsisdn_");
} catch (NoMatchingObjectException e) {
   // If the account has not been previously declared
```

```
   System.out.println(e.getMessage());
   return;
}
```

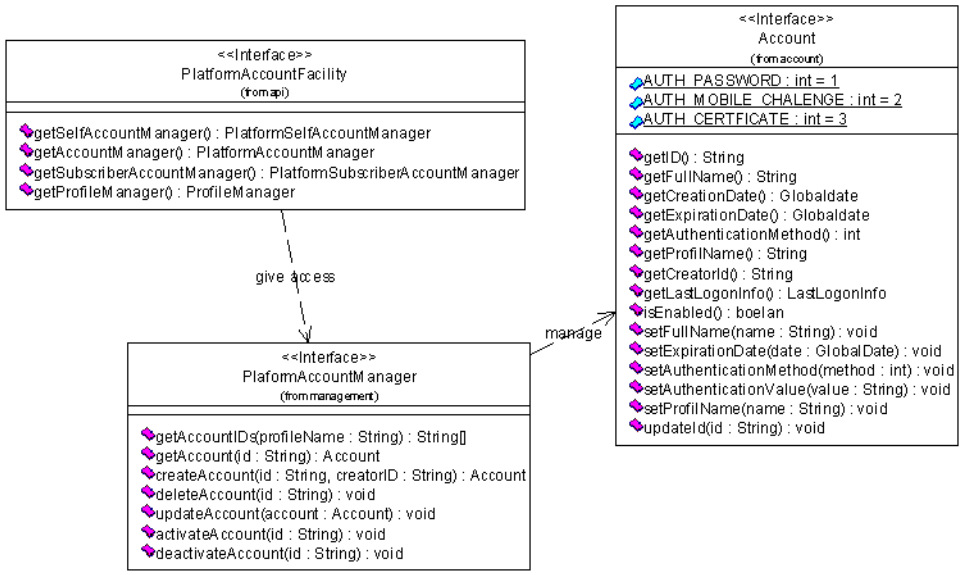
Getting the Number of Subscribers for each Subscriber Profile

The following example shows how to get the number of subscribers for each subscriber profile:

```
// Standard Framework Subscriber Account Manager
PlatformSubscriberAccountManager manager = null;

try {
   manager =
platform.getAccountFacility().getSubscriberAccountManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

String [] subscProfileNames = manager.getSubscriberProfileNames();

for (int i = 0; i < subscProfileNames.length; i++) {

System.out.println(manager.getSubscriberAccountNumber(subscProfileName
s[i])
      + " Subscriber accounts is/are linked with profile " +
subscProfileNames[i]);
}
```

Account Management

Figure 5 - The PlatformAccountManager Interface



The **PlatformAccountManager** interface allows you to manage administrator and customer care agent accounts. This functionality is only available to administrators.

It enables you to create, update, activate, deactivate, and delete accounts (**createAccount**, **updateAccount**, **activateAccount**, **deactivateAccount** and **deleteAccount** methods, respectively) and also to retrieve them using either their identifier (**getAccount** method), or a profile name to retrieve the identifiers of all accounts linked to the given profile (**getAccountIDs** method).

Accessing an Account

This is similar to accessing a subscriber account; refer to “Accessing a Subscriber Account” on page 11.

```
// Standard Framework Account Manager
PlatformAccountManager manager = null;

try {
   manager = platform.getAccountFacility().getAccountManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

Account account = manager.getAccount("_AccountName_");
if (account==null) {
   System.out.println("Account not found");
} else {
   System.out.println("Account found: " + account.getID());
}
```

Creating an Account

This is similar to creating a subscriber account; refer to “Creating a Subscriber Account” on page 12.

```
// Standard Framework Account Manager
PlatformAccountManager manager = null;

try {
   manager = platform.getAccountFacility().getAccountManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

try {
   manager.createAccount("_AccountName_", platform.getAccountID());
} catch (AlreadyExistsException e) {
   // The account may exist
   System.out.println(e.getMessage());
}
```

Updating an Account

This is similar to updating a subscriber account; refer to “Updating a Subscriber Account” on page 12.

```
// Standard Framework Account Manager
PlatformAccountManager manager = null;
// Account Manager
```

```
//AccountManager manager = null;

try {
   manager = platform.getAccountFacility().getAccountManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

// Access to the account
Account account = manager.getAccount("_AccountName_");

account.setProfileName("_ProfileName_");
account.setFullName("John Smith ");
account.setAuthenticationValue("password00");

// Update the account
try {
   manager.updateAccount(account);
} catch (NoMatchingObjectException e) {
   // If the account has not been previously declared
   System.out.println(e.getMessage());
   return;
} catch (PlatformException e) {
   // If the connection between core api and card manager failed
   System.out.println(e.getMessage());
   return;
}

// Activate the account
try {
   manager.activateAccount("_AccountName_");
} catch (NoMatchingObjectException e) {
   // If the account has not been previously declared
   System.out.println(e.getMessage());
}
```

Deleting an Account

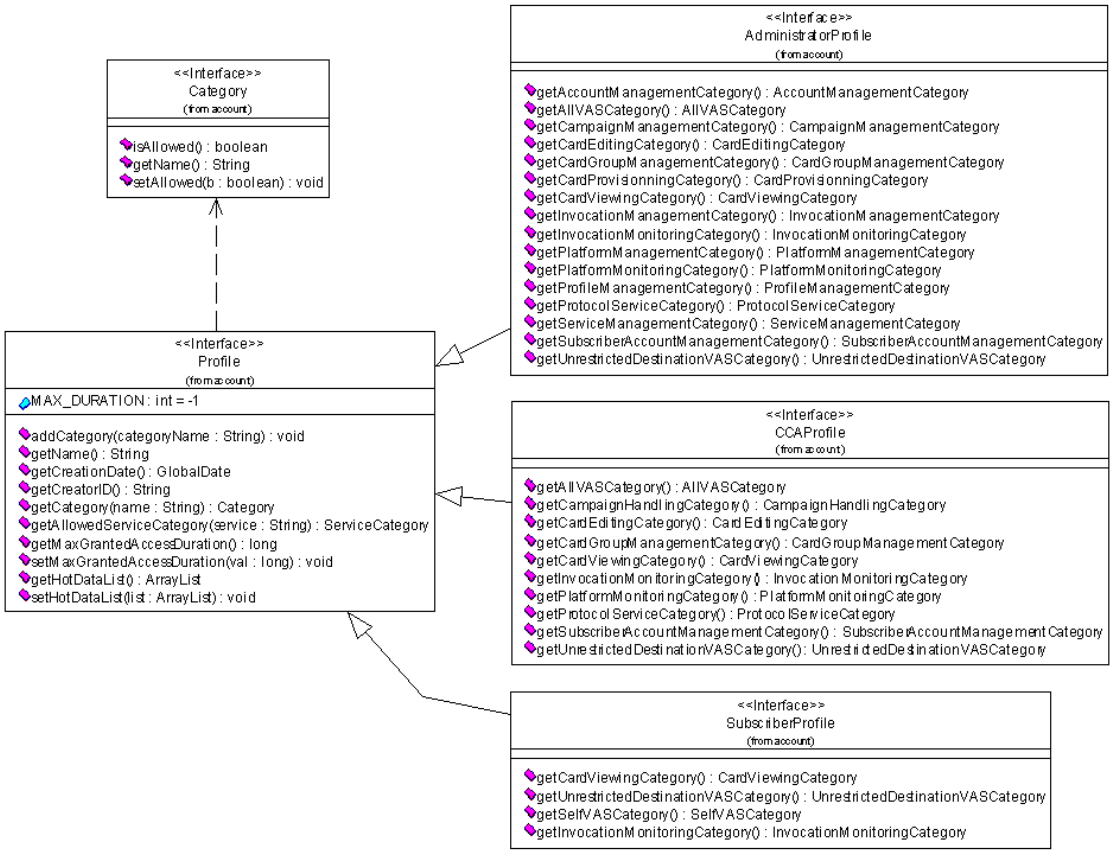
This is similar to deleting a subscriber account; refer to “Deleting a Subscriber Account” on page 13.

```
// Standard Framework Account Manager
PlatformAccountManager manager = null;

try {
   manager = platform.getAccountFacility().getAccountManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

try {
   manager.deleteAccount("_AccountName_");
} catch (NoMatchingObjectException e) {
```

```
        // If the account has not been previously declared
        System.out.println(e.getMessage());
        return;
    }
```

User Profile Management

Figure 6 - Profile Management



A profile is mandatory in order for an account to be active.

A profile represents a user type and its functional characteristics (a set of access rights declared on predefined categories).

A **Profile** is defined by:

- An identifier, accessible though the **getName** method.
- The identifier of its creator, and the creation date; these are automatically set at the profile's creation time and accessible though the **getCreatorId** and the **getCreationDate** methods, respectively.
- A maximum granted access duration, which can be set and modified through the **getMaxGrantedAccessDuration** and **setMaxGrantedAccessDuration** methods, respectively. It represents the period of validity of access to the platform by an account linked to this profile. If a logged-in session exceeds this duration, the session becomes invalid and the user must log in again.
- Access rights, represented by categories (functional access rights) and "hot" data (restricted access to particular data within the scope of particular functional categories).

Each **Category** represents an access right. Each basic **Category** has both a name (accessible through the **getName** method) and a status, specifying whether access to the corresponding controlled function is granted or not. This information is accessible through the **isAllowed** method, and modifiable through the **setAllowed** method.

Some categories contain more complex information, such as the **ServiceCategory**.

Framework access is controlled by the following categories:

- Platform management (**PlatformManagementCategory**): allows access to the platform system commands, such as starting up and shutting down the platform, process deployment, and so on.
- Platform monitoring (**PlatformMonitoringCategory**): allows access to platform monitoring features: log configuration and reading, audit trail configuration and reading, SNMP management, and so on.
- Profile management (**ProfileManagementCategory**): allows the creating, updating, and deleting of account profiles.
- Account management (**AccountManagementCategory** and **SubscriberAccountManagementCategory**): allows the creating, updating and deleting of accounts and subscriber accounts.

The OTA Manager platform contains the following categories:

- Card Provisioning (**CardProvisionningCategory**): allows provisioning (that is, the batchloading of cards into the Card repository).
- Card Viewing (**CardViewingCategory**): allows read-only access to the Card repository.
- Card Editing (**CardEditingCategory**): allows read/write access to the Card repository.
- Card Group Management (**CardGroupManagement**): allows create/delete/update of groups of cards in the Card Manager.
- Service Management (**ServiceManagementCategory**): allows management of (that is, definition of) Card Management System (CMS) services.
- Service Invocation Monitoring (**InvocationMonitoringCategory**): allows execution of its own CMS service requests to be tracked.
- Service Invocation Management (**InvocationManagementCategory**): allows the execution of all CMS service requests to be tracked.
- Campaign Management (**CampaignHandlingCategory**): allows intervention in campaigns that you initiated.
- Campaign Management (**CampaignManagementCategory**): allows intervention in all campaigns running on the platform regardless of whether you initiated them or not.

The three profile classes (**AdministratorProfile**, **CCAProfile** and **SubscriberProfile**) define the basic maximum set of access rights an account of the same type can have. This means that, for example, platform management is only allowed by administrators (a **getPlatfomManagementCategory** method exists only for the **AdministratorProfile** class but not for **CCAProfile** and **SubscriberProfile**). However, this does not mean that all administrators have this basic right granted: the right can be granted for certain profile instances and not for others.

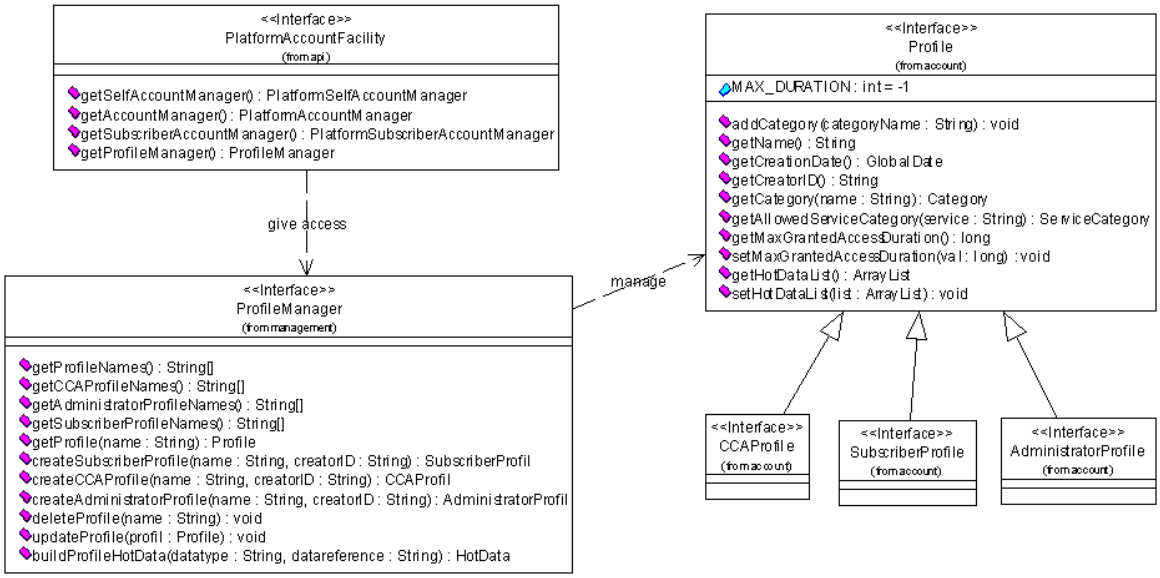
To check whether a functional access right has been granted for the profile, you must check the **isAllowed** method of the relevant **Category** object. The **Category** object is obtained using its relevant accessor (for example, the **getPlatformManagementCategory** method). If this method returns **null**, the relevant access right is NOT granted.

To grant an access right, add the requested category using the **addCategory** method on the profile, then use the **setAllowed** method on the relevant **Category** instance.

To add access rights corresponding to Framework product functions not already handled by the **Profile** sub-classes, a similar access right granting process must be followed.

A profile is initially created without any access rights; access rights must be set manually.

Figure 7 - The ProfileManager Interface



The **ProfileManager** interface allows all profile types to be managed. This functionality is available only for administrators.

It enables you to create any of the three profile types, update and delete profiles (**createAdministratorProfile**, **createCCAProfile**, **createSubscriberProfile**, **updateProfile**, and **deleteProfile** methods), and also to retrieve them by name (**getProfile** method). It is also possible to retrieve a list of all the profile names (**getProfileNames**), or a list of the profile names for each profile type (the **getAdministratorProfileNames**, **getCCAProfileNames**, and **getSubscriberProfileNames** methods).

Accessing a Profile

The following example shows how to access a profile:

```
ProfileManager manager = null;

try {
   manager = platform.getAccountFacility().getProfileManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

Profile profile = manager.getProfile("_ProfileName_");
if (profile==null) {
```

```
   System.out.println("Profile not found");
} else {
   System.out.println("Profile found: " + profile.getName());
}
```

Creating a Profile

The following example shows how to create a profile:

```
ProfileManager manager = null;

try {
   manager = platform.getAccountFacility().getProfileManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

try {
   manager.createAdministratorProfile("_ProfileName_",
platform.getAccountID());
} catch (AlreadyExistsException e) {
   // The profile may already exist
   System.out.println(e.getMessage());
}
```

The profile is created with default values and no rights, so must be configured with an update action. Refer to “Updating a Profile” on page 20.

Updating a Profile

The following example shows how to update a profile:

```
ProfileManager manager = null;

try {
   manager = platform.getAccountFacility().getProfileManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}
Profile profile = manager.getProfile("_ProfileName_");
if (profile == null) {
   System.out.println("Profile not found");
   return;
}

// Set one hour as max granted connection
profile.setMaxGrantedAccessDuration(86400);

try {
   manager.updateProfile(profile);
} catch (NoMatchingObjectException e) {
   // If the profile has not been previously declared
   System.out.println(e.getMessage());
}
```

Managing Profile Access Rights

Checking whether a Profile Has a Particular Access Right

The following example shows how to check whether a profile has a particular access right:

```
// Check the Platform Management Category
Category category =
profile.getCategory(PlatformManagementCategory.NAME);

if ((category == null) || (! category.isAllowed())) {
   System.out.println("Access to " + PlatformManagementCategory.NAME +
" is NOT granted");
} else {
   System.out.println("Access to " + PlatformManagementCategory.NAME +
" is granted");
}
```

Granting or Revoking an Access Right To a Profile

The following example shows how to grant or un-grant a particular access right to a profile:

```
// Get the Platform Management Category, only if it is an
// administrator profile

Category category =
profile.getCategory(InvocationMonitoringCategory.NAME);
// Also possible: category = ((<ProfileType>)
profile).getInvocationMonitoringCategory();

if (category == null) {
   profile.addCategory(InvocationMonitoringCategory.NAME);
   category = profile.getCategory(InvocationMonitoringCategory.NAME);
}

// Reverse the current access right for Invocation Monitoring
category.setAllowed(! category.isAllowed());

try {
   manager.updateProfile(profile);
} catch (NoMatchingObjectException e) {
   // If the profile has not been previously declared
   System.out.println(e.getMessage());
}
```

Setting a Quality of Service On a Profile

Service categories (**ServiceCategory** subclasses) are categories relevant to the card management system. Several types of such categories exist:

- Unrestricted destination Value Added Services (**UnrestrictedDestinationVASCategory** class): services on which the OTA Manager access control CANNOT apply restrictions on the targeted card. The “Send SMS Text” service is an example of an unrestricted destination service: any end user granted access to this service can target any card.
- Restricted destination Value Added Services (**RestrictedDestinationVASCategory** abstract class): services for which the OTA

Manager V5.1 access control system should apply restrictions on the targeted card. The **Update ADN** is an example of a service that may have access right restrictions on the targeted card.

Two concrete sub-classes define the level of restriction:

- **SelfVASCategory**: this category specifies that the service can only target the client's "own" card. This category is specifically for subscribers that are only allowed to modify the content of their own card.
- **AllVASCategory**: this category specifies that the services can target all the cards. This category is specifically for customer care agents that are allowed to target any subscriber's card.

- Protocol services (**ProtocolServiceCategory** class): services that are not directly accessible by end-users, but that are used by automatic applications for sending application data to the card. These services are representative of the use of the OTA Manager platform as an OTA Gateway only (and not as a provider of value added services). The **Send 03.48** service is an example of a protocol service. Such services have no target restrictions.

As service categories are relevant to the card management system, the modification of such access rights and the associated Quality of Service (QoS) has an impact on the OTA Manager platform.

The following example shows how to set quality of service on a profile:

```
ServiceCategory category = null;

// Set default quality of service for service execution for the profile.
// The QoS might have to be set on ALL the Service Categories
if (profile instanceof AdministratorProfile) {
   // Update the QoS for the AllVAS Category
   category = ((AdministratorProfile) profile).getAllVASCategory();
   updateQoS (category.getQoS());
   // Update the QoS for the UnrestrictedDestinationVAS Category
   category = ((AdministratorProfile)
profile).getUnrestrictedDestinationVASCategory();
   updateQoS (category.getQoS());
   // Update the QoS for the ProtocolService Category
   category = ((AdministratorProfile)
profile).getProtocolServiceCategory();
   updateQoS (category.getQoS());

} else if (profile instanceof CCAProfile) {
   // Update the QoS for the AllVAS Category
   category = ((CCAProfile) profile).getAllVASCategory();
   updateQoS (category.getQoS());
   // Update the QoS for the UnrestrictedDestinationVAS Category
   category = ((CCAProfile)
profile).getUnrestrictedDestinationVASCategory();
   updateQoS (category.getQoS());
   // Update the QoS for the ProtocolService Category
   category = ((CCAProfile) profile).getProtocolServiceCategory();
   updateQoS (category.getQoS());

} else if (profile instanceof SubscriberProfile) {
   // Update the QoS for the SelfVAS Category
   category = ((SubscriberProfile) profile).getSelfVASCategory();
   updateQoS (category.getQoS());
```

```
   // Update the QoS for the UnrestrictedDestinationVAS Category
   category = ((SubscriberProfile)
profile).getUnrestrictedDestinationVASCategory();
   updateQoS (category.getQoS());
}

// Update the profile
try {
   manager.updateProfile(profile);
} catch (NoMatchingObjectException e) {
   // if the profile has not been previously declared
   System.out.println(e.getMessage());
}
```

Where **updateQoS** is:

```
protected static void updateQoS (ProfileQoS qos) {
   qos.setGuaranteeOfDelivery(new Boolean(true));
   qos.setGuaranteeOfExecution(new Boolean(true));
   qos.setVelocity(ProfileQoS.VELOCITY_HIGH);
   qos.setPriority(ProfileQoS.PRIORITY_NORMAL);
}
```

Setting Allowed Services and Allowed Protocol for Service Execution

The following example shows how to set allowed services and allowed protocol for service execution:

```
ServiceCategory category = null;
String serviceName = "UpdateADN"; //"_ServiceName_";
String protocol = "WIR"; //Values allowed are WIR, SMS and CATTP


// Only the VAS service list and protocol is modified in this example.
if (profile instanceof AdministratorProfile) {
   // Add the service for the AllVAS Category
   category = ((AdministratorProfile) profile).getAllVASCategory();

} else if (profile instanceof CCAProfile) {
   // Add the service for the AllVAS Category
   category = ((CCAProfile) profile).getAllVASCategory();

} else if (profile instanceof SubscriberProfile) {
   // Add the service for the SelfVAS Category
   category = ((SubscriberProfile) profile).getSelfVASCategory();
}

// Configure Service Category to grant access to the given service (for
service invocation purpose)
// In this example, the service is added to the list of already allowed
services of the profile
String [] currentListServices = category.getAllowedServices();

String [] listServices = null;
if (currentListServices == null) {
   listServices = new String [1];
   listServices[0] = serviceName;
} else {
```

```
   listServices = new String [currentListServices.length + 1];
   System.arraycopy(currentListServices, 0, listServices, 0,
currentListServices.length);
   listServices[currentListServices.length] = serviceName;
}

category.setAllowedServices(listServices);
category.setAllowed(true);

// Configure service invocation rights to be able to use the given
protocol
// In this example, the given protocol becomes the only one allowed
String[] listProtocols = new String[1];
listProtocols[0] = protocol;
category.setAllowedProtocols(listProtocols);

// Update the profile
try {
   manager.updateProfile(profile);
} catch (NoMatchingObjectException e) {
   // If the profile has not been previously declared
   System.out.println(e.getMessage());
}
```

Deleting a Profile

The following example shows how to delete a profile:

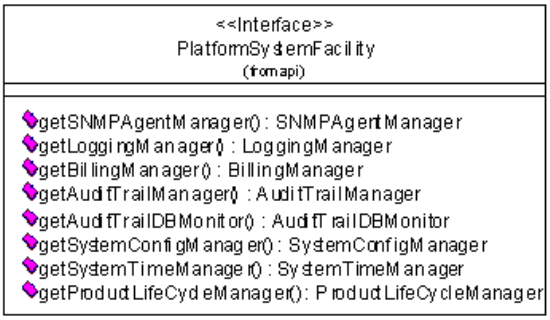
```
ProfileManager manager = null;

try {
   manager = platform.getAccountFacility().getProfileManager();
} catch( PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}
try {
   manager.deleteProfile("_ProfileName_");
} catch (NoMatchingObjectException e) {
   // The profile may not exist
   System.out.println(e.getMessage());
} catch (ViolatedDependencyException e) {
   // The profile may be used by accounts, so deleteion is refused
   System.out.println(e.getMessage());
}
```

System Facilities

The Framework provides a set of **Facility** modules to OTA Manager products: these relate to the management of a central time reference, configuration management, SNMP counter and trap handling, and so on.

The System Facilities part of the Core API allows you to administer one part of these facilities modules.

Figure 8 - PlatformSystemFacility

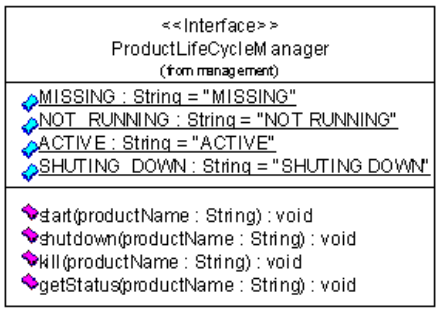


The **PlatformSystemFacility** interface allows an administrator to access the individual facility modules' administration features:

- Administration of the Product Life Cycle facility (**ProductLifeCycleManager**): Product starting up and shutting down.
- Information on the global system, and administration of the Product Configuration Parameters facility (**SystemConfigManager**).
- Get the current time and information on the central time managed by the System Time facility (**SystemTimeManager**).
- Administration of the Logging facility (**LoggingManager**): logging activation, deactivation, and configuration.
- Administration of the Billing facility (**BillingManager**): billing activation, deactivation, and configuration.
- Administration of the Audit Trail facility (**AuditTrailManager** and **AuditTrailDBMonitor**): audit trail activation, deactivation, configuration, and management of queries in the Audit trail database.
- Administration of the SNMP facility (**SNMPManagerAgent**): SNMP agent activation, deactivation, and configuration.

Product Life Cycle Facility Management

Figure 9 - ProductLifeCycleManager

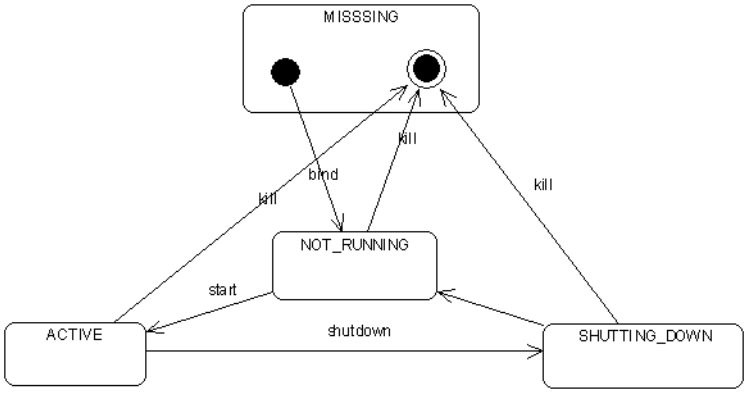


The **ProductLifeCycleManager** objects allow you to start (**start** method), shut down (**shutdown** method) or kill (**kill** method) a specified product.

It also enables you to obtain a product status (**getStatus**), returning one of the following: MISSING (product process is not present), NOT_RUNNING (product process is running, but is not functionally available; it awaits a **start** command), ACTIVE (the product is functionally available), or SHUTTING_DOWN (the product has just received a **shutdown** call; it will soon be in the NOT_RUNNING) status.

"Figure 10" displays the full product life cycle:

Figure 10 - The Product Life Cycle



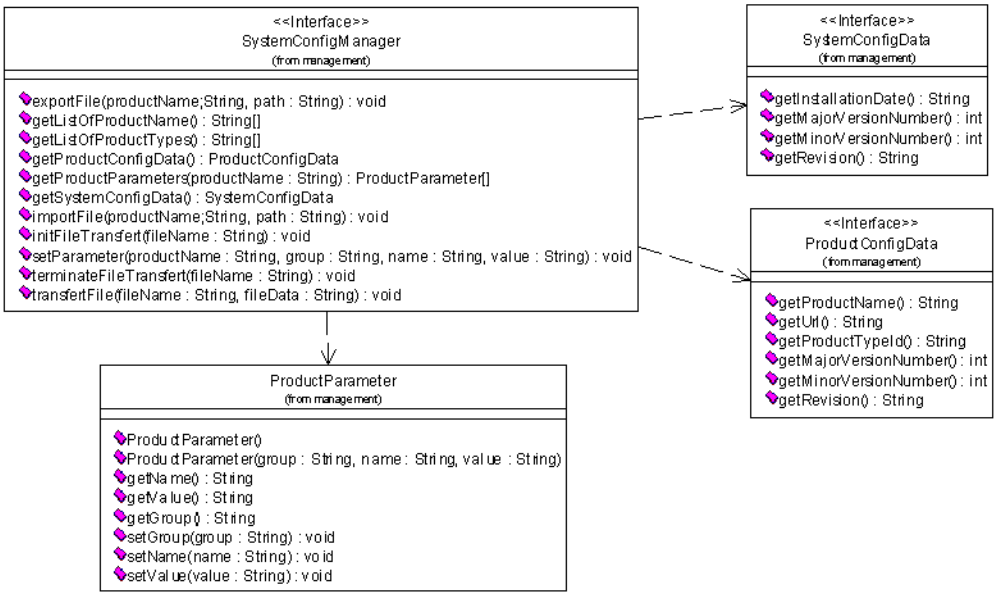
Starting a Product

The following example shows how to obtain a product's status and to start it:

```
// Standard Framework System Facility
PlatformSystemFacility facility = platform.getSystemFacility();

// Get the manager
ProductLifeCycleManager manager = null;
try {
   manager = facility.getProductLifeCycleManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

// Get the Product status...
String status = manager.getStatus("_ProductName_");
System.out.println("Product status : " + status);

// ... and start it, with respect to the Product Life Cycle
if ((status.equals(ProductLifeCycleManager.MISSING)) ||
    (status.equals(ProductLifeCycleManager.NOT_RUNNING))) {

   try {
      manager.start("_ProductName_");
   } catch (Exception e) {
      // Error might occur during the startup process
      System.out.println(e.getMessage());
   }
}
```

System Configuration Facility Management

Figure 11 - The SystemConfigManager Interface



The **SystemConfigManager** allows you to retrieve information concerning the global system version and configuration. It is divided into the following areas:

- Retrieval of the Framework version (**getSystemConfigData** method).
- Retrieval of global information concerning the products currently hosted by the Framework:
  - List of all the available product types (**getListOfProductTypes** method)
  - List of all the available product names (**getListOfProductNames** method)
  - Version information for a particular product (**getProductConfigData** method): type of the product, installation date, and version.
- Retrieval, update, and creation of configuration parameters.

The management of product parameters enables OTA Manager's internal code to be externally configurable and to have a different configuration and behavior depending on the parameters' values: Typical types of information that are stored in the product parameters include:

- The size of internal elements, such as queues
- Default technical or business-related values, such as timeout values
- Any default business-related values.

The product parameters are uniquely identified by a {*group*; *name*} pair. A **String** value can then be associated with a pair. It is possible to retrieve a particular product parameter value by using the **getParameter** method, giving the parameter group and name, and the relevant product.

The full list of configuration parameters for a product can also be retrieved using the **getProductParameters** method. The returned array of **ProductParameter** then allows access to the parameter group, name, and value. Note that, at present, the **set***xxx* methods have no purpose as they only enable you to make local modifications to the **ProductParameter** instance, since there are no methods that enable you to update or to define a new product parameter using a **ProductParameter** instance.

It is, of course, possible to set a particular product parameter value by using the direct **setParameter** method. This method updates the parameter value of an existing parameter only. The direct creation of a new parameter is currently not available through a dedicated entry point: the **import** function should be used to do this: it is possible to directly import the entire set of parameters specified in a comma-separated value (CSV) file, using the **importFile** method. When importing, all the current product parameters are removed, then the new set of parameters is inserted: the imported parameters are consequently NOT appended to the current ones, they replace them. It is thus possible to create new parameters using this import feature.

The reverse action (export) is achieved using the **exportFile** method.

There follows an example of an import/export file:

```
CFM,LOAD_FILE_AID,1122334455
CFM,TONNPI_INT_DEST,91
CFM,TONNPI_NAT_DEST,A1
CHANNELMONITORSMS,ACTIVE,YES
CHANNELMONITORWIR,ACTIVE,YES
CI,AC_TIMEOUT,600000
CI,QUEUES_SIZE,100
CI,RETRY_DELAY,1000
CI,RETRY_NBR,3
GENERAL,JDBC_URL,jdbc:oracle:thin:@G058619:1521:GXS
GENERAL,JDBC_MAX_CONNECTIONS,5
GENERAL,DB_CONNECTIONS_NBR_READ,5
GENERAL,DB_CONNECTIONS_NBR_WRITE,5
....
```

Note that the file names specified in both the import and export method relate to the Framework file system, not to the local Core API client file system. No feature currently exists for transferring this file from the Framework file system to the local Web-based client file system (or vice versa).

Remember also that, on the internal product source code side, retrieving product parameters' values is done using a different API: the operational API of the facility.

Retrieving Product Parameter Values

The following example shows how to retrieve the product configuration parameters:

```
// Standard Framework System Facility
PlatformSystemFacility facility = platform.getSystemFacility();

// Get the manager
SystemConfigManager manager = null;
try {
   manager = facility.getSystemConfigManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

// Get the configuration parameters of a Product
ProductParameter[] params =
manager.getProductParameters("_ProductName_");

for (int i = 0; i < params.length; i++) {
   System.out.println ("Group: " + params[i].getGroup()
```

```
                        + " / Name: " + params[i].getName()
                        + " / Value: " + params[i].getValue());
}
```

Setting a Product Parameter Value

The following example shows how to set a product parameter value:

```
// Standard Framework System Facility
PlatformSystemFacility facility = platform.getSystemFacility();

// Get the manager
SystemConfigManager manager = null;
try {
   manager = facility.getSystemConfigManager();
} catch (PlatformException e) {
   // Acces to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

// Set the value of the _Group_/_Name_ parameter of product
_ProductName_
try {
   manager.setParameter("_ProductName_", "_Group_", "_Name_",
"_Value_");
} catch (NoMatchingObjectException e) {
   // The given parameter does not exist.
   System.out.println("Canceled: The given parameter does not exist.");
   System.out.println(e.getMessage());
}
```

Exporting and Importing a Product Parameters File

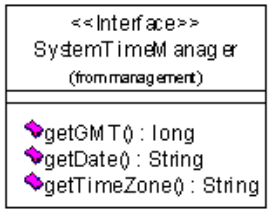
The following example shows how to export and import a Product configuration parameters file:

```
// Standard Framework System Facility
PlatformSystemFacility facility = platform.getSystemFacility();

// Get the manager
SystemConfigManager manager = null;
try {
   manager = facility.getSystemConfigManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

// Here are the various parts of the file to import
// Might of course be obtained differently !
String filePart1 =
   "Group1;Name1;Value1\nGroup1;Name1;Value2\n;Group1;Name2;Value3\n";
String filePart2 = "Group2;Name1;Value1";

// Transfer the file
try {
   // Inits the file transfer
```

```
   manager.initFileTransfert("_FileName_");

   // Transfer the file
   manager.transfertFile("_FileName_", filePart1);
   manager.transfertFile("_FileName_", filePart2);

   // End the transfer
   manager.terminateFileTransfert("_FileName_");
} catch (Exception e) {
   // Error during the transfer
   System.out.println(e.getMessage());
   return;
}

try {
   // Export the file
   manager.exportFile("_ProductName_", "_FileName_");

   // The file supposed to be externally modified being re-imported

   // Here we re-import it immediately
   manager.importFile("_ProductName_", "_FileName_");
} catch (IOException e) {
   // Error during the export/import
   System.out.println(e.getMessage());
}
```

System Time Facility Management

Figure 12 - The SystemTimeManager Interface



The **SystemTimeManager** allows you to retrieve either the current time in the GMT time zone (in a long format), using the **getGMT** method, or in the local time zone (in **String** format), that is, local to where the Framework is currently running, using the **getDate** method.

The local time zone can be retrieved using the **getTimeZone** method.

Getting the Date

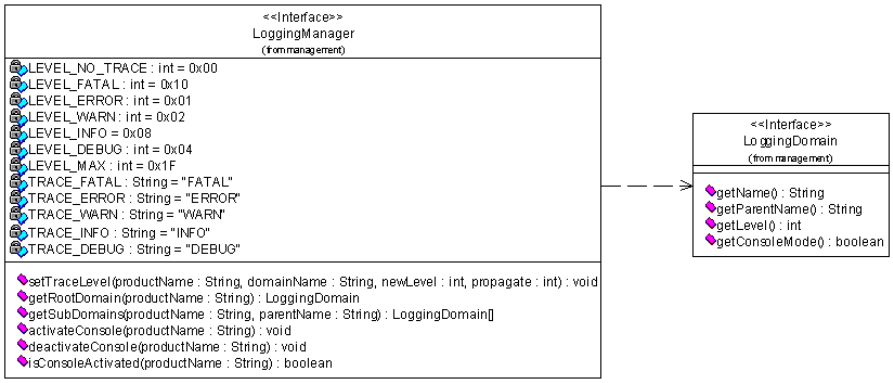
The following example shows how to get the current date:

```
// Standard Framework System Facility
PlatformSystemFacility facility = platform.getSystemFacility();
// GC OTA 2.0 System Facility
//SystemFacility facility = platform.getSystemFacility();

// Get the manager
SystemTimeManager manager = null;
try {
```

```
   manager = facility.getSystemTimeManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

// Get the dates
long gmtDate = manager.getGMT();
System.out.println("GMT date:" + new Date (gmtDate).toString());
System.out.println("Local date: " + manager.getDate());
System.out.println("Local time zone: " + manager.getTimeZone());
```

Logging Facility Management

Figure 13 - The Logging Facility



The **Logging** facility provides the simple ability to write traces to the product console, to a file, or both.

Each product can dynamically build its own hierarchy of domains and sub-domains (as a tree) in order to group log entries together: domains can, for example, be used to differentiate between debugging logs and support logs, but also to divide the overall flow of logs into functional or technical modules of the product.

A trace level should also be specified at log generation time. Five trace levels are defined: **Fatal**, **Error**, **Warning**, **Info,** and **Debug**.

The **LoggingManager** interface allows you to configure the **Logging** facility. It then enables you to:

- Get the logging domain tree structure for a particular product: the **getRootDomain** method returns the root of the tree, and the **getSubDomains** method returns the sub-domains of a given domain (specified by its name).

  Domains are represented by a **LoggingDomain** instance that gives the domain name (**getName**), its parent domain name (**getParentName**), and information concerning the trace level (**getLevel**) and the console activation status (**getConsoleMode**) for the domain.
- Set the domain trace level (the **setTraceLevel** method): given information are the name of the domain, the level of trace, and whether the trace level setting should be propagated to all the sub-domains of the targeted one.
- Activate (**activateConsole**) or deactivate (**deactivateConsole**) the trace display on the product console. The trace console activation status can also be retrieved for a product (**isConsoleActivated**).

The trace level is expressed as an integer value, which is a coding scheme based on a one-byte mask:

- No trace: 0x00
- Fatal: 0x10
- Error: 0x01
- Warn: 0x02
- Info: 0x08
- Debug: 0x04
- Maximum: 0x1F

Trace levels have a priority. This means that the **Error** level is “higher” than the **Warn** level, which is higher than the **Info** level, which is in turn higher than the **Debug** level.

When specifying a particular trace level through the **setTraceLevel** method, the traces corresponding to the given level and all higher levels are enabled. For example, an integer value of 1 means that only the **Error** traces are enabled. 2 means that **Warn** and **Error** traces are enables, 8 (not 4!) means that **Info**, **Warn** and **Error** are enabled, and 4 (not 8!) means that all levels are granted.

Zero (0) means that no trace is generated.

When a trace of a unselected domain is generated by a product, it is ignored and is neither displayed on the product console nor written to file.

Retrieving Logging Trace Domains

The following example shows how to retrieve the Logging trace domains:

```
// Standard Framework System Facility
PlatformSystemFacility facility = platform.getSystemFacility();

// Get the manager
LoggingManager manager = null;
try {
   manager = facility.getLoggingManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

// Inner class that recursively print the logging domains
class SubDomainPrinter {
   LoggingManager mgr = null;
   String productName = null;
   String indent = "";

   public SubDomainPrinter (LoggingManager mgr, String productName) {
      this.mgr = mgr;
      this.productName = productName;
   }

   public void print (String domain) {
      // Print the domain name
      System.out.println (indent + domain);
      String oldIndent = indent;
      indent = indent + "   ";
```

```
      // Recursive call on all the sub-domains
      LoggingDomain[] subDomains = mgr.getSubDomains(productName,
domain);
      for (int i = 0; i < subDomains.length; i++) {
         print (subDomains[i].getName());
      }

      indent = oldIndent;
   }
}

// Recursively print the domain names, starting by the root domain
SubDomainPrinter printer = new SubDomainPrinter (manager,
"_ProductName_");
printer.print(manager.getRootDomain("_ProductName_").getName());
```

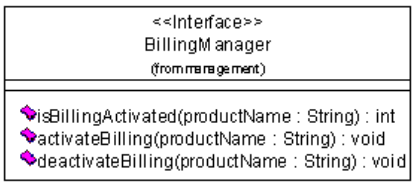
Setting the Logging Trace Level

The following example shows how to set the Logging trace level:

```
// Standard Framework System Facility
PlatformSystemFacility facility = platform.getSystemFacility();

// Get the manager
LoggingManager manager = null;
try {
   manager = facility.getLoggingManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}
// Set the loggin trace level to the root domain, propagating to all the
sub-domains
// Trace level is set to Error, which means 1
manager.setTraceLevel("_ProductName_",
                      manager.getRootDomain("_ProductName_").getName(),
                      manager.LEVEL_MAX, true);
```

Billing Facility Management

Figure 14 - Billing Facility Management



The **BillingManager** allows you to activate (**activateBilling**) and deactivate (**deactivateBilling**) CDR-format billing ticket generation for a particular product. The billing activation status can also be retrieved for a particular product (**isBillingActivated**).

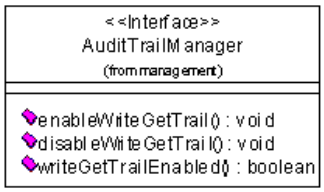
Activating and Deactivating Billing

The following example shows how to activate and deactivate Billing:

```
// Standard Framework System Facility
PlatformSystemFacility facility = platform.getSystemFacility();

// Get the manager
BillingManager manager = null;
try {
   manager = facility.getBillingManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

// Activate / deactivate the billing
System.out.println("Is Billing activated: " +
manager.isBillingActivated("_ProductName_"));

manager.activateBilling("_ProductName_");
System.out.println("Is Billing activated: " +
manager.isBillingActivated("_ProductName_"));

manager.deactivateBilling("_ProductName_");
System.out.println("Is Billing activated: " +
manager.isBillingActivated("_ProductName_"));
```

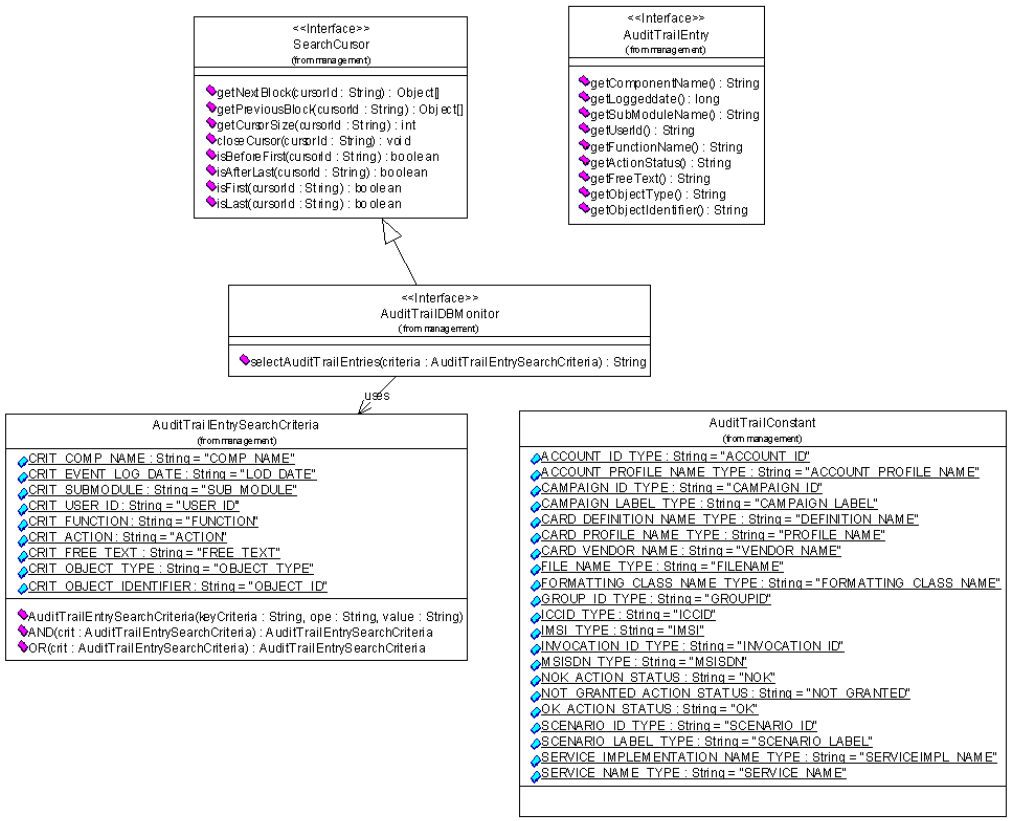
Audit Trail Facility Management

Figure 15 - The AuditTrailManager Interface



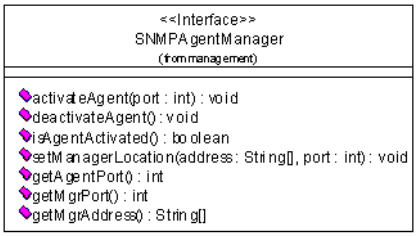
The **AuditTrailManager** allows you to activate (**enableWriteGetTrail**) and deactivate (**disableWriteGetTrail**) Audit Trail record generation for the OTA Manager platform. The Audit Trail activation status can also be retrieved (**writeGetTrailEnabled**).

Figure 16 - The Audit Trail Facility



SNMP Facility Management

Figure 17 - SNMP Facility Management



The **SNMPAgentManager** allows you to administer the SNMP agent of the Framework.

This agent manages SNMP traps and counters handled by products hosted by the Framework. It does this using a dedicated port open on the machine that is running the SNMP agent facility. This port is specified at the agent's activation time (**activateAgent**) and can be retrieved through the **getAgentPort** method.

Note that, in order to change the port of the agent, you must first deactivate the agent, then reactivate it with the new port number. Activation of an already active agent has no effect.

Once activated, the SNMP Agent is able to answer to the standard **get** and **getNext** commands, giving the object OID. The value of the SNMP objects managed by the various Framework is then delivered:

- By calling **get**, the value of the object corresponding to the given OID is returned
- By calling **getNext**, the value of the object corresponding to the given OID, and the next OID of the MIB (which has a bound object) is returned (as a **String** array containing the object value in position 0, and the next OID in position 1).

Connecting to an SNMP Manager also enables you to send any SNMP traps raised by the Framework products. The various SNMP Manager host names and ports are specified using the **setManagerLocation** method.

Refer to the *OTA Manager V5.1 Administration Guide* for a description of the various MIBs of the Framework and of the hosted products.

Activating the SNMP Agent

The following example shows hows to activate the SNMP agent:

```
// Standard Framework System Facility
PlatformSystemFacility facility = platform.getSystemFacility();

// Get the manager
SNMPAgentManager manager = null;
try {
   manager = facility.getSNMPAgentManager();
} catch (PlatformException e) {
   // Access to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

// Activate the SNMP Agent on port 3161
try {
   manager.deactivateAgent();
```

```
   manager.activateAgent(3161);
} catch (IOException e) {
   // Error during the port opening
   System.out.println(e.getMessage());
}

// Give a SNMP Manager location (localhost, as an example)
manager.setManagerLocation(new String [] {"127.0.0.1"}, 162);
```

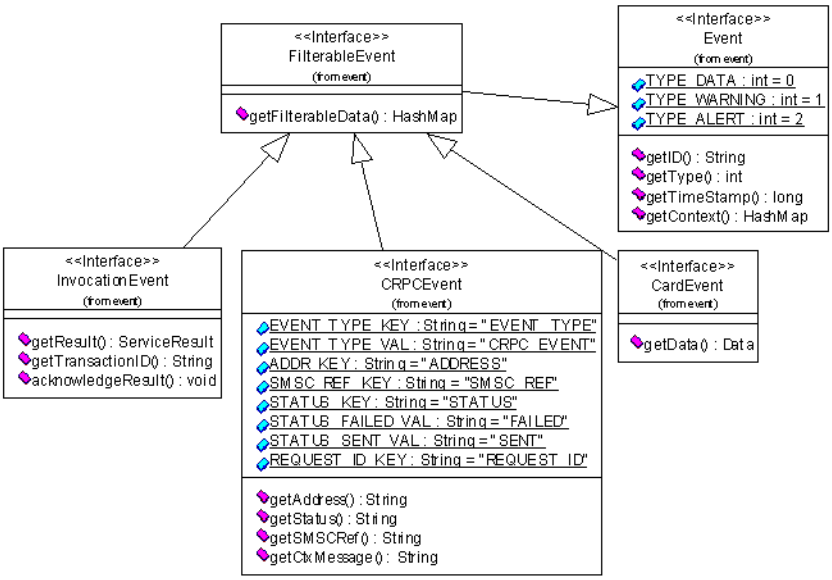
Getting an SNMP Object Value

The following example shows how to get an SNMP object's current value:

```
// Standard Framework System Facility
PlatformSystemFacility facility = platform.getSystemFacility();

// Get the manager
SNMPAgentManager manager = null;
try {
   manager = facility.getSNMPAgentManager();
} catch (PlatformException e) {
   // Acces to the manager can be denied
   System.out.println(e.getMessage());
   return;
}

// Get a particular object
System.out.println (manager.get("1.3.6.1.4.1.5496.2.1.0"));

// Run over the overall MIB to get all the objects, using the getNext
method.
String previousOID = "Starting...";
String [] node = {"", "1.3.6.1.4.1.5496"};

while ((node.length == 2) && (node[1] != null) && (!
node[1].equals(""))) {
   System.out.println (previousOID + " : " + node [0]);

   previousOID = node [1];
   node = manager.getNext(node [1]);
}
```

Events

Figure 18 - Events



An event is something the platform can spontaneously publish to all Core API client applications.

Each event carries the following information:

- An identifier
- A type: DATA, WARNING, ALERT
- A time stamp (the date and time when the event is published)
- Contextual data depending on the nature of the event (on which filtering data applies)

A filterable event retrieves the filter data it matches.

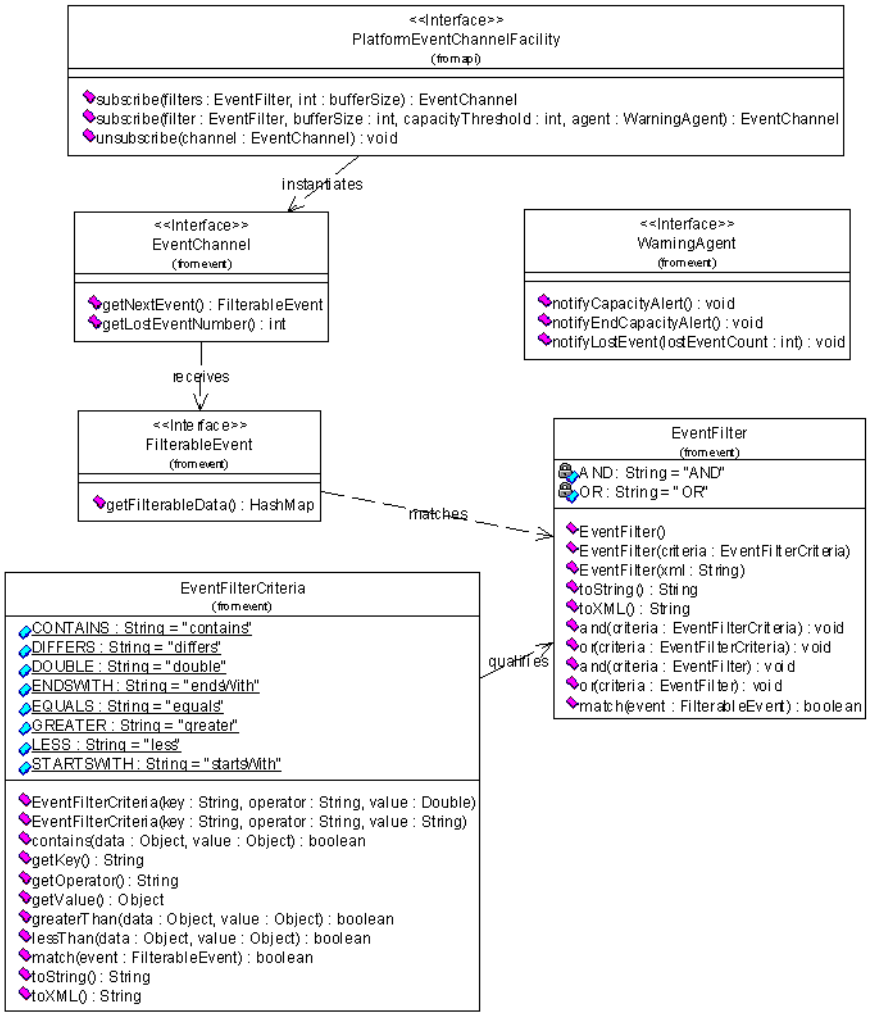
An event can be raised by either a Framework module or any product deployed on the Framework. At present, three kinds of concrete events are published:

- **CardEvent**, raised by OTA Manager: contains the data received from an SMS_SUBMIT (MO) message issued by a card. This can be a GSM 03.48 transport command packet, in which case the data is unformatted (using the security configuration in the message). For any other type of secured transport, the data is published exactly as received.
- **CRPCEvent**, raised by OTA Manager: published when the first SMS of a Card Management System service execution has been acknowledged by the SMSC.
- **InvocationEvent**, raised by OTA Manager: published each time a Card Management System invocation execution ends (only for an asynchronous execution request)). It carries information on the end status (and possibly additional GSM 03.48 response packet data, if the service was configured to request a proof of receipt (PoR)).

An MO message containing a GSM 03.48 response packet is not published as an event. The response packet (if requested) is used by OTA Manager V5.1 to determine whether the service execution failed or succeeded. The received response packet is included in the result returned to the service execution submitter (**Result.getResultData()** method).

Note that in OTA Manager V5.1, the **Invocation** event is not managed in exactly the same way as card and CRPC events. Refer to “Invocation Events” on page 42 for more information.

Figure 19 - Event Management



The **EventChannelFacility** allows you to subscribe and unsubscribe to events by specifying event filters.

When subscribing to an event, you are requested to specify a buffer size where received events are stored (in memory only) before being delivered using the **EventChannel.getNextEvent()** method.

If using a **WarningAgent**, then you are also requested to specify a capacity threshold on the buffer. **WarningAgent** is notified of events concerning management of the buffer.

Card Events

A **CardEvent** is characterized by filterable data containing:

```
"EVENT_TYPE" = "MO_EVENT"
```

A **CardEvent** can be filtered on the following keywords:

```
"ADDRESS", "iADDRESS", "MSISDN", "iMSISDN", "ICCID", "iICCID",
"TONNPI","PROTOCOL", "CHANNEL", "PID", "DCS", "TP_OA", "iTP_OA",
"TONNPI_OA"
```

A **CardEvent** containing a binary MO is characterized by:

```
"DATA_TYPE" = "BINARY"
```

Additional keys supported for this specific card event are:

```
"DATA", "UDH", "FORMAT_TYPE", "SPI", "TAR", "APPLI_NUMBER"
```

A **CardEvent** containing a text MO is characterized by:

```
"DATA_TYPE" = "TEXT"
```

Additional keywords supported for this specific card event are:

```
"DATA"
```

Subscribing to any MO Event

The following example shows how to subscribe to a Card event:

```
EventFilter eventFilter = null;
EventChannelchannel = null;
EventFilterCriteriacriteria = null;

PlatformEventChannelFacility facility =
platform.getPlatformEventChannelFacility();

// Build an empty filter
try {
   criteria = new EventFilterCriteria("EVENT_TYPE",
EventFilterCriteria.EQUALS, "MO_EVENT");
   eventFilter = new EventFilter(criteria);
} catch (MalformedEventFilterException e) {
   // If event is Malformed
   System.out.println(e.getMessage());
   return;
}

channel = facility.subscribe(eventFilter, 100);
```

Subscribing to any Binary MO Event

The following example shows how to subscribe to any binary Card event:

```
EventFilter eventFilter = null;
EventChannelchannel = null;
EventFilterCriteriacriteria = null;
EventFilterCriteriacriteria2 = null;

PlatformEventChannelFacility facility =
platform.getPlatformEventChannelFacility();
```

```
// Build an empty filter
try {
   criteria = new EventFilterCriteria("EVENT_TYPE",
EventFilterCriteria.EQUALS, "MO_EVENT");
   eventFilter = new EventFilter(criteria);
   criteria2 = new EventFilterCriteria("DATA_TYPE",
EventFilterCriteria.EQUALS, "BINARY");
   eventFilter.and(criteria2);

} catch (MalformedEventFilterException e) {
   // If event is Malformed
   System.out.println(e.getMessage());
   return;
}

channel = facility.subscribe(eventFilter, 100);
```

Receiving Events

Reception of events is done using the **getNextEvent** blocking method of a previously created **EventChannel**.

```
FilterableEvent event = null;
     while (true) {
        // Waiting CardEvent
        event = channel.getNextEvent();

        //Event should be a CardEvent
        CardEvent cardEvent = (CardEvent) event;
        HashMap filterableData = event.getFilterableData();

        //Get for exemple the MSISDN for filterable data
        String msisdnReceived = (String)filterableData.get("MSISDN");
        if(msisdnReceived == null){
           System.out.println("Msisdn is null Event ignored
                              (it should not be a data event)");
           continue;
        }
        System.out.println("Received event with msisdn:"
                           + msisdnReceived);

        Data dataReceived = cardEvent.getData();
        if(! (dataReceived instanceof ByteBufferData)) {
           System.out.println("Received  event doesn't contain
                     binary data:" + dataReceived.getClass() );
           continue;
        }
        ByteBufferData bufferData = (ByteBufferData)dataReceived;

        byte[] dataBytes = bufferData.getValue();

        //...
     }
```

Unsubscribing an Event

Unsubscribing is done on a particular **EventChannel**, so it is required to retrieve the event channel at unsubscription time.

The following example shows how to unsubscribe from a Card event:

```
EventChannelFacility facility = platform.getEventChannelFacility();

// Give the channel event you want to close
facility.unsubscribe(channel);
```

CRPC Events

A **CRPCEvent** is characterized by filterable data containing:

```
"EVENT_TYPE" = "CRPC_EVENT"
```

A **CardEvent** can be filtered on following keys:

```
"ADDRESS", "SMSC_REF", "STATUS", "REQUEST_ID"
```

The “STATUS” key can contain “FAILED” or “SENT”.

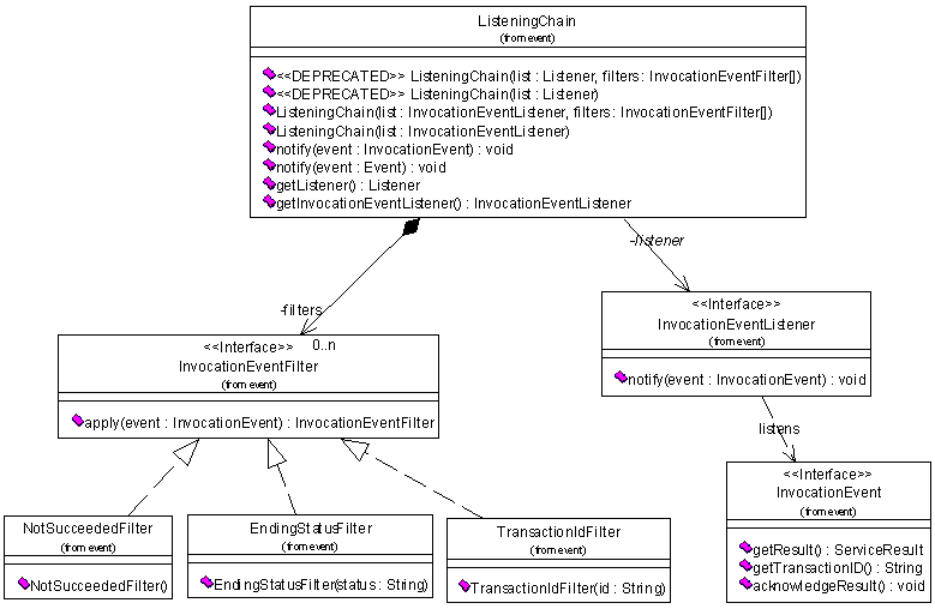
Refer to “Card Events” on page 40 for examples on how to subscribe to and unsubscribe from particular events.

Invocation Events

InvocationEvent, in this version of OTA Manager and for backwards compatibility, cannot be received in the same way as the other events described previously.

Subscription on **InvocationEvent** is done using **ListeningChain** and **InvocationFacility**, and by implementing an **InvocationEventListener**.

Figure 20 - Invocation Events



The **EndingStatusFilter** can be specified using the following values (which can be found as constants on the **ServiceResult** interface):

```
"FAILED", "SUCCEEDED", "EXPIRED", "DELETED", "SUPPRESSED"
```

An **InvocationEvent** is short-lived. If nothing is subscribed to a particular event, the API ignores it. Nevertheless, if guarantee of delivery was required when the invocation was submitted, the equivalent of the event can be obtained by monitoring actions using the invocation handler or manager (depending on your user profile).

Subscribing To Any Invocation Event

To do this, register your listener without specifying any filter:

```
class LocalListener implements InvocationEventListener{
   /**
    * Receive events
    */
   public void notify(InvocationEvent event) {
      System.out.println(event.toString());
   }
}

//How to access the CardManagerAccessPoint?
CardManagerAccessPoint point = null;
try {
   point = (CardManagerAccessPoint) platform.getProductAccessPoint(
           CardManagerAccessPoint.class, "RCA1");
} catch (ProductNotFoundException exp) {
   System.out.println("Unable to access 'CardManagerAccessPoint'");
   return;
}

// Build the listening chain
Listener listener = new LocalListener();
ListeningChain chain = new ListeningChain(listener , null);

try {
   point.getInvocationFacility().addChain(chain);
} catch (ChainAlreadyDefinedException e) {
   // If the chain is already installed
   System.out.println(e.getMessage());
} catch (PlatformException e) {
   // This may happen if Card manager is not available
   System.out.println(e.getMessage());
}
```

Subscribing To Any Failed Service Execution Event

The following example shows how to subscribe to any failed service execution event:

```
EndingStatusFilter[] filters = new EndingStatusFilter[1];
InvocationEventListenerlistener;
ListeningChainchain;

// Builds the filter on the FAILED ending status
filters[0] = new EndingStatusFilter(ServiceResult.ENDSTATUS_FAILED);

// Build the listener
listener = new MyInvocationEventListener();

// Build the listening chain
chain = new ListeningChain(listener , filters);
```

```
try {
   platform.getInvocationFacility().addChain(chain);
} catch (ChainAlreadyDefinedException e) {
   // If the chain is already installed
   System.out.println(e.getMessage());
} catch (PlatformException e) {
   // This may happen if Card manager is not available
   System.out.println(e.getMessage());
}
```

Unsubscribing to Invocation Events by Removing the Listening Chain

The following example shows how to un-subscribe to Invocation events by removing listening chain:

```
try {
   platform.getInvocationFacility().removeChain(chain);
} catch (NoMatchingChainException e) {
   // If chain undefined
   System.out.println(e.getMessage());
} catch (PlatformException e) {
   // This may happen if Card manager is not available
   System.out.println(e.getMessage());
}
```

Implementing an Invocation Filter

The filter must implement the **InvocationEventFilter** interface. The method **apply(InvocationEvent event)** must return **null** if the event does not match the filter, or itself (the filter) if the event matches.

The following example shows how to implement an Invocation filter:

```
package com.gemplus.gxs.api.invocation.event;

import com.gemplus.gxs.api.invocation.ServiceResult;
/**
 * This invocation event filter matches only failed invocations
 */
public class NotSucceededFilter implements InvocationEventFilter {
   /**
    * Implicit Constructor
    */
   public NotSucceededFilter(){
   }

   /**
    * Checks if the event matches this filter
    * @return invocation event filter if the event matches, else null
    */
   public InvocationEventFilter apply(InvocationEvent event){
      if (event!=null) {
         ServiceResult sr = event.getResult();
         if (sr!=null){
            String status = sr.getEndingStatus();
            if (! ServiceResult.ENDSTATUS_SUCCEEDED.equals(status)){
               return this;
```

```
                }
            }
        }
        return null;
    }
}
```

4

Business APIs

Access

Once connected to the platform, it is possible to access business-specific APIs using the **PlatformAccessPoint.getProductAccessPoint()** method. This enables you to provide access to the Core API extensions of all products that have implemented the Product Access Point concept.

The business to retrieve is specified using the relevant **ProductAccessPoint** class and the name of the product to connect to.

The following example shows how to retrieve the Card Manager access point, providing the targeted Card Manager product name:

```
CardManagerAccessPoint point = null;
try {
   //Use the product name declared
   point = (CardManagerAccessPoint) platform.getProductAccessPoint(
           CardManagerAccessPoint.class, productName);
} catch (ProductNotFoundException exp) {
   System.out.println("Unable to access 'CardManagerAccessPoint'");
   return;
}

System.out.println("CardManagerAccessPoint name" +
point.getProductName());
```

Optionally, when starting the Core API you can set the two system properties **gxs.api.eventnotifier.threadpool.size** and **gxs.api.eventnotifier.threadpool.queuesize**, which respectively determine the number of threads to use and the queue size that the Core API creates in order to handle event reception. The default values are 5 (threads) and 100 (queue capacity for events), respectively.

Invoking a Service

With respect to service invocations, the Core API can be used in two different ways:

- *Simple* mode, allowing service invocations but with a minimum of parameters. The platform is free to fill in missing parameters with the default values set in its configuration. The Core API is then “easy” to use and only a few lines of code are necessary to request service execution.

- *Expert* mode, when all the concepts are exposed, but must be set. The same set of services are available, but with improved control over their execution. The Core API is more “complex” to use and more code is necessary.

Service invocation is performed in several steps:

1 First, you must build a “chain of responsibility”, where each element of the chain allows certain settings to be set.

2 Second, you build invocation parameters (allowing you to set certain “quality of service” parameters).

3 Third, you issue the service request on a particular element of the chain.

Step One: Building the Invocation Chain

The following describes each element of the invocation chain.

Card

The card is the core business of the Card Manager product. A card may be contained in a terminal (**CardHolder**). You can specify the card holder by using the **SmartCard.setHost()** method. A card may contain applications.

Card Holders

The **CardHolder** object contains the card holder for the service execution.

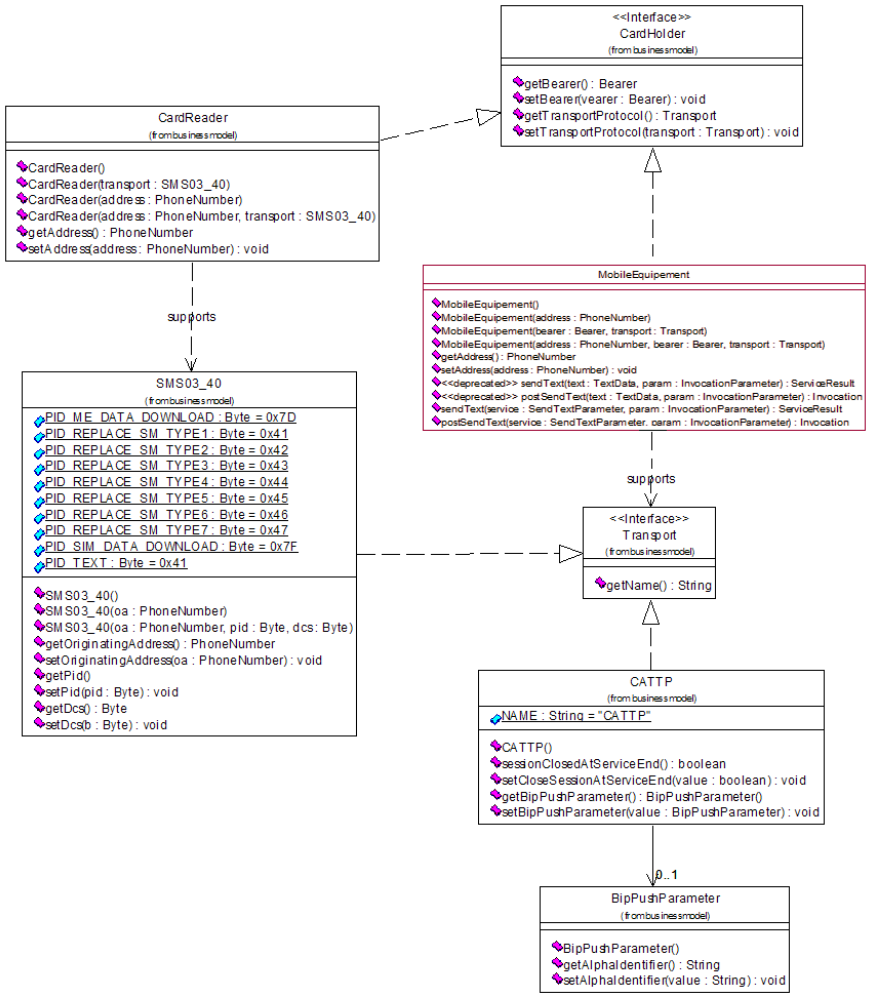
The holder of a card is specified using the method **SmartCard.setHost()** method. A **CardHolder** may be:

- A **MobileEquipment**. The phone number is optional. If the mobile handset contains a card then the phone number is identified as the one carried by the card. Nevertheless, it is possible to invoke the “send text” service directly on the mobile. In this case, it is not mandatory to specify the card in the invocation chain (but in this case, the phone number becomes mandatory in order to identify the mobile).
- A **CardReader**. The phone number is optional. If the card reader contains a card then the phone number is identified as the one carried by the card.

Each **CardHolder** can support several different transport protocols. Currently SMS (GSM 03.48) and CAT-TP (Card Application Toolkit Transport Protocol) are supported.

The selected equipment indicates the way in which the platform attempts to reach the card. **MobileEquipment** indicates OTA mode using the GSM 03.40 or CAT-TP protocol. The **CardReader** indicates connected mode, known as WIR (Wired Internet Reader), also using the GSM 03.40 transport protocol.

Figure 21 - Managing Card Holders and CAT-TP Objects



The **SMS03_40** object allows you to specify, for each service execution:

- The originating address, as a **PhoneNumber**
- The protocol identifier (PID), as a **Byte** value
- The data coding scheme (DCS), as a **Byte** value.

These values override any default values set for the Card Manager.

A **CATTP** object carries:

- **BipPushParameter** object that allows you to specify values to be included in the SMS push sent for connection establishment.

  The **BipPushParameter** contains an **alphaId**: a message that is sent in the SMS push (user notification).

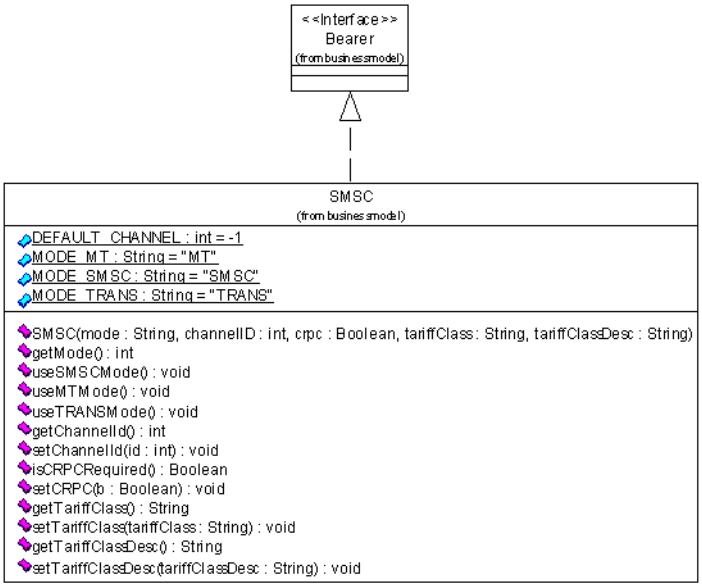
- **closeSession** flag allows you to manage the closing of the CAT-TP connection at the end of service execution. A “True” value (default) indicates closing the connection.

Bearer

A **CardHolder** can be contacted through a network using a **Bearer**. In practice, the bearer hides the network complexity, so the network concept is not represented as an object in the responsability chain.

The bearer can be set using the **CardHolder.setBearer()** method.

Figure 22 - The Bearer Interface



The **SMSC** bearer allows you to specify, for each service execution:

- The SMSC mode (MT, SMSC, or TRANS)
- The channel ID to use for SMS sending
- The CRPC: required (true) or not (false)
- The tariff class
- The tariff class description

Step Two: Build Invocation Parameters

The following invocation parameters are available:

- **Processing date**: the date to process the invocation
- **Expiration date** and **validity period**: an absolute date or a period (in seconds). The platform is authorized to execute the service until the expiration date is reached or the validity period is expired. If the expiration date is reached or the validity period expires before the service is completely executed, then the service execution state is set to EXPIRED.
- **Priority**: NORMAL, URGENT, or EMERGENCY.
- **Guarantee of execution**: Useful to prevent a crash or a voluntary shutdown of the platform.
- **Guarantee of delivery** of the result: the platform retains the service execution result until the client explicitly acknowledges reception of the result.

Additionally, information that does not influence invocation behaviour can be set as invocation parameters:

- **End use**r: identifier of the user that requested the service execution (may be different from the one that issued the invocation)
- **Transaction ID**: external identifier provided by the client for the global tracking of the service's execution in its information system

The **InvocationParameter** is built using the **InvocationFacility.buildInvocationParameter()** method.

Figure 23 - The InvocationFacility Interface

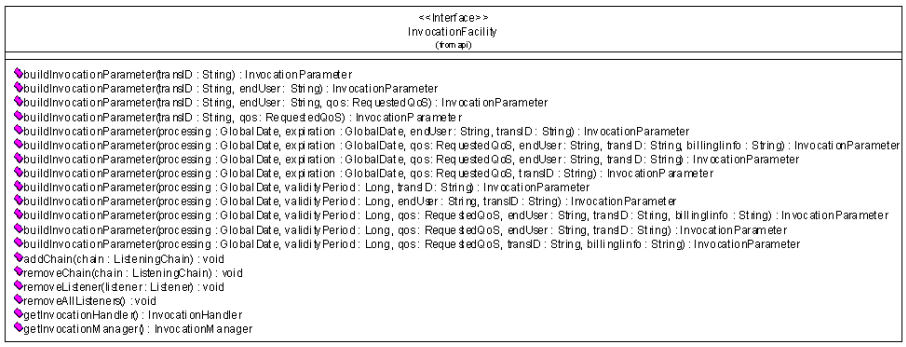


Figure 24 - Building the InvocationParameter

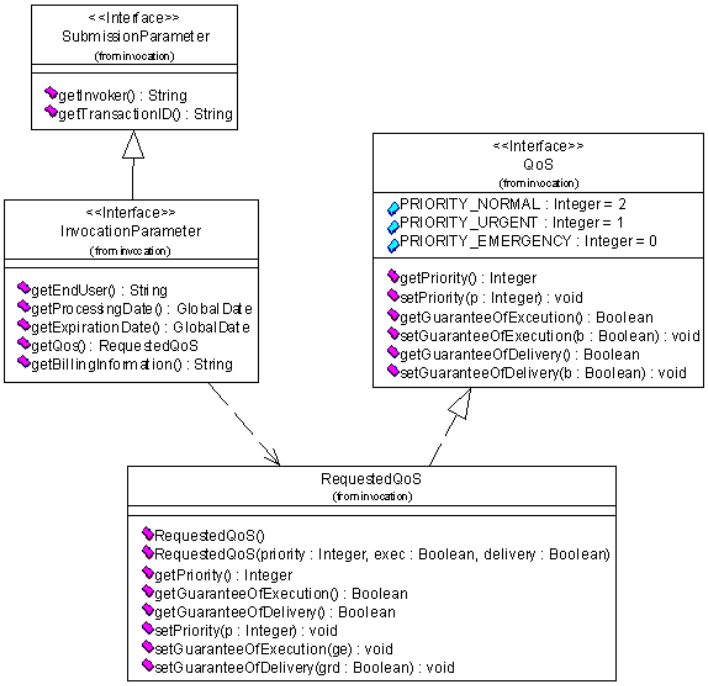
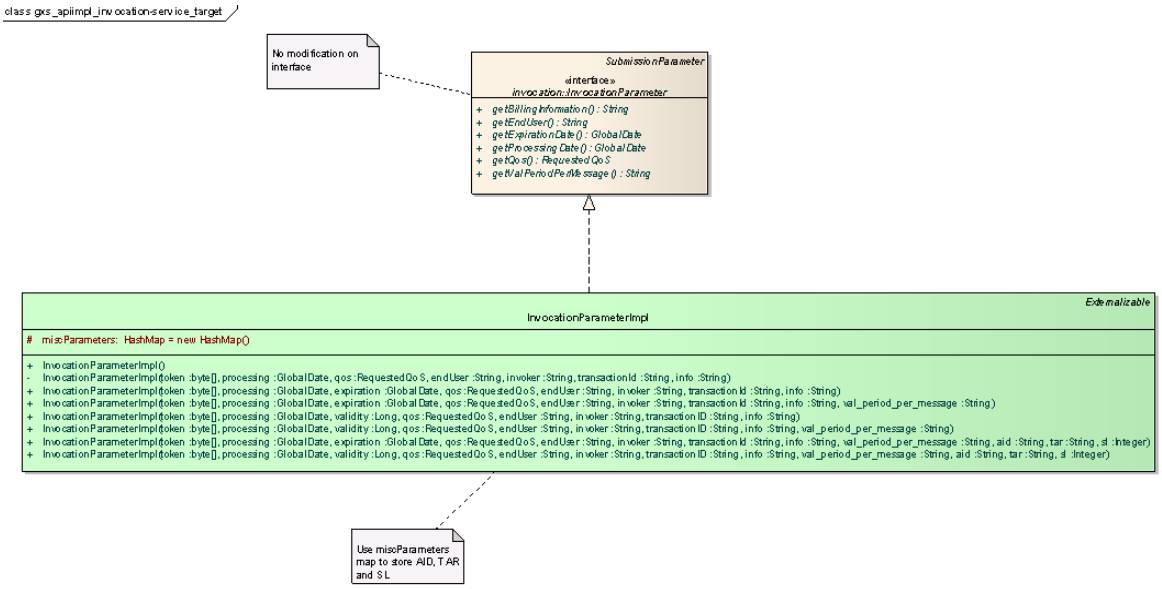


Figure 25 - The InvocationParameter Interface



The **InvocationParameter** interface allows you to select a target application using either the Application Identifier (AID) or the Toolkit Application Reference (TAR). You can also specify the Security Level to use for the targeted application. These parameters override the configuration set in the corresponding service implementation on the Card Manager instance.

Step Three: Request Service Execution

When you request service execution, you can choose to do it in a synchronous or asynchronous mode.

Two entry points are always available:

- **X**xx() : the service is called synchronously
- **PostX**xx() : the service is called asynchronously.

In synchronous mode, the client execution thread is suspended until the service execution ends in the platform. The result of the service execution is returned directly.

In asynchronous mode, the client thread is released as soon as the platform has accepted (or rejected) the service execution request. In this case, the client must implement an invocation event listener to be notified with the service execution result (see “Invocation Events” on page 42). This event is published as soon as service execution ends. (Any method with a “post” prefix indicates asynchronous mode).

As new services can be plugged in the platform, the generic entry points, **apply()** and **post()** methods on the **SmartCard** interface allow you to call them.

Service execution applies to the object for which the method is called: for example, if the method **MobileEquipment.sendText()** is called, then the service execution targets the mobile equipment.

Data are given in parameters and must be **Data** or a subtype of **Data**. The type requested for service execution is defined in each method for service invocation.

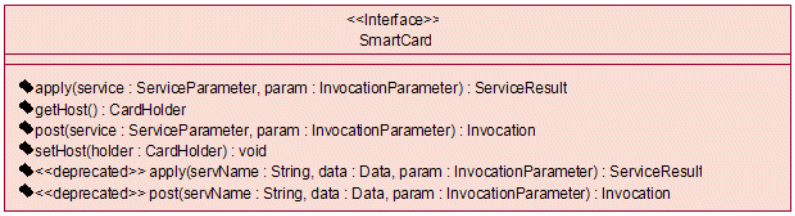
Generic **apply()** and **post()** methods only use the parent type **Data**.

All the standard types are defined in the package **com.gemplus.gxs.api.businessmodel.data**.

SmartCard

The **SmartCard** interface allows you to request any of the services available on the platform (both standard and custom services), in either asynchronous mode (**post()** method) or in synchronous mode (**apply()** method).

Figure 26 - The SmartCard Interface



SmartCard.post()

This method asynchronously requests service execution. This means that service execution ends after the method exits.

The method has the following parameters:

- **service**. Identifies the requested service and contains data for execution.
- **param**. The invocation parameters, such as quality of service, priority and so on

The method returns:

- **Invocation**. The invocation as it has been taken into account by the platform.

The method can raise the following exceptions:

- **DeniedAccessException** Raised if service access is denied.
- **InvalidInvocationException** Raised if invocation (data or invocation parameters) is invalid.
- **ExpiredAccountException** Raised if invoker account is expired.
- **BusyException** Raised if platform is busy and cannot accept the new invocation.

For example:

```
public Invocation post(ServiceParameter service,
        InvocationParameter param)
    throws DeniedAccessException,
         InvalidInvocationException,
         ExpiredAccountException,
         BusyException
```

SmartCard.apply()

This method synchronously requests service execution. Execution is synchronous. This means that service execution ends when the method exits.

The method has the following parameters:

- **service.** Identifies the requested service and contains data for execution.
- **param.** Invocation parameter such as quality of service, priority, and so on.

The method returns:

- **Invocation**. The invocation as it has been taken into account by the platform.

The method can raise the following exceptions:

- **DeniedAccessException.** Raised if service access is denied.

- **InvalidInvocationException.** Raised if invocation (data or invocation parameters) is invalid.
- **ExpiredAccountException.** Raised if invoker account is expired.
- **BusyException**. Raised if platform is busy and cannot accept the new invocation.

For example:

```
public ServiceResult apply(ServiceParameter service,
                  InvocationParameter param)
     throws DeniedAccessException,
          InvalidInvocationException,
          ExpiredAccountException,
          BusyException
```

MobileEquipment

MobileEquipment.postSendText()

The **postSendText()** method is used to asynchronously request “send text” service execution:

This method’s execution is asynchronous. This means that service execution will probably end after method exit.

Parameters are as follows:

- **service.** Identifies the requested service and contains data for execution.
- **param.** Invocation parameter such as quality of service, priority, and so on.

The method returns:

- **Invocation.** The invocation as it has been taken into account by the platform.

For example:

```
public Invocation postSendText(SendTextParameter service,
                          InvocationParameter param)
   throws DeniedAccessException,
        InvalidInvocationException,
        ExpiredAccountException,
        BusyException
```

MobileEquipment.sendText()

The **sendText()** method is used to synchronously request “send text” service execution. This execution is synchronous, this means that service execution ends after the method exits.

Parameters are as follows:

- **service**. Identifies the requested service and contains data for execution.
- **param**. Invocation parameter such as quality of service, priority, and so on.

The method returns:

- **Invocation**. The invocation as it has been taken into account by the platform.

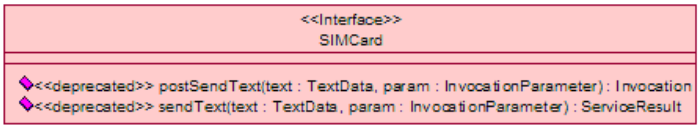
For example:

```
public ServiceResult sendText (ServiceParameter service,
                          InvocationParameter param)
     throws DeniedAccessException,
          InvalidInvocationException,
          ExpiredAccountException,
          BusyException
```

SIMCard, SIMCardViewer

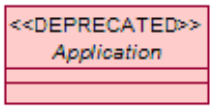
The **SIMCard** interface and its **sendText()** and **postSendText()** methods are deprecated in OTA Manager V5.1.

Figure 27 - The Deprecated SIMCard Interface

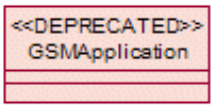


Application Interface and Sub-Interfaces

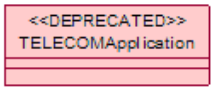
The **Application** interface is deprecated in OTA Manager V5.1.



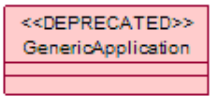
The **GSMApplication** interface is deprecated in OTA Manager V5.1:



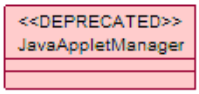
The **TelecomApplication** interface is deprecated in OTA Manager V5.1:



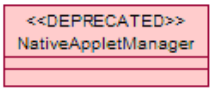
The **GenericApplication** interface is deprecated in OTA Manager V5.1:



The **JavaAppletManager** interface is deprecated in OTA Manager V5.1:



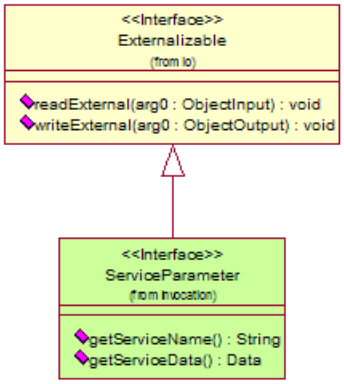
The **NativeAppletManager** interface is deprecated in OTA Manager V5.1:



ServiceParameter

The **com.gemplus.gxs.api.businessmodel.parameter.ServiceParameter** interface allows you to submit new invocations.

Figure 28 - The ServiceParameter Interface

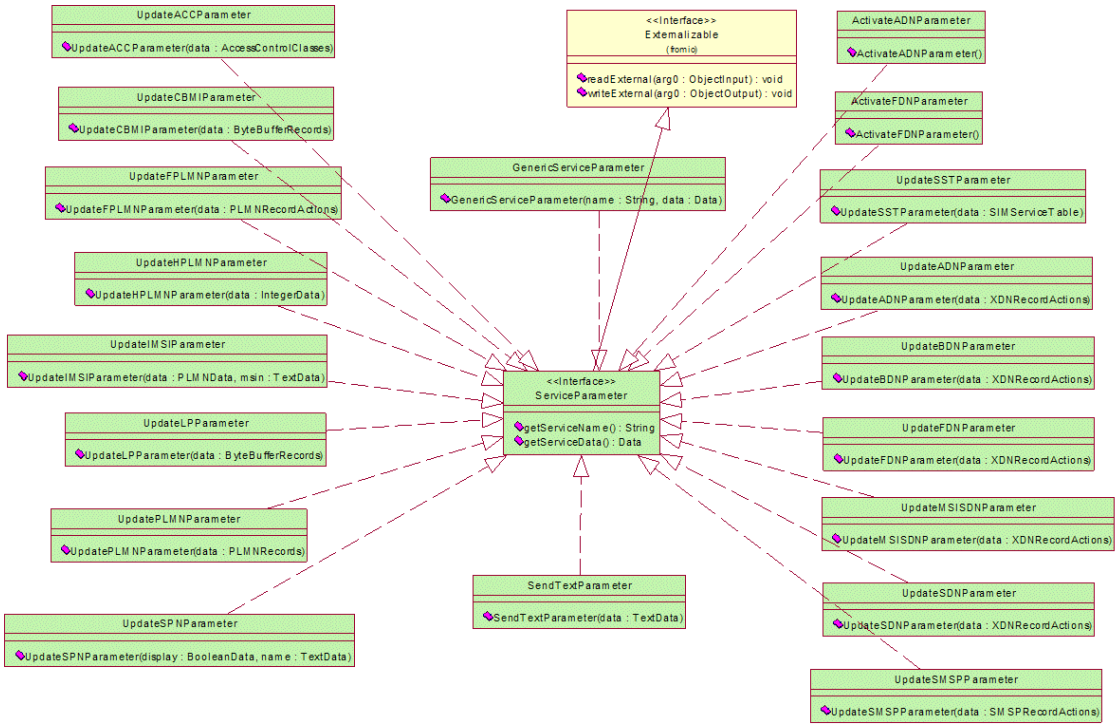


The **ServiceParameter.getServiceName()** method returns the name of the requested service. This name must match the name of the service declared on the server side (this is the same name as that returned by the corresponding **Service.getName()** method).

The **ServiceParameter.getServiceData()** method returns the data requested to be used for service execution.

GenericServiceParameter

Figure 29 - The GenericServiceParameter Interface

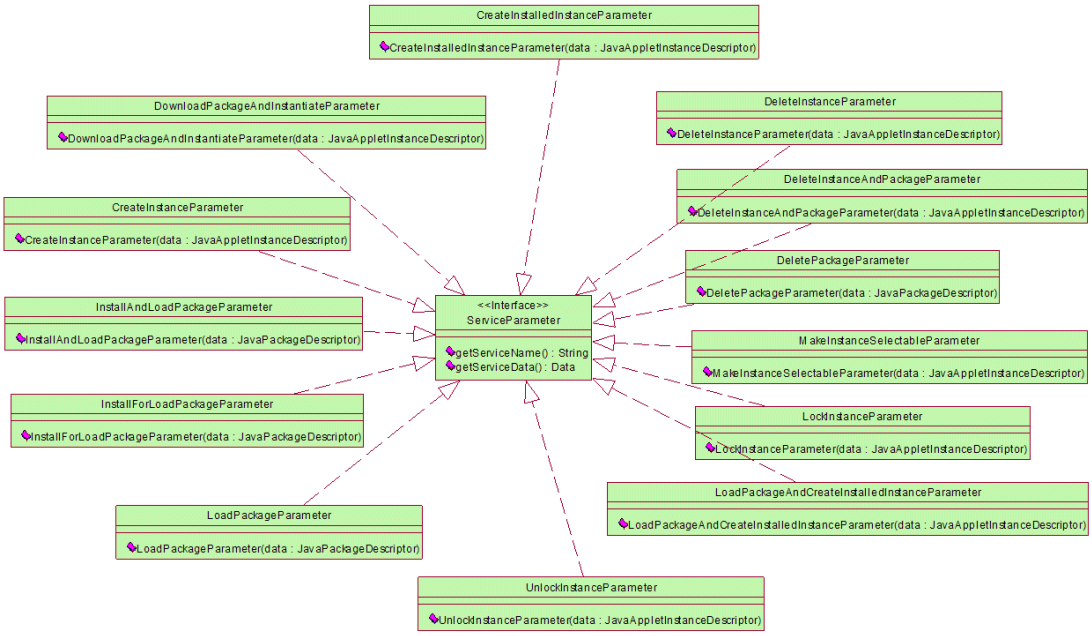


This concrete class allows you to request execution of any service by providing its name and data in the constructor.

In addition, the Core API defines concrete classes for each standard service:

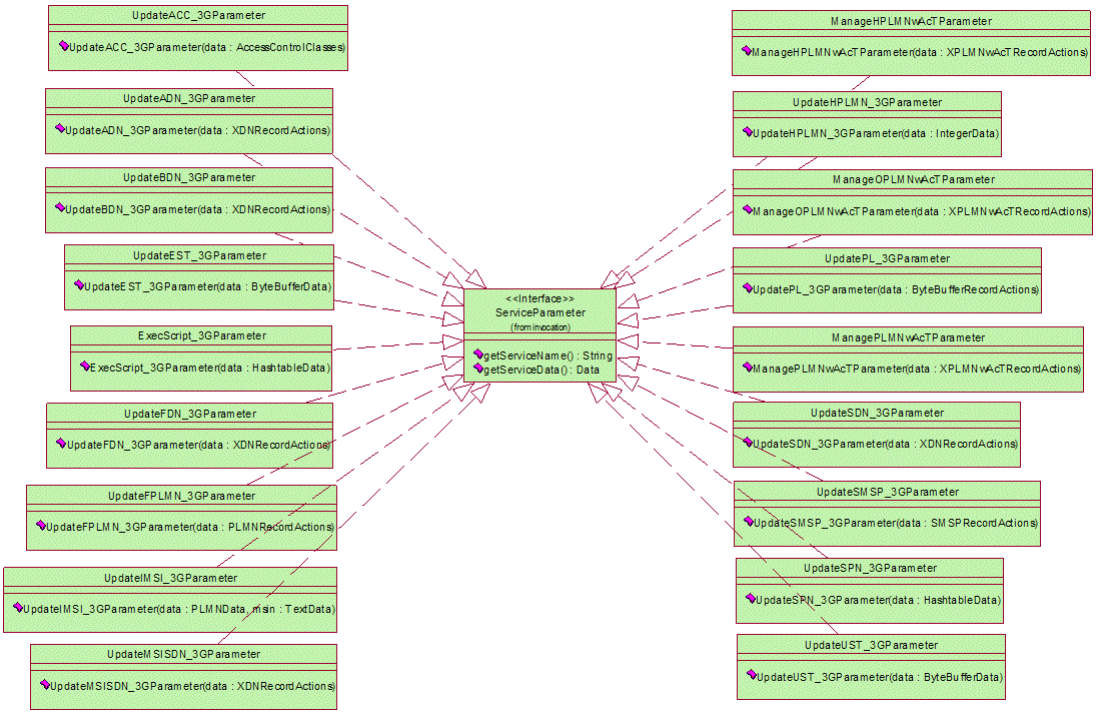
- **ActivateADNParameter**: concrete class allowing you to request ActivateADN service execution.
- **ActivateFDNParameter**: concrete class allowing you to request ActivateFDN service execution.
- **UpdateADNParameter**: concrete class allowing you to request UpdateADN service execution.
- **UpdateBDNParameter**: concrete class allowing you to request UpdateBDN service execution.
- **UpdateFDNParameter**: concrete class allowing you to request UpdateFDN service execution.
- **UpdateMSISDNParameter**: concrete class allowing you to request UpdateMSISDN service execution.
- **UpdateSDNParameter**: concrete class allowing you to request UpdateSDN service execution.
- **UpdateSMSPParameter**: concrete class allowing you to request UpdateSMSP service execution.
- **UpdateACCParameter**: concrete class allowing you to request UpdateACC service execution.
- **UpdateCBMIParameter**: concrete class allowing you to request UpdateCBMI service execution.
- **UpdateFPLMNParameter**: concrete class allowing you to request UpdateFPLMN service execution.
- **UpdateHPLMNParameter**: concrete class allowing you to request UpdateHPLMN service execution.
- **UpdateIMSIParameter**: concrete class allowing you to request UpdateIMSI service execution.
- **UpdateLPParameter**: concrete class allowing you to request UpdateLP service execution.
- **UpdateONSParameter**: concrete class allowing you to request UpdateONS service execution.
- **UpdatePLMNParameter**: concrete class allowing you to request UpdatePLMN service execution.
- **UpdateSPNParameter**: concrete class allowing you to request UpdateSPN service execution.
- **UpdateSSTParameter**: concrete class allowing you to request UpdateSST service execution.
- **SendTextParameter**: concrete class allowing you to request SendText service execution.

All these classes are located in the **com.gemplus.gxs.businessmodel.parameter** package.



- **CreateInstalledInstanceParameter**: concrete class allowing you to request CreateInstalledInstance service execution.
- **CreateInstanceParameter**: concrete class allowing you to request CreateInstance service execution.
- **DeleteInstanceParameter**: concrete class allowing you to request DeleteInstance service execution.
- **DeleteInstanceAndPackageParameter**: concrete class allowing you to request DeleteInstanceAndPackage service execution.
- **DeletePackageParameter**: concrete class allowing you to request DeletePackage service execution.
- **DownloadPackageAndInstantiateParameter**: concrete class allowing you to request DownloadPackageAndInstantiate service execution.
- **InstallAndLoadPackageParameter**: concrete class allowing you to request InstallAndLoadPackage service execution.
- **InstallForLoadPackageParameter**: concrete class allowing you to request InstallForLoadPackage service execution.
- **LoadPackageParameter**: concrete class allowing you to request LoadPackage service execution.
- **LoadPackageAndCreateInstalledInstanceParameter**: concrete class allowing you to request LoadPackageAndCreateInstalledInstance service execution.
- **LockInstanceParameter**: concrete class allowing you to request LockInstance service execution.
- **MakeInstanceSelectableParameter**: concrete class allowing you to request MakeInstanceSelectable service execution.

- **UnlockInstanceParameter**: concrete class allowing you to request UnlockInstance service execution.



- **UpdateACC_3GParameter**: concrete class allowing you to request UpdateACC_3G service execution.
- **UpdateADN_3GParameter**: concrete class allowing you to request UpdateADN_3G service execution.
- **UpdateBDN_3GParameter**: concrete class allowing you to request UpdateBDN_3G service execution.
- **UpdateEST_3GParameter**: concrete class allowing you to request UpdateEST_3G service execution.
- **ExecScript_3GParameter**: concrete class allowing you to request ExecScript_3G service execution.
- **UpdateFDN_3GParameter**: concrete class allowing you to request UpdateFDN_3G service execution.
- **UpdateFPLMN_3GParameter**: concrete class allowing you to request UpdateFPLMN_3G service execution.
- **UpdateIMSI_3GParameter**: concrete class allowing you to request UpdateIMSI_3G service execution.
- **UpdateMSISDN_3GParameter**: concrete class allowing you to request UpdateMSISDN_3G service execution.
- **ManageHPLMNwAcTParameter**: concrete class allowing you to request ManageHPLMNwAcT service execution.
- **UpdateHPLMN_3GParameter**: concrete class allowing you to request UpdateHPLMN_3G service execution.
- **ManageOPLMNwAcTParameter**: concrete class allowing you to request ManageOPLMNwAcT service execution.

- **UpdatePL_3GParameter**: concrete class allowing you to request UpdatePL_3G service execution.
- **ManagePLMNwAcTParameter**: concrete class allowing you to request ManagePLMNwAcT service execution.
- **UpdateSDN_3GParameter**: concrete class allowing you to request UpdateSDN_3GParameter service execution.
- **UpdateSMSP_3GParameter**: concrete class allowing you to request UpdateSMSP_3G service execution.
- **UpdateSPN_3GParameter**: concrete class allowing you to request UpdateSPN_3G service execution.
- **UpdateUST_3GParameter**: concrete class allowing you to request UpdateUST_3G service execution.

Service Result

A synchronous service call always returns a **ServiceResult** object. This carries all the information concerning the service execution result:

- The invocation ID it is issued from (**getInvocationID()**)
- The client transaction ID (**getTransactionID()**)
- Invoker name (**getInvoker()**)
- End user name (**getEnduser()**)
- Global ending status (**getEndingStatus()**): can be FAILED, SUCCEEDED, EXPIRED, DELETED, or SUPPRESSED
- Synthetic execution indicator (**isOK()**): the use of this method is deprecated; returns true

---

Note: Multi-destination service execution is deprecated, because the Campaign Manager has been introduced for this purpose. So, in the case of single destination service execution, the **isOK()** method, which is an artificial execution indicator, is redundant with the **getEndingStatus()** method. Moreover, **getResultSet()** and **getFailureResultSet()** methods always contain no more than one element.

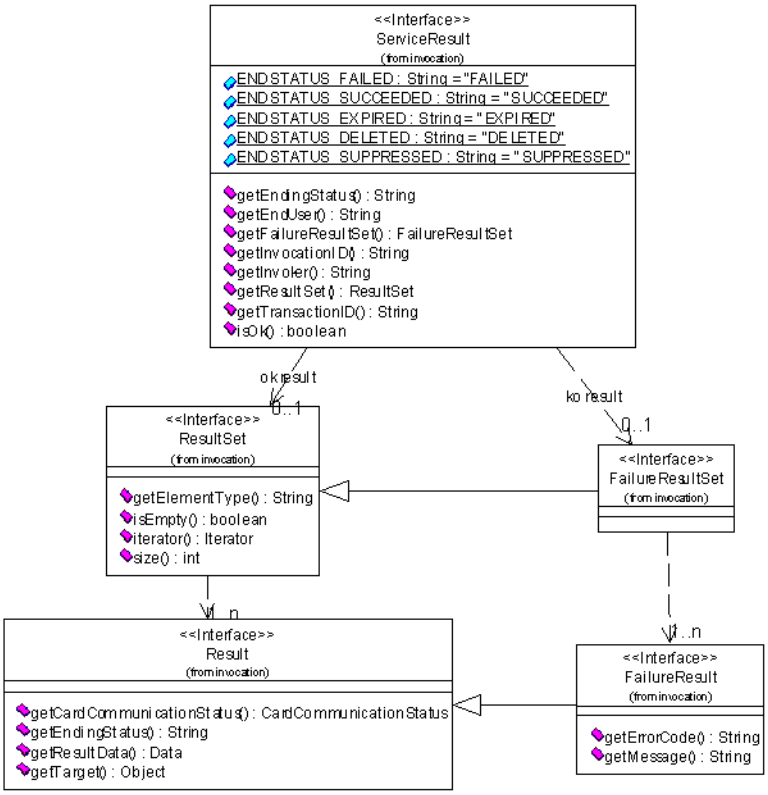
If service execution fails, **ResultSet** contains zero elements and **FailureResultSet** contains one element. If service execution succeeds, **ResultSet** contains one element and **FailureResultSet** contains zero elements.

The **Result** carries:

- The targeted element (**getTarget()**). (The method **ResultSet.getElementType()** allows you to discover the type of the element returned by the **getTarget()** method)
- Ending status for the target (**getEndingStatus()**): can be FAILED, SUCCEEDED, EXPIRED, DELETED, or SUPPRESSED
- Result data, if any (optional, depends on the service).

The **FailureResult** allows you to retrieve in addition the error code and error message.

Figure 30 - The Service Result



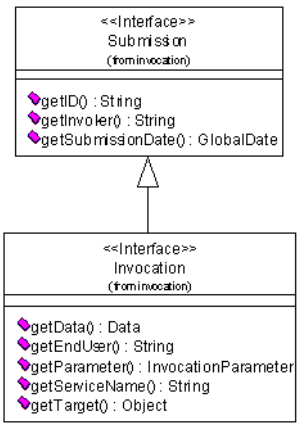
Invocation

An asynchronous service call always returns an **Invocation**. The **Invocation** is a type of acknowledgment from the platform. **Invocation** has an identifier that is used to retrieve the real service execution result. All information given at service execution request time (data, parameters, target, end user) is accessible from this **Invocation** object.

The actual service execution result is returned asynchronously by the platform using event notification (**InvocationEvent**). See “Invocation Events” on page 42 for information on how to get this result.

The **InvocationEvent.getResult()** method access the **ServiceResult**. Use the invocation ID on the **InvocationEvent** to match the expected result.

Figure 31 - The Invocation Interface



Code Examples

Performing a Service Invocation (ActivateADN) in Synchronous Mode

```
//How to perform a basic service invocation (ActivateADN service) in
synchronous mode?

ICCID iccid = new ICCID("2000000");
SIMCardImage simcard;
TELECOMApplication appli = null;
SIMCardViewer simCardViewer;
InvocationParameter param;
InvocationFacility invocationFacility;
ServiceResult result;

//Get the card (may exist or not)
//But must exist if service is "profile dependant"
simCardViewer =

cardManagerAccessPoint.getCardManagementFacility().getSIMCardViewer();
simcard = simCardViewer.getSIMCard(iccid);

//Build the application, and set the targetted card
appli = new TELECOMApplication();
appli.setHost(simcard);

//Set the client transactionID
invocationFacility = cardManagerAccessPoint.getInvocationFacility();
param = invocationFacility.buildInvocationParameter("TRANS_01");

//Invoke service ActivateADn
result = appli.activateADN(param);

System.out.println("EndingStatus : " + result.getEndingStatus());
```

Performing a Service Invocation (ActivateADN) in Asynchronous Mode

```
//How to perform a basic service invocation (ActivateADN service) in
asynchronous mode?

ICCID iccid = new ICCID("2000000");
SIMCardImage simcard;
TELECOMApplication appli = null;
SIMCardViewer simCardViewer;
InvocationParameter param;
InvocationFacility invocationFacility;
Invocation invocation;

//Get the card (may exist or not)
//But must exist if service is "profile dependant"
simCardViewer =
cardManagerAccessPoint.getCardManagementFacility().getSIMCardViewer();
simcard = simCardViewer.getSIMCard(iccid);

//Build the application, and set the targetted card
```

```
appli = new TELECOMApplication();
appli.setHost(simcard);

//Set the client transactionID
invocationFacility = cardManagerAccessPoint.getInvocationFacility();
param = invocationFacility.buildInvocationParameter("TRANS_01");

//Invoke service ActivateADn
invocation = appli.postActivateADN(param);

System.out.println("Service name : " + invocation.getServiceName());
```

Performing a Service Invocation (UpdateADN) in Synchronous Mode

```
//How to perform a basic service invocation (UpdateADN service) in
synchronous mode?

ICCID iccid = new ICCID("2000000");
SIMCardImage simcard;
TELECOMApplication appli = null;
SIMCardViewer simCardViewer;
InvocationParameter param;
InvocationFacility invocationFacility;
ServiceResultresult;

//Get the card (may exist or not)
//But must exist if service is "profile dependant"
simCardViewer =
cardManagerAccessPoint.getCardManagementFacility().getSIMCardViewer();
simcard = simCardViewer.getSIMCard(iccid);

//Build the application, and set the targetted card
appli = new TELECOMApplication();
appli.setHost(simcard);

//Set the client transactionID
invocationFacility = cardManagerAccessPoint.getInvocationFacility();
param = invocationFacility.buildInvocationParameter("TRANS_01");

//Build service parameters
XDNRecords records = new XDNRecords();
XDNRecord record1 = new XDNRecord(1, new TextData("adn1"),
        new PhoneNumber("060101010101"));
XDNRecord record2 = new XDNRecord(1, new TextData("adn1"),
        new PhoneNumber("060101010101"));
records.addRecord(record1);
records.addRecord(record2);

//Invoke service ActivateADn
result = appli.updateADN(records, param);

System.out.println("EndingStatus : " + result.getEndingStatus());
```

Setting Invocation Parameters and SMSC Options

```
ICCID iccid = new ICCID("2000000");
SIMCardImage simcard;
TELECOMApplication appli = null;
SIMCardViewer simCardViewer;
InvocationParameter param;
InvocationFacility invocationFacility;
ServiceResult result;
MobileEquipement equipment;

//Force SMSC channel parameters
// - channel driver to use
// - mode MT
SMSC smsc = new SMSC();
smsc.useMtMode();
String portStr =
utilities.getProperties().getProperty("rca.driver.identifier");
if (portStr==null)
   fail("Property rca.driver.identifier is not defined.");
int driverID;
try {
   driverID = Integer.parseInt(portStr);
   smsc.setChannelId(driverID);
} catch (NumberFormatException e) {
   fail("Property rca.driver.identifier contains a bad value:" +
portStr + " " + e.toString());
}
equipment = new MobileEquipement();
equipment.setBearer(smsc);

//Get the card (may exist or not)
//But must exist if service is "profile dependant"
simCardViewer =
cardManagerAccessPoint.getCardManagementFacility().getSIMCardViewer();
simcard = simCardViewer.getSIMCard(iccid);
simcard.setHost(equipment);

//Build the application, and set the targetted card
appli = new TELECOMApplication();
appli.setHost(simcard);

//Set the invocation parameters
//Guarantee of execution : false
//Guarantee of delivery : true
//Priority: URGENT
//Validity period 10s
//Immediate execution
invocationFacility = cardManagerAccessPoint.getInvocationFacility();
RequestedQoS qos = new RequestedQoS(QoS.PRIORITY_URGENT, new
Boolean(false), new Boolean(true));
param = invocationFacility.buildInvocationParameter(
          null, new Long(10000), qos, "TEST_AUTO", "Trans01");

//Invoke service ActivateADn
result = appli.activateADN(param);

System.out.println("EndingStatus : " + result.getEndingStatus());
```

Performing a CAT-TP Service Invocation (Activate ADN) in Synchronous Mode

```
ICCID iccid = new ICCID("2000000");
SIMCardImage simcard = null;
TELECOMApplication appli = null;
SIMCardViewer simCardViewer = null;
InvocationFacility invocationFacility = null;

//Gets the card
simCardViewer=cardManagerAccessPoint.getCardManagementFacility().getSI
MCardViewer();
 simcard = simCardViewer.getSIMCard(iccid);

//Create CATTP transport object
CATTP aCattp = new CATTP();
//Set values to be included in the SMS push sent for connection
establishment

//1. The connection will not be closed after service execution
aCattp.setCloseSessionAtServiceEnd(false);

//2. The user will be notify by message
//Create BipPusshParameter to set user notification
BipPushParameter abipPushParameter = new BipPushParameter();

//Set user notification (message to send in SMS push)
abipPushParameter.setAlphaIdentifier("Connection With CAT-TP protocol
OK");

//Set BipPusshParameter objet in CATTP
aCattp.setBipPushParameter(abipPushParameter);

//Create the card holder
MobileEquipement aMobile = new MobileEquipement();

//Add CAT-TP protocol
aMobile.setTransportProtocol(aCattp);

//Set the card holder in SIMCARD
simcard.setHost(aMobile);

//build application
appli = new TELECOMApplication();

//Set the targetted card in the application
appli.setHost(simcard);


//Build optional parameters
InvocationParameter
params=cardManagerAccessPoint.getInvocationFacility().buildInvocationP
arameter("ACTIVATE_ADN");

//Invoke service ActivateADN
ServiceResult aResult = appli.activateADN(params);
```

```
//Display the result
System.out.println("Ativate ADN service Result is: " +
aResult.getEndingStatus());
```

Performing the sendText Service on the Mobile

```
//How to perform sendText service on the mobile

InvocationFacility invocationFacility;
ServiceResult result;
MobileEquipement equipment;
InvocationParameter param;

//Build the mobile equipment and set the address
equipment = new MobileEquipement();
equipment.setAddress( new PhoneNumber("060101010101"));

//Set the client transactionID
invocationFacility = cardManagerAccessPoint.getInvocationFacility();
param = invocationFacility.buildInvocationParameter("TRANS_01");

//Invoke service SendText
result = equipment.sendText(new TextData("Hello world!"), param);

System.out.println("EndingStatus : " + result.getEndingStatus());
```

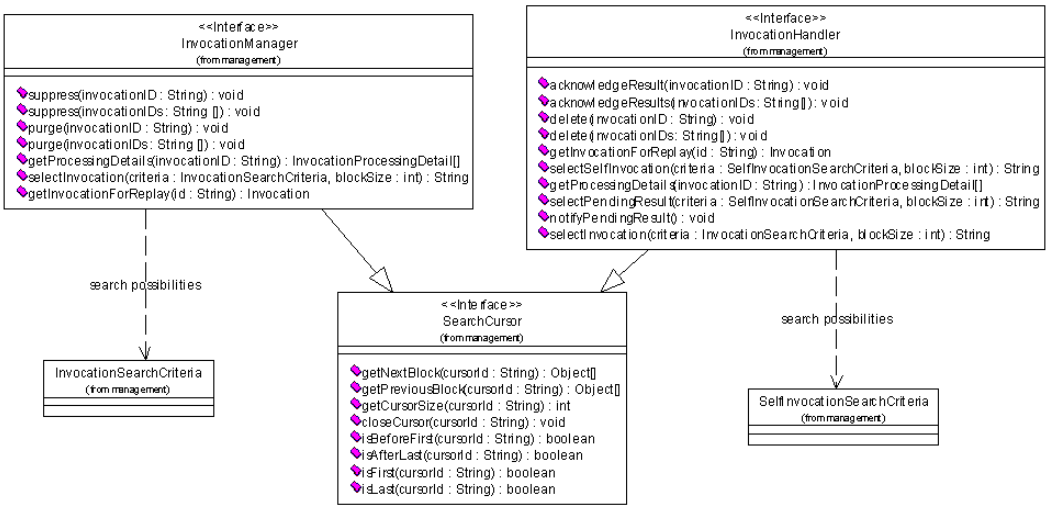
Performing a Service Invocation Using Generic Call

```
// How to perform a service invocation using generic call

ICCID iccid = new ICCID("2000000");
SIMCardImage simcard;
TELECOMApplication appli = null;
SIMCardViewer simCardViewer;
InvocationParameter param;
InvocationFacility invocationFacility;
ServiceResult result;


//Get the card (may exist or not)
//But must exist if service is "profile dependant"
simCardViewer =
cardManagerAccessPoint.getCardManagementFacility().getSIMCardViewer();
simcard = simCardViewer.getSIMCard(iccid);

//Build the application, and set the targetted card
appli = new TELECOMApplication();
appli.setHost(simcard);

//Set the invocation parameters
//Guarantee of execution : false
//Guarantee of delivery : true
//Priority: URGENT
//Validity period 10s
//Immediate execution
```

```
invocationFacility = cardManagerAccessPoint.getInvocationFacility();
RequestedQoS qos = new RequestedQoS(QoS.PRIORITY_URGENT, new
Boolean(false), new Boolean(true));
param = invocationFacility.buildInvocationParameter(
          null, new Long(10000), qos, "TEST_AUTO", "Trans01");

//Build service parameters
XDNRecords records = new XDNRecords();
XDNRecord record1 = new XDNRecord(1, new TextData("adn1"),
     new PhoneNumber("060101010101"));
XDNRecord record2 = new XDNRecord(1, new TextData("adn1"),
     new PhoneNumber("060101010101"));
records.addRecord(record1);
records.addRecord(record2);

//Invoke service ActivateADn
result = appli.apply("UpdateADN", records, param);

System.out.println("EndingStatus : " + result.getEndingStatus());
```

Monitoring a Request

Figure 32 - Monitoring a Request



The **InvocationHandler** interface is reserved to subscribers and customer care agents, while **InvocationManager** is reserved to administrator.

InvocationHandler allows you to perform operations concerning only its own invocations (there is no way to perform operations on another user's invocations). Searches are performed using the **SelfInvocationSearchCriteria** class.

Searching for Invocations

```
// Getting the invocation handler
try {
   handler = platform.getInvocationFacility().getInvocationHandler();
} catch (PlatformException e) {
   // This may happen if Card manager is not available
   System.out.println(e.getMessage());
```

```
   return;
}

// Build criteria for searching all invocation concerning UpdateADN
executed before "2003.12.10"
SimpleDateFormat formatter = new SimpleDateFormat("yyyy.MM.dd");
try {
   long date = formatter.parse("2003.12.10").getTime();
   dateLong = (new Long(date)).toString();
} catch (ParseException e) {
   // Something wrong
   System.out.println(e.getMessage());
   return;
}

try {
   criteria  = new
SelfInvocationSearchCriteria(SelfInvocationSearchCriteria
   .CRIT_SERVICE_NAME,SearchOperator.EQUAL, "UpdateADN");
   criteria.AND(new SelfInvocationSearchCriteria
                (SelfInvocationSearchCriteria.CRIT_END_EXECUTION_DATE,
                SearchOperator.LESSTHANEQUAL, dateLong));
} catch (MalformedSearchCriteriaException e) {
   // Something wrong
   System.out.println(e.getMessage());
   return;
}

try {
   // Do the select, this returns a cursor identifier
   // Use selectSelfInvocationForwardOnly to prevent OutOfMemoryError
      when many invocations are selected
   cursor = handler.selectSelfInvocationForwardOnly(criteria, 10);
} catch (InvalidCursorException e) {
   // May be impossible to create cursor
   System.out.println(e.getMessage());
   return;
} catch (NoMoreResourceException e) {
   // May be impossible to create cursor
   System.out.println(e.getMessage());
   return;
}

// Get all the results
try {
   while (!handler.isLast(cursor)) {
      Object[] records =  handler.getNextBlock(cursor);
      for (int i=0;i<records.length;i++) {
         record = (InvocationRecord) records[i];
         // Do something...
         System.out.println("Invocation identifier:" +
record.getInvocationID());
      }
   }
} catch (InvalidCursorException e) {
   // If cursor is invalid
```

```
   System.out.println(e.getMessage());
}
```

Obtaining Invocation Processing Details

```
// How to get invocation processing details?

InvocationHandler invocationHandler;
SelfInvocationSearchCriteria criteria;
String cursor;
InvocationRecord record;
InvocationProcessingDetail details;

//Search for invocation. Search can be done for example using the
invocation ID criteria
//Exception management are silently forgotten is this example
invocationHandler =
cardManagerAccessPoint.getInvocationFacility().getInvocationHandler();
criteria  = new
SelfInvocationSearchCriteria(SelfInvocationSearchCriteria.
   CRIT_INVOCATION_ID,SearchOperator.EQUAL, "3111015CC096AA");
// Use selectSelfInvocationForwardOnly to prevent OutOfMemoryError when
   many invocations are selected
cursor = invocationHandler.selectSelfInvocationForwardOnly(criteria,
10);

//Consider the first element,if found
Object[] records =  invocationHandler.getNextBlock(cursor);
if (records.length!=1) {
   System.out.println("Invocation not found");
   return;
}
record = (InvocationRecord) records[0];

//As it is a mono destination invocation, it only retirns one
InvocationProcessingDetail
details =
(invocationHandler.getProcessingDetails(record.getInvocationID()))[0];

System.out.println("Target:                " +
details.getTarget().toString());
System.out.println("ProcessingStatus:      " +
details.getProcessingStatus());
System.out.println("Error code (if any):   " + details.getErrorCode());
System.out.println("Error message(if any): " + details.getMessage());

System.out.println("Total to send: " +
details.getCardCommunicationStatus().getTotalMessagesNumber());
System.out.println("Sent:          " +
details.getCardCommunicationStatus().getSentMessagesNumber());
System.out.println("Delivered:    " +
details.getCardCommunicationStatus().getDeliveredMessagesNumber());
```

Acknowledging an Invocation

```
//How to acknowledge invocation ?
```

```
InvocationHandler invocationHandler;
SelfInvocationSearchCriteria criteria;
String cursor;
InvocationRecord record;

//Search for invocation. Search can be done for example using the
invocation ID criteria
//Exception management are silently forgotten is this example
invocationHandler =
cardManagerAccessPoint.getInvocationFacility().getInvocationHandler();
criteria  = new
SelfInvocationSearchCriteria(SelfInvocationSearchCriteria.CRIT_INVOCAT
ION_ID,
     SearchOperator.EQUAL, "3111015CC096AA");
// Use selectSelfInvocationForwardOnly to prevent OutOfMemoryError when
   many invocations are selected
cursor = invocationHandler.selectSelfInvocationForwardOnly(criteria,
10);


//Consider the first element,if found
Object[] records =  invocationHandler.getNextBlock(cursor);
if (records.length!=1) {
   System.out.println("Invocation not found");
   return;
}
record = (InvocationRecord) records[0];

//Acknowledge invocation
invocationHandler.acknowledgeResult(record.getInvocationID());
```

Provisioning and Data Management

The Card Manager allows you to manage all its related data though the Core API. This includes:

- Card vendors
- Card definitions
- Card profiles
- Service implementations
- Service declarations
- Card instances
- Card security

Each type of data is accessible though dedicated managers and is access-controlled. Access control is broken down into levels:

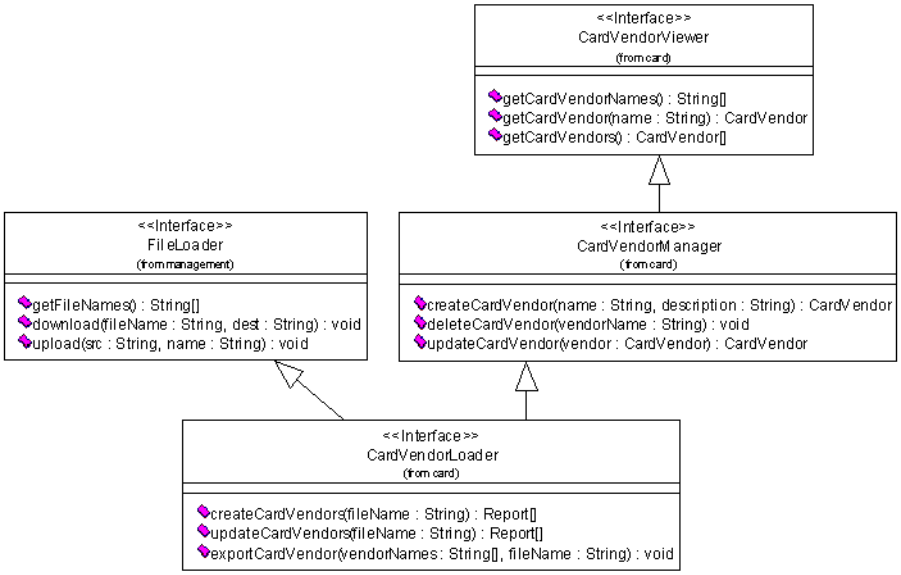
- View: access information in read mode only
- Edit: access information in read/write mode
- Provision: access information in read/write mode and allows batchloading of data.

These rights may be granted on a **Profile** (see “User Profile Management” on page 17).

"Figure 33" illustrates the organization of various managers for a card vendor.

Note: Managers for other data types have the same organization. Therefore, the description that follows for a Card Vendor is also valid for the other data types.

Figure 33 - Data Management



The interfaces are located in the **com.gemplus.gxs.api.management.card** package.

Viewer

The **CardVendorViewer** interface allows access to all card vendors defined in the platform:

- Get a list of names of all card vendors (**getCardVendorNames()**)
- Get a card vendor (**getCardVendor()**)
- Get all card vendors (**getCardVendors()**)

The **CardVendorViewer** interface is access-controlled by the category **CardViewingCategory**.

Manager

CardVendorManager interface inherits **CardVendorViewer**, so offers the same capabilities and also allows card vendor management:

- Create card vendor (**createCardVendor()**)
- Delete card vendor (**createCardVendor()**)
- Update card vendor (**updateCardVendor()**).

The **CardVendorManager** interface is access-controlled by the category **CardEditingCategory**.

Loader

The **CardVendorLoader** interface inherits **CardVendorManager**, so offers the same capabilities, and also allows card vendor batchload management:

- Batchload card vendors in create mode (**createCardVendors()**)
- Batchload card vendors in update mode (**updateCardVendors()**)
- Export card vendors (**exportCardVendors()**)

Files for batchload by must be XML formatted and conform to the **cardframework.dtd** (export is done using the same format).

A file may be batchloaded only if it is available in the platform repository. Each kind of data has its own repository location (for example, the card vendor repository is located in the $PLATFORM_HOME`/fileLoader/CardVendor` directory). Exported files are created in this repository.

For file management, **CardVendorLoader** also inherits from the **FileLoader** interface. This interface is dedicated to file management:

- Get all the file names available on the platform side repository for batchloading
- Upload a file from the client side to the platform repository
- Download a file from the platform repository to the client side.

Code Examples

The following examples demonstrate using a card definition, but can be easily adapted to use card vendors.

Creating a Card Definition

```
//How to create a card definition
CardDefinitionManager cardDefMgr;

//Access to the manager
try {
   cardDefMgr = cardManagementFacility.getCardDefinitionManager();
} catch (DeniedAccessException exp1) {
   System.out.println("Access denied for CardDefinitionManager:" +
exp1.toString());
   throw exp1;
}

try {
   //Create the new card definition. All information is given as
parameter of the method
   cardDefMgr.createCardDefinition("Name", "commercialName", "osName",
           "osVersion","chipMaker", "chipModel",
           10,  //eepromSize
           20,  //ramSize
           200, //romSize
           30  //freeSize
           );
   } catch (IllegalArgumentException exp) {
      System.out.println("Unable to create card definition cause:" +
exp.toString());
      throw exp;
   } catch (AlreadyExistsException exp) {
      System.out.println("Unable to create card definition cause:" +
exp.toString());
      throw exp;
   }
```

```
System.out.println("Card definition created.");
```

Updating a Card Definition

```
//How to update a card definition?
CardDefinitionManager cardDefMgr;
CardDefinitioncardDef;

//Access to the manager
try {
   cardDefMgr = cardManagementFacility.getCardDefinitionManager();
} catch (DeniedAccessException exp1) {
   System.out.println("Access denied for CardDefinitionManager:" +
exp1.toString());
   throw exp1;
}

//Get card definition
cardDef = cardDefMgr.getCardDefinition("ExampleName");

//Change a value
cardDef.setChipMaker("New chip maker");

//Update the card definition
try {
   cardDefMgr.updateCardDefinition(cardDef);
} catch (IllegalArgumentException exp) {
   System.out.println("Unable to update card definition cause:" +
exp.toString());
   throw exp;
} catch (NoMatchingObjectException exp) {
   System.out.println("Unable to update card definition cause:" +
exp.toString());
   throw exp;
}

System.out.println("Card definition updated.");
```

Deleting a Card Definition

```
//How to delete a card definition?
CardDefinitionManager cardDefMgr;
CardDefinitioncardDef;

//Access to the manager
try {
   cardDefMgr = cardManagementFacility.getCardDefinitionManager();
} catch (DeniedAccessException exp1) {
   System.out.println("Access denied for CardDefinitionManager:" +
exp1.toString());
   throw exp1;
}

//delete the card definition
try {
   cardDefMgr.deleteCardDefinition("ExampleName");
} catch (IllegalArgumentException exp) {
```

```
   System.out.println("Unable to update card definition cause:" +
exp.toString());
   throw exp;
} catch (NoMatchingObjectException exp) {
   System.out.println("Unable to update card definition cause:" +
exp.toString());
   throw exp;
}

System.out.println("Card definition deleted.");
```

Batchloading a Card Definition

```
//How to batch load a card definition?
CardDefinitionLoader cardDefLoader;
String     fileName = "/gcota/example/ExampleCardDef.xml";
Report     reports;

//Access to the loader
try {
   cardDefLoader = cardManagementFacility.getCardDefinitionLoader();
} catch (DeniedAccessException exp1) {
   System.out.println("Access denied for CardDefinitionloader:" +
exp1.toString());
   throw exp1;
}

//Upload file on server side. For the example the file is opened as
//ressource (in the classpath)
URLurl;
Filefile;
url = getClass().getResource(fileName);
file = new File(url.getFile());
cardDefLoader.upload(file.getPath(), file.getName());

// Batchload the card Definition
reports = cardDefLoader.createCardDefinitions(file.getName());

// Returns one report in case of error as we create one card Definition
if (reports.length>0) {
   System.out.println("Error in batchloed of card definition");
   System.out.println("Report code:" + reports[0].getErrorCode());
   System.out.println("Report message:" +
reports[0].getErrorMessage());
} else {
   System.out.println("Batchloed of card definition succeeded");
}
```

SIM Card Provisioning

The following description provides code examples for performing unitary SIM card provisioning.

The **SIMCardManager** interface allows you to create a new SIM card and update, activate, or deactivate an existing SIM card. To do these tasks, your user profile must include the **CardEditingCategory**.

SIM card creation is performed directly, using a single method call to the interface, whereas updating a SIM card involves two steps:

1 Access the card, using a **get**XXX() method, or a search with specified criteria

2 Set new values on the card then request the update method on the interface.

Activation and deactivation of SIM cards is also performed with a unique call to the interface, identifying the card using the ICCID, MSISDN or IMSI (remember that the MSISDN and IMSI may not be set on the card).

Creating a SIM Card

```
//How to create a SIM card?
SIMCardManager cardMgr;

//Access to the manager
try {
   cardMgr = cardManagementFacility.getSIMCardManager();
} catch (DeniedAccessException exp1) {
   System.out.println("Access denied for SIMCardManager:" +
exp1.toString());
   throw exp1;
}

//Create an inactive card without specifying MSISDN or IMSI
try {
   cardMgr.createSIMCard(
      new ICCID("0601010101"),
      "GX3G_v2.2_128K",
      SIMCardImage.STATE_INACTIVE,
      null, //MSISDN is null
      null);//IMSI is null
} catch (AlreadyExistsException exp) {
   System.out.println("Unable to create SIM card cause:" +
exp.toString());
   throw exp;
} catch (InvalidStateException exp) {
   System.out.println("Unable to create SIM card cause:" +
exp.toString());
   throw exp;
} catch (NoMatchingObjectException exp) {
   System.out.println("Unable to create SIM card cause:" +
exp.toString());
   throw exp;
}

Provision an MSISDN on an existing SIM card
//How to provision an MSISDN on an existing SIM card?
SIMCardManager cardMgr;
SIMCardImagesimCard;

//Access to the manager
try {
   cardMgr = cardManagementFacility.getSIMCardManager();
} catch (DeniedAccessException exp1) {
   System.out.println("Access denied for SIMCardManager:" +
exp1.toString());
```

```
   throw exp1;
}

//Get the SIMCardImage
simCard = cardMgr.getSIMCard(new ICCID("111"));

//Set the MSISDN
simCard.setMSISDN(new MSISDN("0601010101"));

try {
   //Commit the update on the card
   cardMgr.updateSIMCard(simCard);
} catch (InvalidStateException exp) {
   System.out.println("Unable to create SIM card cause:" +
exp.toString());
   throw exp;
} catch (NoMatchingObjectException exp) {
   System.out.println("Unable to create SIM card cause:" +
exp.toString());
   throw exp;
}
```

Activating an Existing SIM Card

```
//How to activate an existing SIM card?
SIMCardManager cardMgr;

//Access to the manager
try {
   cardMgr = cardManagementFacility.getSIMCardManager();
} catch (DeniedAccessException exp1) {
   System.out.println("Access denied for SIMCardManager:" +
exp1.toString());
   throw exp1;
}

try {
//Activate the SIMCard identified by its ICCID (identification using
MSISDN is also possible)
//Deactivation is performed in the same way
   cardMgr.activateSIMCard(new ICCID("111"));
} catch (NoMatchingObjectException exp) {
   System.out.println("Unable to create SIM card cause:" +
exp.toString());
   throw exp;
}
```

Deleting a SIM Card

```
//How to delete a SIM card?
SIMCardManager cardMgr;

//Access to the manager
try {
   cardMgr = cardManagementFacility.getSIMCardManager();
} catch (DeniedAccessException exp1) {
```

```
   System.out.println("Access denied for SIMCardManager:" +
exp1.toString());
   throw exp1;
}

//Delete the SIM Card
try {
   cardMgr.deleteSIMCard( new ICCID("111"));
} catch (NoMatchingObjectException exp) {
   System.out.println("Unable to create SIM card cause:" +
exp.toString());
   throw exp;
}
```

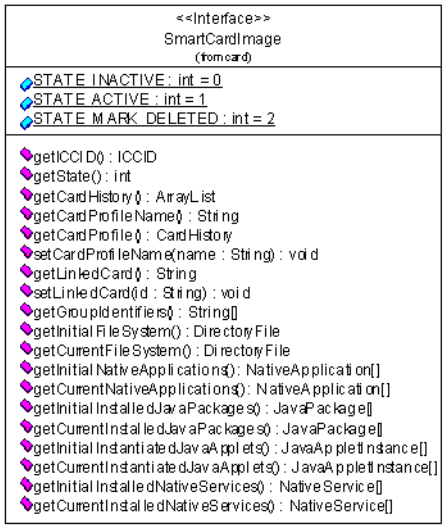
Card Content Representation and Access

The Card Manager is able to store and update though service execution the content of any card existing in the platform. Card content representation includes:

- The file system with all file types: Dedicated files (also called Directory files in OTA Manager), transparent files, linear files and cyclic files.
- Native applications and services
- Java packages, applets and applet instances

Card content is accessible through the Core API, but in read mode only. Card content can only be updated by service execution.

Figure 34 - The SmartCardImage Interface



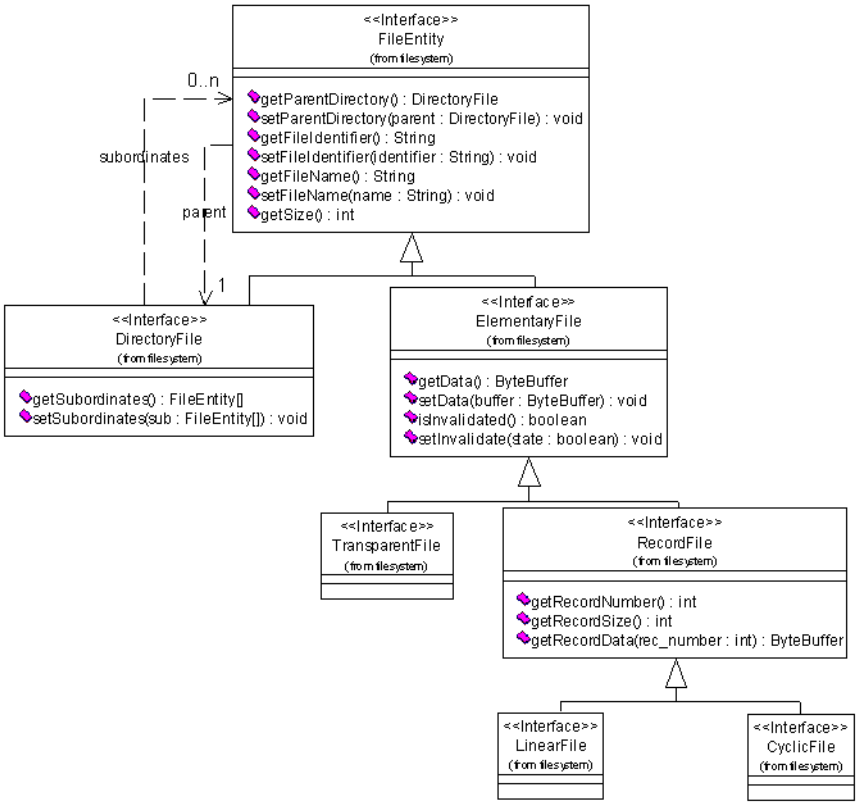
Card content can be accessed through the **SmartCardImage** interface. This provides two types of view:

- The initial card content: the content as issued after personalization. This representation is accessed using **getInitial**XXX() methods.
- The current card content: the content with all the updates that have been performed by all the executed services. This is therefore a representation of the card as it is in the field. This representation is accessed using **getCurrent**XXX() methods.

File System Representation

The **getCurrentFileSystem()** and **getInitialFileSystem()** methods provide access to the master file (MF 3F00) of the current file system representation and the initial file system representation, respectively. By starting from the master file, it is possible to access any file in the file system.

Figure 35 - File System Representation



All these interfaces are defined in the **com.gemplus.components.cardservices.filesystem** package.

Only **DirectoryFile**, **TransparentFile**, **LinearFile** and **CyclicFile** are present in a file system and have a concrete implementation. **FileEntity**, **ElementaryFile** and **RecordFile** are abstract notions used to regroup common information.

A **FileEntity** is the most abstract representation of an entity in the file system. It includes the following information:

- Parent directory: always set, except for the master file, which has no parent.
- File identifier. Example for ADN file: “6F3A”
- File name. Example: “ADN”
- Size. For elementary files, size corresponds to that of the file. For directory files, it is the sum of the size of all the sub-file entities.

An **ElementaryFile** object includes the following information:

- Data. A byte buffer compliant with the format defined in the 3GPP TS 31.101, 3GPP TS 31.102 and TS 102.221 specifications
- State: validated or invalidated.

A **RecordFile** object represents any file that is structured with records (when transparent file content is free). It includes the following information:

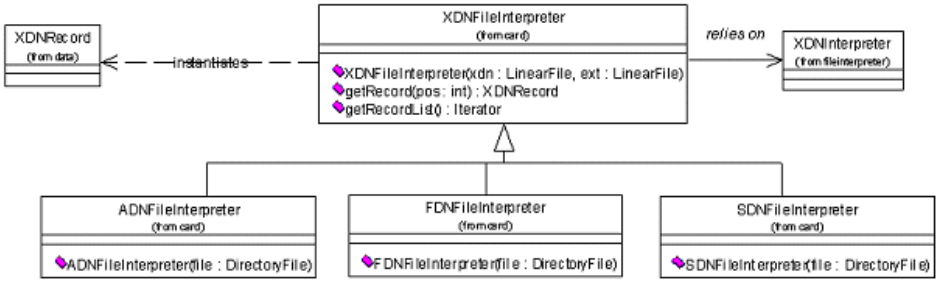
- Number of records
- Size of record. The size of a record file can be calculated using the formula (Number of records) * (Size of record)
- Data of each record.

```
//How to read ADN content?

SIMCardViewer cardViewer;
SIMCardImage  simcard;
LinearFileadnFile = null;

//Access to the manager
try {
   cardViewer = cardManagementFacility.getSIMCardViewer();
} catch (DeniedAccessException exp1) {
   System.out.println("Access denied for SIMCardViewe:" +
exp1.toString());
   throw exp1;
}

//Get the card. At this moment the existence is not verified
simcard = cardViewer.getSIMCard(new ICCID("2000000"));

//Get the master file subordinates
FileEntity[] filesMF =
simcard.getCurrentFileSystem().getSubordinates();

//Search for DF with identifier=7F10
DirectoryFile dfTelecom = null;
for (int i = 0; i < filesMF.length; i++) {
   if (filesMF[i].getFileIdentifier().equals("7F10")) {
      dfTelecom = (DirectoryFile)filesMF[i];
   }
}
if (dfTelecom==null) {
   System.out.println("Unable to find TELECOM DF");
   return;
}

//Search for ADN with identifier=6F3A
FileEntity[] filesTelecom = dfTelecom.getSubordinates();
for (int j= 0; j < filesTelecom.length; j++) {
   if (filesTelecom[j].getFileIdentifier().equals("6F3A")) {
      adnFile = (LinearFile) filesTelecom[j];
   }
}
if (adnFile==null) {
   System.out.println("Unable to find ADN file");
   return;
}

System.out.println("File identifier:   " +
adnFile.getFileIdentifier());
System.out.println("File name:         " + adnFile.getFileName());
```

```
System.out.println("File size:          " + adnFile.getSize());
System.out.println("Number of records: " + adnFile.getRecordNumber());
System.out.println("Record size:       " + adnFile.getRecordSize());
System.out.println("Data:              " + adnFile.getData());
```

It is possible to obtain an interpreted view of the ADN content by using the **com.gemplus.gxs.api.businessmodel.card.ADNFileInterpreter**. This class must be instantiated using the MF and retrieves the ADN file 6F3A in the file hierarchy given at instantiation time. This avoids the client application having to navigate the file hierarchy.

Figure 36 - File Interpreters



Other interpreters exist for the files FDN, SDN, PLMN, FPLMN, PLMNwACT, OPLMNwACT and HPLMNwACT.

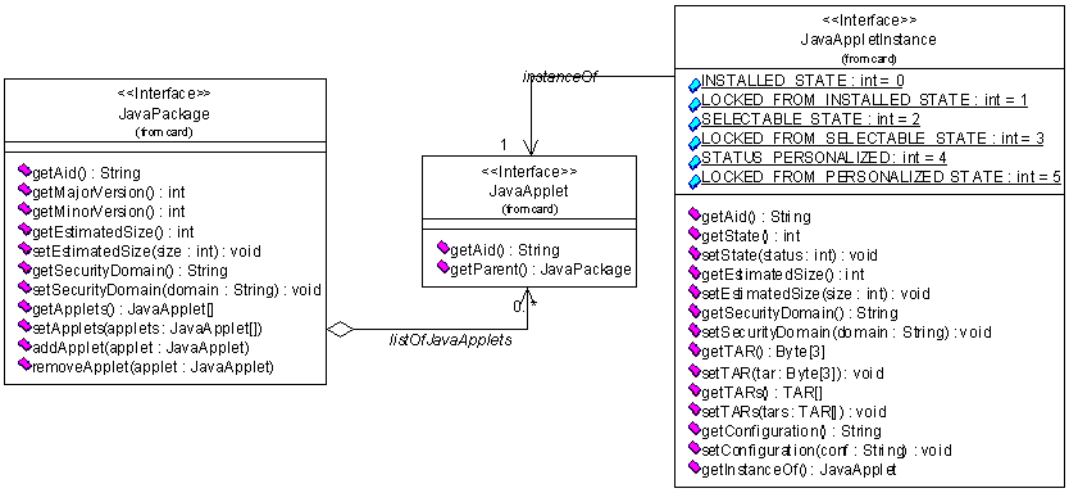
```
//Use interpreter to get directly access to ADN
ADNFileInterpreter adnInterpreter = new
ADNFileInterpreter(simcard.getCurrentFileSystem());
Iterator iterator = adnInterpreter.getRecordList();
while(iterator.hasNext()) {
   XDNRecord record = (XDNRecord) iterator.next();
   System.out.println("Record "+ record.getRecordNumber() + ": " +
record.toString());
}
```

Java Content Representation

The **getCurrentInstalledJavaPackages()** and **getInitialInstalledJavaPackages()** methods give access to the list of installed Java packages of the current representation and the initial list of installed Java packages, respectively.

The **getCurrentInstantiatedJavaApplets()** and **getInitialInstantiatedJavaApplets()** methods give access to the list of instantiated Java applets in the current representation and the initial list of instantiated Java applets, respectively.

Figure 37 - Java Application Representation



JavaPackage includes the following information:

- The AID, major and minor version
- Estimated size occupied on the card
- Security domain AID (if any)
- List of applets it contains.

JavaAppletInstance includes the following information:

- AID
- State
- Estimated size occupied on the card
- Security domain AID: link to the security information to use for the service execution
- Name of the configuration file used to configure the applet on the card
- A reference to the applet it is instantiated from.

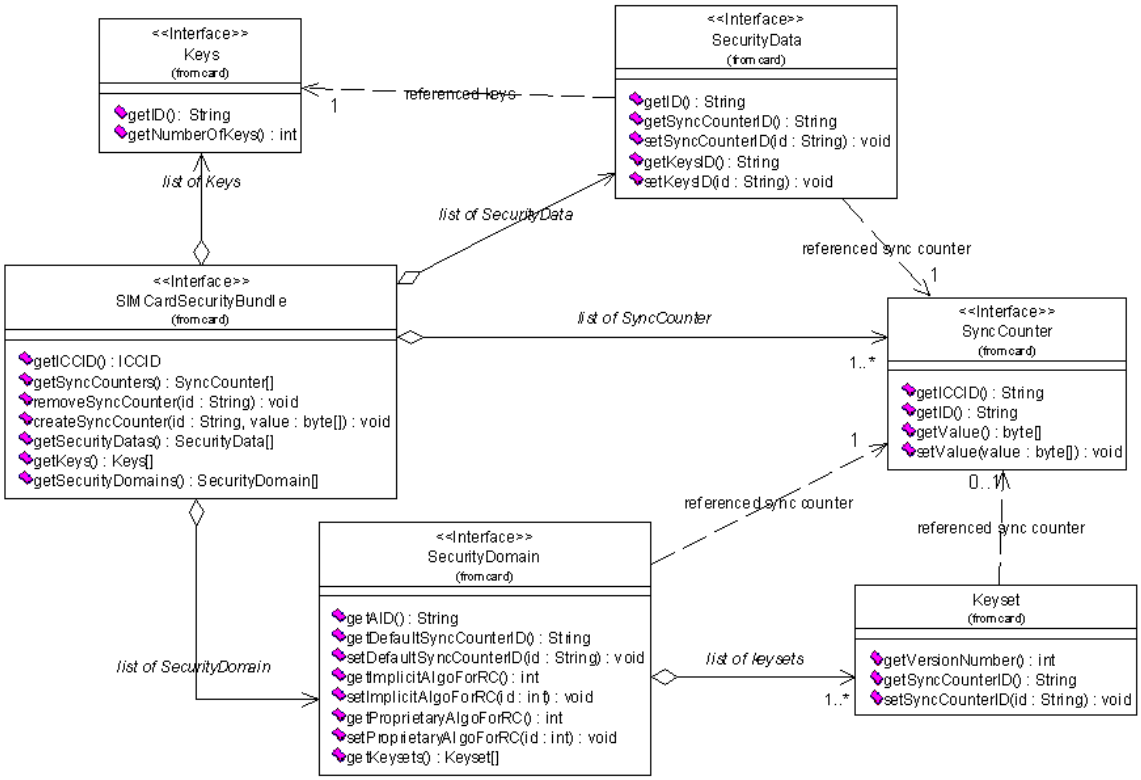
```
//How to read java content?
SIMCardViewer cardViewer;
SIMCardImage  simcard;

//Access to the manager
try {
   cardViewer = cardManagementFacility.getSIMCardViewer();
} catch (DeniedAccessException exp1) {
   System.out.println("Access denied for SIMCardViewe:" +
exp1.toString());
   throw exp1;
}

//Get the card. At this moment the existence is not verified
simcard = cardViewer.getSIMCard(new ICCID("2000000"));

//Get the package list
JavaPackage[] packages = simcard.getCurrentInstalledJavaPackages();
for (int i=0; i<packages.length;i++) {
   System.out.println("Package AID: "+ packages[i].getAid());
```

```
   System.out.println("Version:     "+ packages[i].getMajorVersion() +
"." + packages[i].getMinorVersion());
   System.out.println();
}

//Get the instance list
JavaAppletInstance[] instances =
simcard.getCurrentInstanciatedJavaApplets();
for (int i=0; i<instances.length;i++) {
   System.out.println("Instance AID:        " + instances[i].getAid());
   System.out.println("Status:              " +
instances[i].getStatus());
   System.out.println("Security domain AID: " +
instances[i].getSecurityDomain());
   System.out.println();
}
```

Card Security Information

The Core API allows card security information to be managed. For more information on security management, refer to the *OTA Manager V5.1 Administration Guide*.

Figure 38 - Card Security Information



The **SIMcardSecurityBundle** is the root object from which to obtain all card security information. The **SIMcardSecurityBundle** can be obtained using the **SIMCardSecurityViewer** interface, updated using **SIMCardSecurityManager,** and batchloaded using **SIMCardSecurityLoader** (see “Provisioning and Data Management” on page 70 for more information).

Security information includes:

- Security data and keys. This was the security model prior to GemXplore Suite 2.0 SP4, and is available to ensure backward compatibility. It allows the modeling of native security (keys contained in files) and GSM 03.48 security emulation (but without the ability to set one synchronization counter per key set and with limitations on redundancy check (RC) parameters).
- Security domain and key set. This is the security model representation conforming to 3GPP 31.115 and TS 102.225. It allows you, for example, to set one synchronization counter for each key set.
- Synchronization counters.

The **SecurityData** interface gives access to:

- Identifier: may be a file identifier for native security or a security domain AID for GSM 03.48 emulation.
- Synchronization counter ID: identifier of the synchronization counter associated with the security data.
- Keys ID: identifier of the keys associated with the security data.

Keys represent a logical group of keys (not to be confused with the GSM 03.48 key set concept, which is represented with the **Keyset** interface). The interface gives access to:

- Identifier
- The number of keys in the group.

Note: It is impossible to access either key values or the associated algorithm.

A **SyncCounter** is defined with:

- An identifier: allows other objects to reference it.
- The card it belongs to
- Its value. The value can be changed.

A **SecurityDomain** is defined with:

- An AID
- A reference to a default synchronization counter (defined on the card)
- Implicit algorithm for RC computation
- Proprietary algorithm for RC computation
- Key sets

A **Keyset** is defined with:

- A version number
- A reference to a synchronization counter (defined on the card). This reference is optional. If not set, service execution using this key set and requiring a synchronization counter update uses the **SecurityDomain** object's default synchronization counter.

Note: It is impossible to access either the key value or the associated algorithm.

Searching for Card Security Information

```
//How to search card security information?

SIMCardSecurityViewer cardSecurityViewer;
```

```
String cursor;

//Access to the manager
try {
   cardSecurityViewer =
cardManagementFacility.getSIMCardSecurityViewer();
} catch (DeniedAccessException exp1) {
   System.out.println("Access denied for SIMCardViewe:" +
exp1.toString());
   throw exp1;
}

//Build criteria for searching all the cards with
2000000<=ICCID<=2000002
CardSecuritySearchCriteria criteria1 = new CardSecuritySearchCriteria(
     CardSecuritySearchCriteria.CRIT_ICCID,
     CardSecuritySearchCriteria.GREATERTHANEQUAL,
     "2000000");
CardSecuritySearchCriteria criteria2 = new CardSecuritySearchCriteria(
     CardSecuritySearchCriteria.CRIT_ICCID,
     CardSecuritySearchCriteria.LESSTHANEQUAL,
     "2000002");
criteria1.AND(criteria2);

//Search
cursor = cardSecurityViewer.selectSIMCardSecurityBundle(criteria1, 10);

//Display result for all result.
while (!cardSecurityViewer.isLast(cursor)) {
   Object[] list = cardSecurityViewer.getNextBlock(cursor);

   for (int i=0;i<list.length;i++) {
      SIMCardSecurityBundle  securityBundle =
         (SIMCardSecurityBundle) list[i];
      System.out.println("Security for ICCID: " +
         securityBundle.getICCID());
   }
}
```

Creating a New Synchronization Counter In An Existing Security Bundle

```
//How to create a new sync counter in an existing security bundle?

SIMCardSecurityManager cardSecurityMgr;
SIMCardSecurityBundlecardBundle;

//Access to the manager
try {
   cardSecurityMgr =
cardManagementFacility.getSIMCardSecurityManager();
} catch (DeniedAccessException exp1) {
   System.out.println("Access denied for SIMCardViewe:" +
exp1.toString());
   throw exp1;
}
```

```
//Get the card security bundle
cardBundle = cardSecurityMgr.getSIMCardSecurityBundle( new
ICCID("2000000") );

//Create the sync counter
try {
   cardBundle.createSyncCounter("SYNCTEST", new byte[] {0x00, 0x01,
0x02, 0x03, 0x04});
} catch(IllegalArgumentException e) {
   System.out.println(e);
   return;
}

//Commit the creation
cardSecurityMgr.updateSIMCardSecurityBundle(cardBundle);
System.out.println("New sync counter created");
```

Updating a Synchronization Counter In An Existing Security Bundle

```
//How to update a sync counter in an existing security bundle?

SIMCardSecurityManager cardSecurityMgr;
SIMCardSecurityBundlecardBundle;

//Access to the manager
try {
   cardSecurityMgr =
cardManagementFacility.getSIMCardSecurityManager();
} catch (DeniedAccessException exp1) {
   System.out.println("Access denied for SIMCardViewe:" +
exp1.toString());
   throw exp1;
}

//Get the card security bundle and sync counters
cardBundle = cardSecurityMgr.getSIMCardSecurityBundle( new
ICCID("2000000") );
SyncCounter[] syncs = cardBundle.getSyncCounters();

//Search the rigth sync
SyncCounter sync = null;
for (int i=0;i<syncs.length;i++) {
   if (syncs[i].getID().equals("SYNCTEST") ) {
      sync = syncs[i];
   }
}
//Set the new value (a sync value is always 5 bytes long)
if (sync != null) {
   sync.setValueAsString("0101010101");
} else {
   System.out.println("Sync counter not found!");
}

//Commit the update
try {
```

```
   cardSecurityMgr.updateSIMCardSecurityBundle(cardBundle);
} catch(IllegalArgumentException e) {
   System.out.println(e);
   return;
}

System.out.println("Sync counter updated");
```

Deleting a Synchronization Counter in an Existing Security Bundle

```
//How to delete a sync counter in an existing security bundle?

SIMCardSecurityManager cardSecurityMgr;
SIMCardSecurityBundlecardBundle;

//Access to the manager
try {
   cardSecurityMgr =
cardManagementFacility.getSIMCardSecurityManager();
} catch (DeniedAccessException exp1) {
   System.out.println("Access denied for SIMCardViewe:" +
exp1.toString());
   throw exp1;
}

//Get the card security bundle and sync counters
cardBundle =
   cardSecurityMgr.getSIMCardSecurityBundle( new ICCID("2000000") );

cardBundle.removeSyncCounter("SYNCTEST");

//Commit the delete
try {
   cardSecurityMgr.updateSIMCardSecurityBundle(cardBundle);
} catch(IllegalArgumentException e) {
   System.out.println(e);
   return;
}

System.out.println("Sync counter deleted");
```

The Delegated Management Facility

OTA Manager V5.1 supports the Delegated Management mode of operation described in the GlobalPlatform 2.1.1 specification. The Delegated management facility is not supported in the CCI; it is only available through the Core API.

In Delegated Management mode, the mobile network operator (MNO) and the Trusted Service Manager (TSM) each operate and maintain their own OTA platforms. OTA Manager V5.0 can therefore be deployed by the MNO, the TSM, or both. As the card issuer, the MNO can authorize a TSM to perform delegated loading, installing and deleting of applications on the card.

Note: Deployment by a TSM in Delegated Management mode is outside the scope of OTA Manager V5.1.

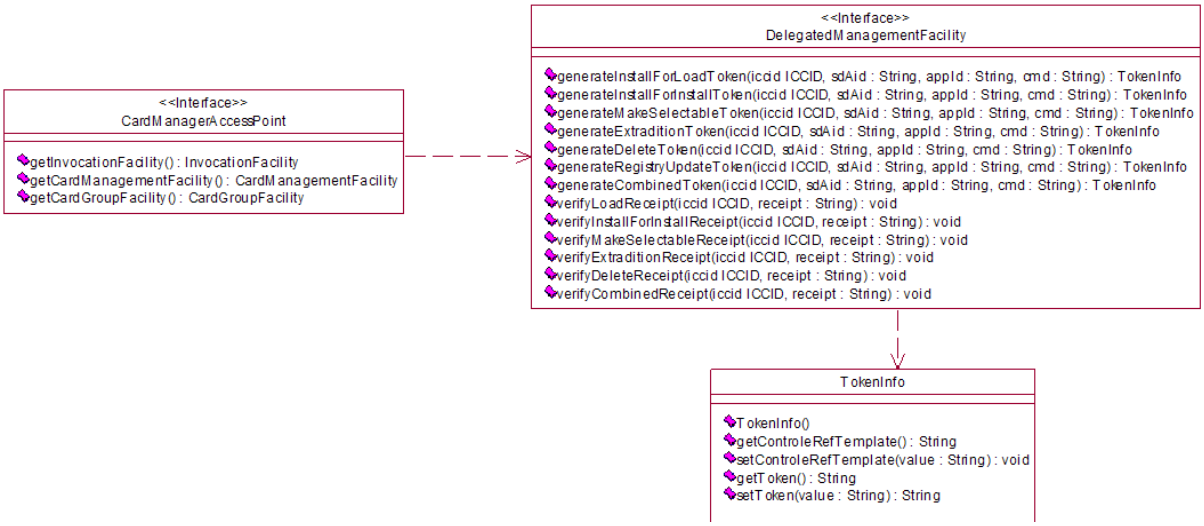
Tokens and *receipts* are used to allow verification of a TSM's authority to perform a task before actually performing the task. For example, the TSM must request a suitable Load token from the MNO's OTA platform before invoking any service that utilises the **Install (for Load)** GlobalPlatform command.

The Delegated Management facility allows:

- **Install for Load** token generation
- **Install for Install** token generation
- **Install for Make Selectable** token generation
- **Install for Extradition** token generation
- **Delete** token generation
- **Load** receipt verification
- **Install for Install** receipt verification
- **Make Selectable** receipt verification
- **Extradition** receipt verification
- **Delete** receipt verification

Access to the **com.gemplus.gxs.api.DelegatedManagementFacility** interface is through a method in the **CardManagementFacility** interface:

Figure 39 - The Delegated Management Facility Interface



Each entry point of the **DelegatedManagementFacility** interface is a synchronous method call, that is, the client is suspended until the method completes.

5

Invocation Submission Flow Control

OTA Manager V5.1 includes an extension to the Core API that enables you to control the flow of invocations submitted to the platform.

Busy Status Exception Handling

OTA Manager V5.1 provides a facility for handling invocations when the platform is operating under a heavy load. Previously, whenever the platform was overloaded, the invocation submission API (typically the `SmartCard.post()` method) systematically blocked the client thread call until an invocation was accepted by the platform.

OTA Manager does not block the client thread call, but instead returns an exception to notify the client application that the platform is busy.

OTA Manager's invocation API manages this "busy" status notification separately for each invocation priority level. This means, for example, that the platform might return a "busy" exception for a "normal" priority invocation, while at the same time accepting an "urgent" invocation.

OTA Manager uses the size of the InvocationManager component's internal queues (one queue for each priority) to manage the "busy" exception. When the queue associated with a given priority reaches its maximum size, new invocations are rejected and a "busy" exception is raised.

Determining when the Platform Lifts Busy Status

The value of the Card Manager product parameter INVOCATION_MANAGER/SEQUENCER_ENDOFBUSY_SIZE specifies the number of invocations in a queue that signals the end of the "busy" status. For example, if this parameter is set to 50 and there are currently 80 invocations in the queue, the product is in "busy" status. When the number of invocations in the queue next falls to 49, the "busy" status is lifted and new invocations are again accepted.

The value of the INVOCATION_MANAGER/SEQUENCER_ENDOFBUSY_SIZE product parameter must be set at Card Manager start time (it is a "cold start" product parameter).

The value of the INVOCATION_MANAGER/SEQUENCER_ENDOFBUSY_SIZE product parameter must be greater than 0 and less than the value of INVOCATION_MANAGER/SEQUENCER_SIZE.

The INVOCATION_MANAGER/SEQUENCER_ENDOFBUSY_SIZE product parameter value determines the end of “busy” state for all queue priorities.

Once OTA Manager enters “busy” status, all new invocations submitted are rejected with a “busy” exception until the corresponding queue size returns below the value of the INVOCATION_MANAGER/SEQUENCER_ENDOFBUSY_SIZE product parameter.

Refer to the *OTA Manager V5.1 Administration Guide* for more details on these product parameters.

6

Runtime Exceptions

This chapter lists and describes possible runtime exceptions that display on the client interface when developing applications for OTA Manager V5.1.

Table 1 - List of Exceptions

Exception	Description
TransportException	Thrown if an exception occurs during the transport phase
ClosedException	Thrown when the Registry thread pool for the Invocation Manager is closed (Invocation Manager is not running well)
ApduException	Thrown if there is a problem while performing Download on GemXplore 98 cards. Check contextual message.
CampaignManagerException	General Exception for campaigns. Check contextual message: Init, starting, parameters check, execution, campaign data base management.
DatabaseConcurrencyException	If multi-threaded clients work on the same card, this exception may be thrown when two applications attempt to update a SIM card concurrently.
InvalidEligibilityLevelException	Target (MSISDN) cannot be marked for execution. Check contextual message for campaign ID and MSISDN.
MalformedSQLRequestException	Internal SQL requests malformed. Check contextual message.
MIException	GUI exception, thrown if there is a driver management problem
InvocationRegistryException	General exception for invocation registry. Check contextual message: 1. Checking invocation (invocation object, identifier, params and QoS parameters) 2. Invocation registry management (delete, insertion, getting and processing)
DupplicatedInvocationIdentifier Exception	Thrown if an identical invocation identifier is inserted in the registry
UnknownInvocationIdentifierException	Thrown if invocation defined by its identifier does not exist in registry
InvocationManagerException	General Exception for Invocation Manager. Check contextual message: Init, starting, parameters check, execution.
ContainerException	General Exception for remote access. Check contextual message.

Table 1 - List of Exceptions (continued)

Exception	Description
IllegalServiceDomainException	Thrown if there is problem with wrong service publisher domain (secure mode)
IllegalServiceTypeException	Thrown if there is problem with wrong service publisher type
MarshalException	Thrown if there is problem with serialization or class version compatibility
MissingPropertyException	Thrown if the property associated to the published service does not exist
NoMatchingServiceProperty Exception	Thrown if the property of the service to be managed is undefined
NoMatchingServiceClassException	Thrown if the service specified cannot be found. For example: ■ The requested service has not previously been published (a service is an entry point of the Core API, for example). ■ The server application that publishes the requested service could be not started or the requested service doesn't exist.
ServiceAccessDeniedException	Thrown if access rights for service are not allowed
ServiceBreakdownException	Thrown if it is impossible to access the published service (for example, if the Card Manager (RCA) is not started and the card profile service cannot be accessed)

Abbreviations

ADN	Abbreviated Dialing Numbers
AID	Application Identifier
APDU	Application Protocol Data Unit
API	Application Programming Interface
BIP	Bearer Independent Protocol
CC	Cryptographic Checksum
CAT-TP	Card Application Toolkit Transport Protocol
CCA	Customer Care Agent
CDR	Call Detail Recording
CORBA	Common Object Request Broker Architecture
CMS	Card Management System
CSV	Comma Separated Value
DCS	Data Coding Scheme
DF	Dedicated File
DTD	Document Type Definition
ETSI	European Telecommunications Standards Institute
GMT	Greenwich Mean Time
GP	Global Platform
GSM	Global System for Mobile communications
JAR	Java Archive
JVM	Java Virtual Machine
MF	Master File
MIB	Management Information Base
MO	Mobile Originating
MT	Mobile Terminating
MSISDN	Mobile Station International Subscriber Directory Number
NPI	Numbering Plan Identifier
OID	Object Identifier
ORB	Object Request Broker
OTA	Over the Air
PID	Protocol Identifier
PoR	Proof of Receipt

QoS	Quality of Service
RC	Redundancy Check
SIM	Subscriber Identity Module
SMS	Short Message Service
SMSC	Short Message Service Center
SNMP	Simple Network Management Protocol
SPI	Security Parameters Indicator
TON/NPI	Type of Number/Numbering Plan Identifier
STK	SIM Toolkit
VAS	Value Added Services
WIR	Wired Internet Reader
XML	Extensible Markup Language

Glossary

Applet	A Java application loaded onto a Java card compatible SIM card.
Audit trail	A chronological record of the progress of a transaction that allows an auditor to determine that no errors occurred in the transaction.
Batchloading	A method of loading multiple objects defined in XML format into a database simultaneously. For example, multiple card definitions can be batchloaded directly into the Card Manager database.
Bearer Independent Protocol	A protocol that allows applications to use a distribution channel other than SMS to communicate with cards.
Campaign	The action of downloading, updating, or activating services on a targeted set of cards.
Card content	An image of the content of any card defined in the OTA Manager platform. Card content includes: ■ The file system with all file types: Dedicated files, transparent files, linear files and cyclic files. ■ Native applications and services ■ Java packages, applets and applet instances
Card instance	An image of the content of a SIM card, stored in the Card Manager instance. OTA Manager V5.1 updates the card instance whenever service requests or campaigns successfully update the corresponding SIM card.
Card structure	The physical composition of a SIM card in terms of the hierarchical file structure (master files and directory files) and applications (native applications and Java applets).
CORBA	A distributed object computing infrastructure that allows applications to work together over networks.

Core API	An application programming interface that allows the development of custom programs and integration with other platforms or third-party systems.
Cryptographic checksum	A string of bits derived from some secret information (for example, a secret key), part or all of the message's contents, and possibly additional information (for example, part of the security header). The secret key is known to both the sending entity and receiving entity. The Cryptographic Checksum is often referred to as a Message Authentication Code (MAC).
Data coding scheme	Indicates the format of the text or data contained in the user data part of an OTA message (for example, 7-bit text or 8-bit data). The data coding scheme tells the mobile equipment how to handle the data contained in the message (for example, whether to display a text message on the mobile equipment's display), how to store the data, and whether the data contained in the message is compressed.
Key set	A set of encryption keys used to authenticate the owner.
Master file	The mandatory DF representing the root of the GSM file system.
Provisioning	The process of populating OTA Manager V5.1 databases with objects (card profiles, user profiles, group definitions, security settings, and so on).
Security domain	A security domain is a special kind of applet used to store another applet's security data (synchronization counters, cryptographic key sets, and so on). Each Java Card applet must be associated with a security domain (either its own or the default one).
SNMP agent	See *agent.*
Synchronization counter	A security mechanism used to prevent replay attacks. The synchronization counter is incremented each time an OTA message is sent and its value sent with the message. The receiving entity increments its own synchronization counter each time a message is received, and compares this to the value in the message.
Trap	A message from an SNMP agent indicating a situation that requires immediate attention. You can set a threshold value that determines when the trap is sent.
User	An individual or external system connected and logged on to OTA Manager V5.1. User types are administrator, customer care agent and subscriber.
User account	Based on and linked to a user profile, a user account contains authentication information and specific characteristics of an individual user, such as account status, and expiry date.
User profile	A "template" describing the operations that a particular type of user is authorized to perform. Types of user profiles include administrators, customer care agents and subscribers