# cgJCE running on Java Systems

## User Guide

Version 1.0

06/27/2016

Autor: certgate

## Document History

| Version | Date | Autor | Notes |
|---|---|---|---|
| 1.0 | 27.06.2016 | certgate | - |

**Content**

0 About this Guide ..... 4
1 Introduction to JCE ..... 6
1.1 Provider ..... 6
1.2 Service ..... 6
1.3 SPI ..... 6
2 cgSmartCardProvider ..... 8
2.1 SDKs ..... 8
2.2 Architecture ..... 9
2.3 Concepts ..... 12
2.4 SmartCardAuthProvider ..... 16
3 Managing Classes ..... 22
3.1 ReaderManager ..... 22
3.2 ModuleManager / P11ModuleManager ..... 23
3.3 SmartCardAuthProviderManager ..... 25
4 Differences JCE ..... 26
4.1 Objects ..... 26
4.2 KeyStore ..... 26
4.3 KeyPairGenerator / KeyGenerators ..... 27
4.4 Configuration ..... 28
5 Loading our JCE ..... 30
5.1 Loading our JCE on Windows / Linux – standard Oracle Java ..... 30
5.2 Loading our JCE on Android ..... 31
6 Examples ..... 34

## List of Figures

Figure 1: abstraction layer / SDKs ........ 9
Figure 2: System architecture JCE / JCE SDK ........ 10
Figure 3: roughly overview of SDKs ........ 11
Figure 4: SmartCardCryptoFactory hirarchie ........ 14

## List of Tables

Table 1: crypto interfaces - impls - mapping to smartcard interfaces ........ 20

## 0 About this Guide

This guide - written for application developers – gives information how to use certgate’s JCE provider implementation with dedicated smartcards. Furthermore we give a brief overview about the chosen architecture, the main idea behind our concepts, some featured managing classes and how to load our JCE provider into your system. This guide is devided as follows:

Chapter 1.

First we introduce you with some basic knownledge about the JCE and its underlying class layout.

Chapter 2.

Then we provide detailed information about the concepts used for implementing our SmartCardAuthProviders and explain our SDKs.

Chapter 3.

Afterwards we go more into detail and show the features of the managing classes responsible for creating and registering SmartCardAuthProviders to the system.

Chapter 4.

Followed by revealing the differences between our smartcard based provider services and usual provider services.

Chapter 5.

Before we come to the end we explain how to load our providers.

Chapter 6.

Finally we give some examples which explain obscurities which may arised during reading the previous chapters.

**Who is this guide for?**

This guide is written for developers who intend to implement own smartcard based providers (use of SDK) or just use the implemented SmartCardAuthProvider's features to provide smart-card based hardware security to their programs.

**What typographical conventions are used?**

Warning

Provides mandatory information which should always kept in mind

Note
Provides additional information on a topic, and emphasize important facts and considerations.

**Note**
You should have some basic understanding about Public-Key cryptography, digital certificates, digital signature and Public Key Infrastructure (PKI) in order for you to understand the discussed topics.

# 1 Introduction to JCE

The Java Cryptographic Engine – short JCE – is a concept to simply supply developers with cryptographic services. The concept includes several interfaces which shall be implemented via SPI classes. A SPI class implements a specific cryptographic service and is then registered to a Provider. Providers[1] are registered[2] to the system and can be used by everyone who needs their functionality. Within this chapter we give you some information about the basic concepts of the JCE where detailed information can be found at www.oracle.com.

## 1.1 Provider

A provider can be seen as a kind of container offering different services. Each of its services is registered to the provider before or after the provider is registered to the system. During system registration the provider receives a priority number starting by 1 signaling highest priority. This priority is necessary when someone requests a service not indicating from what provider it shall come from. The JCE then seeks for the service and – assuming more than one provider would offer that service – returns the one from the provider with the highest priority.

## 1.2 Service

A Service is an implementation of a SPI class offering a specific algorihm. Services can be registered before the containing provider is registered to the system or at any time later. Furthermore a Service can be de-registered at any time.

## 1.3 SPI

The JCE describes several SPI classes containing pre-defined functions. These functions are specific for the class-type and shall be implemented by the extending class implementing the specific algorithm. The JCE distinguishes between the following class types[3]:

- Cipher
- Signature
- MessageDigest
- KeyGenerator
- KeyPairGenerator
- SecureRandom
- KeyStore

[1] cryptographic or non-cryptographic
[2] static- or dynamically
[3] there are even more SPI classes – within this document we only focus the ones we implemented

Each SPI class offers static functions in order to request a specific algorithm. Triggering this request is done provider specific or even anonym when it is not important where the service shall come from. The following example shows three common variants how to receive e.g. a cipher object implementing the RSA algorithm:

```
//get RSA cipher from any provider who implements it
//
//in fact the service will come from the provider implementing it
//having highest priority
Cipher c = Cipher.getInstance("RSA");


//get RSA cipher from "myProvider"-provider
Cipher c = Cipher.getInstance("RSA", "myProvider");


Provider xy = null;
//xy is retrieved somewhere
…
//get RSA cipher from provider-instance xy
Cipher c = Cipher.getInstance("RSA", xy);
```

The last two examples do exactly the same. They use a specific provider – once identified by the provider's name, once identified by the instance – in order to get the RSA algorithm from it.

`getInstance` is supported by each SPI class and is always used for receiving algorithms. When the algorithm can not be found at any provider / is unsupported by the provider an exception is thrown.

# 2 cgSmartCardProvider

Overall the JCE is more a software based crypto-solution i.e. the concept was not designed to use cryptographic hardware. Nevertheless we implemented an own crypto provider based on smartcards where most of the cryptographic calculations[4] are executed in hardware. For this purpose we desgined an own SDK extending providers as smartcard-providers. In this chapter we give an overview of our basic concepts and all extra-features offered by smartcard-providers.

## 2.1 SDKs

For future purposes we designed two SDKs. Both of them define base classes and implement basic functionality. The SDKs are:

- cgSDK CORE

  The CORE SDK is completely independent and defines the interfaces for the smartcard layer. Furthermore it implements reader managing services and defines basic data types.

- cgSDK JCE

  The JCE SDK uses the CORE SDK and specifies the interfaces mandatory for the JCE layer. Furthermore it implements the Java SPI classes mentioned in 1.3 but does not implement any algorithm. Compared to usual SPI implementations they behave more general and forward function calls to specialized implementations. For this purpose we specified a set of own SPI interfaces (see 2.3.1 for details) which shall be implemented by those specialisations.

  The JCE SDK already implements the SmartCardProviders which serve the JCE.

Figure 1 shows how the SDKs are layered. On top there is an application which wants to make use of the bottomed smartcard-layer. Between there are our SDKs which handle the requests in conjunction with the Java JCE and the the layers which implement our interfaces.

The CORE SDK is independent and has no knownledge about JCE functionality. Its main purpose is to manage smartcard readers and communicate with smartcards.

The JCE SDK uses the CORE SDK and serves the JCE where it does not need to know anything about how the CORE SDK communicates with smartcards.

Due to this design decision we are be able to switch both implementation layers without changing the other.

[4] all secure operations are completely executed on the smartcard. Public operations like verifying a signure may be done in software.

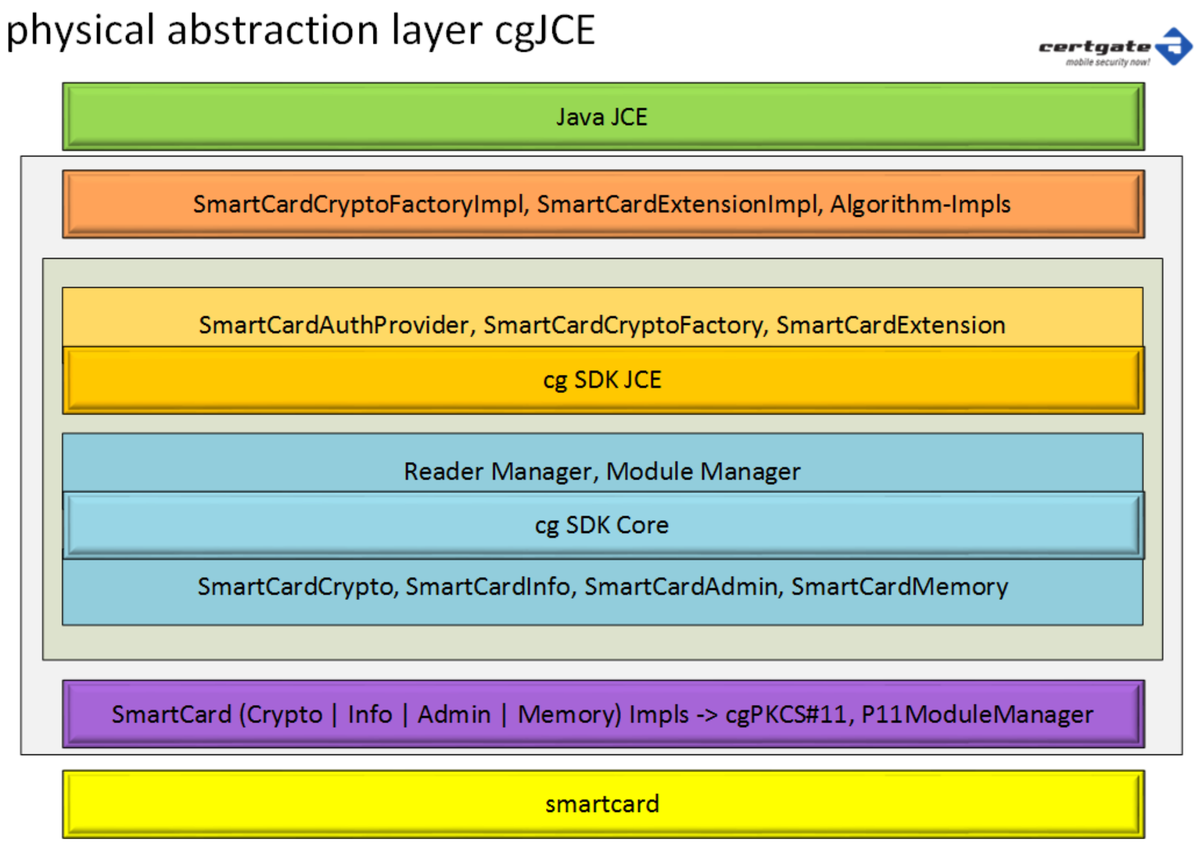
physical abstraction layer cgJCE
certgate mobile security now!
Java JCE
SmartCardCryptoFactoryImpl, SmartCardExtensionImpl, Algorithm-Impls
SmartCardAuthProvider, SmartCardCryptoFactory, SmartCardExtension
cg SDK JCE
Reader Manager, Module Manager
cg SDK Core
SmartCardCrypto, SmartCardInfo, SmartCardAdmin, SmartCardMemory
SmartCard (Crypto | Info | Admin | Memory) Impls -> cgPKCS#11, P11ModuleManager
smartcard

**Figure 1: abstraction layer / SDKs**

## 2.2 Architecture

Before going into detail we first have to discuss the JCE's architecture rougly depicted in Figure 2. On the left side there are the standard JCE classes where our special implementations are on the right. Providers[5] always contain the algorithms they provide as services. An algorithm is implemented in an extended SPI class and then connected to a service. A service is then added to a provider.

For extending the JCE with smartcard functionality we developed two smartcard concepts explained in 2.3. The SmartCardCryptoFactory handles the usual functionality offered by the standard JCE where the SmartCardExtension brings new functionality dedicated for smartcards (e.g. changing the PIN, unlock the user PIN, etc. …). For both concepts we created interfaces which are used by our SmartCardAuthProvider.

Figure 3 gives a deeper look at the architecture and the SDK barriers. On top there is the JCE-Impl layer which implements the JCE SDK's interfaces and all necessary algorithms. Furthermore it implements a SmartCardAuthProviderManager which automates the whole creation & registration process of providers (more in 3.3).

The JCE- and the CORE layer define interfaces for the top and bottom layers. They implement the providers and other managing classes which are described in 3.

[5] Provider / AuthProvider

On the hardware layer also called the SmartCard Impl we handle the communication with the smartcard and serve the interfaces defined in SDK CORE. We decided to use cgPKCS#11 which already handles smartcard communication in conjunction with PCSC-lite.

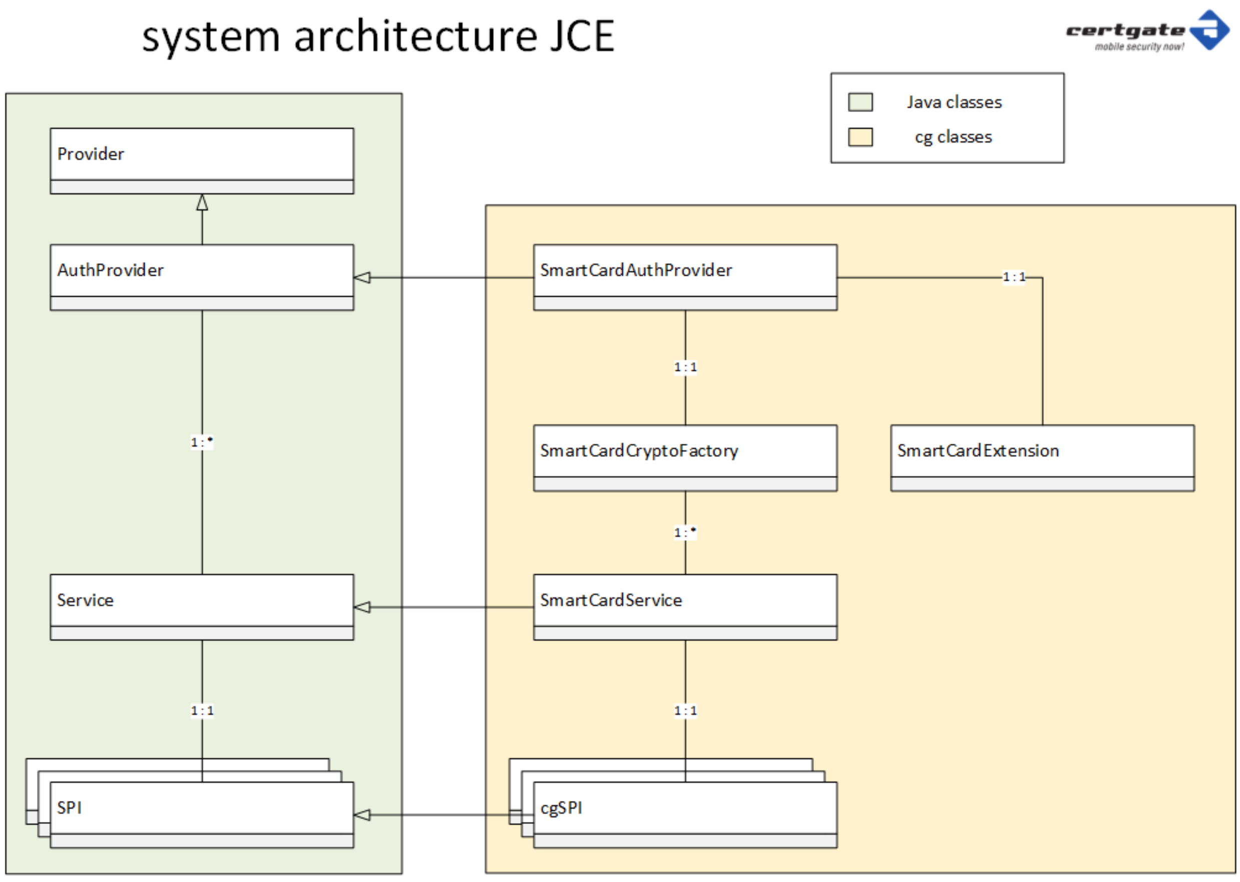

**Figure 2: System architecture JCE / JCE SDK**

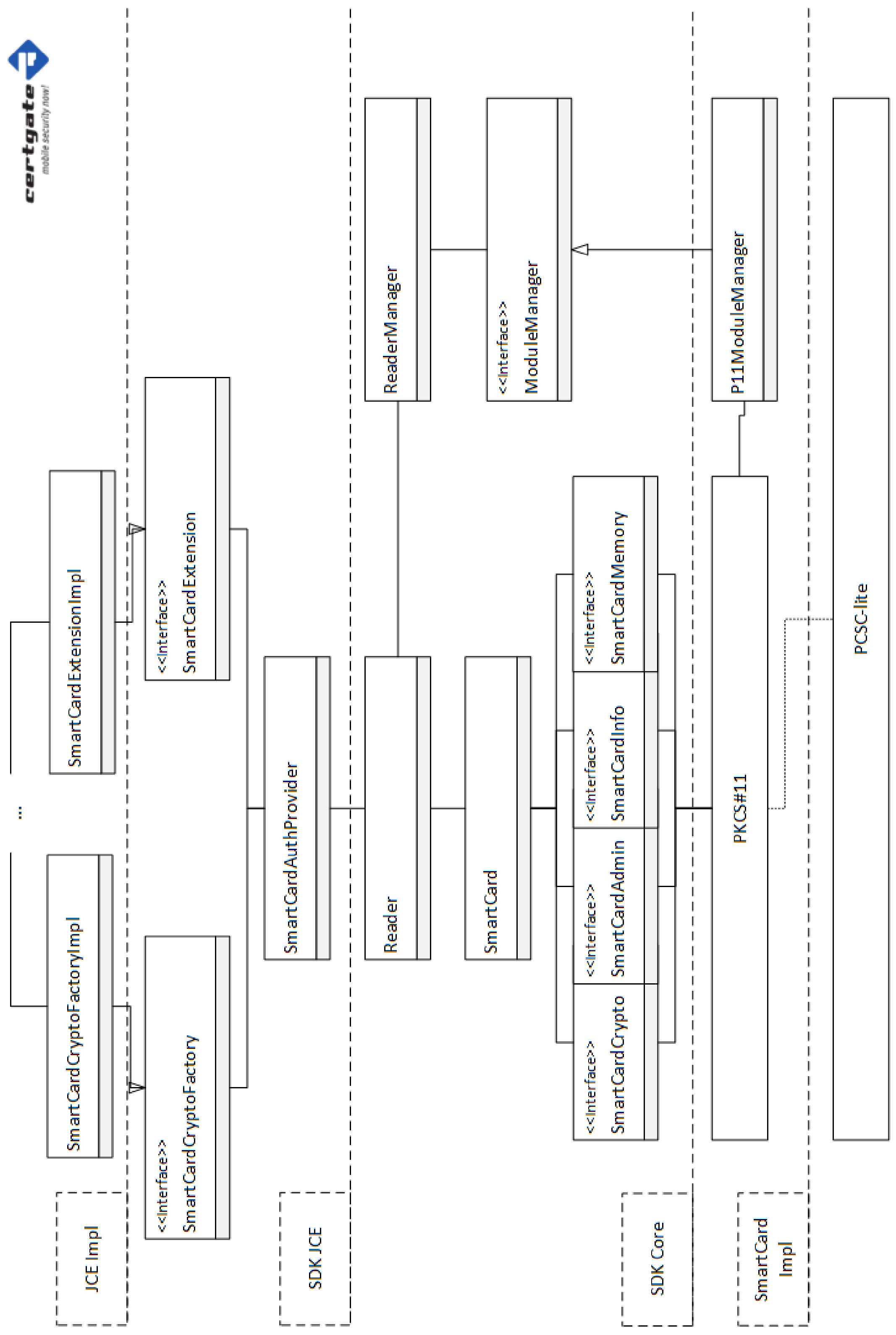
JCE Impl
SmartCardCryptoFactoryImpl
SmartCardExtensionImpl
...
<<Interface>> SmartCardCryptoFactory
<<Interface>> SmartCardExtension
SDK JCE
SmartCardAuthProvider
Reader
ReaderManager
SmartCard
<<Interface>> ModuleManager
<<Interface>> SmartCardCrypto
<<Interface>> SmartCardAdmin
<<Interface>> SmartCardInfo
<<Interface>> SmartCardMemory
SDK Core
PKCS#11
P11ModuleManager
SmartCard Impl
PCSC-lite

Figure 3: roughly overview of SDKs

### 2.3 Concepts

In order to implement smartcard based providers we developed two different concepts – one for handling usual JCE tasks and another extending a provider's functionality with smartcard specific features. The SmartCardCryptoFactory covers the JCE features while the SmartCardExtension brings the extra functionality. In the following we handle both concepts.

#### 2.3.1 SmartCardCryptoFactory

The SmartCardCryptoFactory is a concept which serves the standard JCE functionality. The factory cooperates with the CORE SDK and has the following tasks:

- Query the CORE SDK which algorithms are supported by the smartcard:
  - `getSupportedStreamCiphers`
  - `getSupportedSingleBlockCiphers`
  - `getSupportedKeyPairGens`
  - `getSupportedKeyGens`
  - `getSupportedSignatures`
  - `getSupportedDigests`

  Those functions class-specifically fetch the smartcards supported algorithms and return them to the caller.
- Create crypto instances which can be forwarded to the JCE:
  - `getStreamCipher`
  - `getSingleBlockCipher`
  - `getKeyGenerator`
  - `getKeyPairGenerator`
  - `getSignature`
  - `getSecureRandom`
  - `getKeyStore`
  - `getDigest`

  Those functions create algorithm specific objects which are transferred to the corresponding generalized SPI class mentioned in 2.1. To be more precise each function is called with the demanded algorithm as parameter. SmartCardCryptoFactory's task is then to create the matching instance and return it to the caller.

The SmartCardCryptoFactory uses the SmartCardCryptoFunctions- and SmartCardPersistentMemoryFunctions-interfaces defined in the CORE SDK. Both interfaces need to be implemented at the smartcard layer.

The JCE SDK already implements generalized SPI classes for all the ones mentioned in 1.3. Usually someone has to extend from one of the Java SPI classes in order to implement a single algorithm. Our SPI classes are implemented more generally and not algorithm specific since they shall support dynamic loading i.e. when a provider detects that a card is present it querys which hardware algorithms are supported by the smartcard and adds them as its provided services.

Of course we could implement each supported algorithm as own SPI class but we found a solution making our providers quite more flexible.

First we analized the Java SPI classes and extracted all abstract functions to own SPI interfaces (com.certgate.sdk.jce.crypto.interfaces). These interfaces are defined within the JCE SDK. Then we extended and implemented the Java SPI classes (com.certgate.sdk.jce.crypto.spis) which later make use of the objects implementing the SPI interfaces. For provider implementations it is usually necessary to exactly know which algorithms are supported. Our design avoids that circumstance and allows us to dynamically load the smartcards supported algorithms. Furthermore this design allows us that the JCE SDK already provides the whole logic for implementing a provider – only the implementations of our SPI interfaces need to be additionally delivered. This gives us the opportunity to deliver pre-signed code running on all machines[6]. Figure 4 illustrates the hirarchie of the concept. See 2.4.5 for more details.

[6] on Oracle Java systems it is necessary to sign the provider with an Oracle signed certificate. Otherwise Cipher SPI objects will not operate with the Oracle Java JCE!

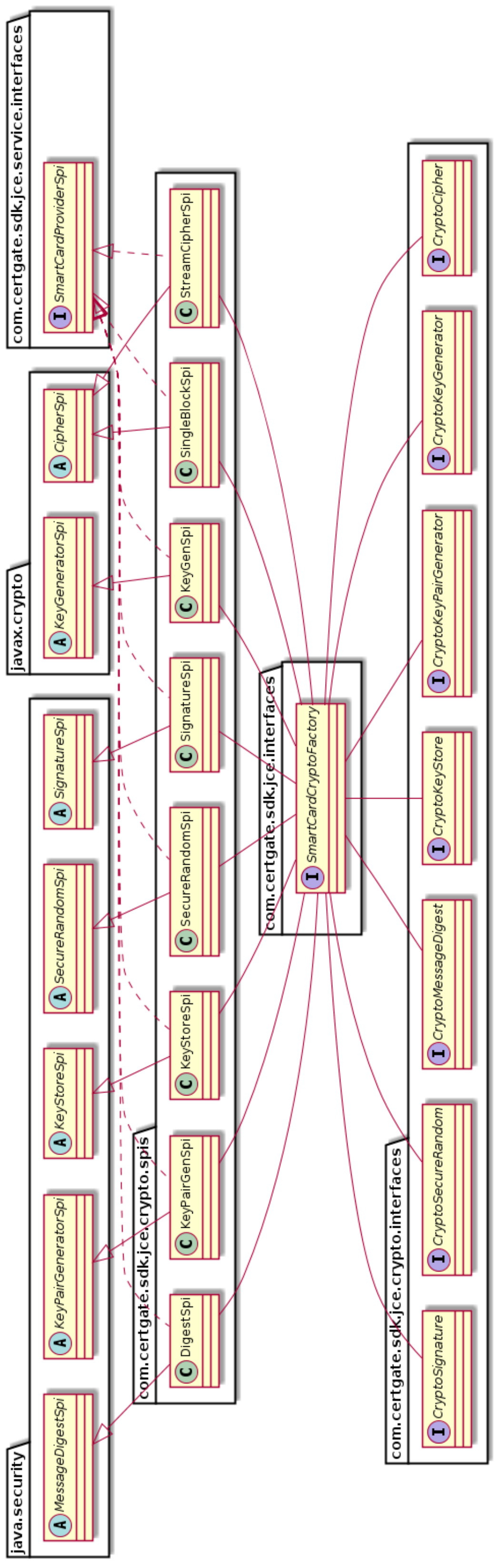

**Figure 4: SmartCardCryptoFactory hirarchie**

### 2.3.2 SmartCardExtension

The SmartCardExtension is a concept which extends the functionality of a SmartCardAuthProvider with additional features. These features are:

- `getRemainingAttempts`

  This function returns an enum representing how many login attemps can be executed until the card is locked. These values represent “at least 2”, “last try” and “locked”. Unfortunately there is no indicator[7] which enables us to differentiate between two or three remaining tries.

- `verifyPIN`

  This function validates the PIN for a specific user (SO / USER).

- `changePIN`

  Changes the PIN for a specific user (SO / USER).

- `overwriteUserPIN`

  Overwrites the users PIN.

- `getFirmwareVersion`

  Returns the firmware version of the smartcard.

- `getManufacturerID`

  Returns the manufacturer ID of the smartcard.

- `getSerialNumber`

  Returns the smartcards serial number.

- `getCardType`

  Returns the type of smartcard.

- `getReaderName`

  Returns the reader name.

- `getFreeMemorySizeByte`

  Returns the size of free memory in bytes.

- `getSupportsRNG`

  Returns whether the smartcard supports a hardware random number generator.

- `registerCardEventObserver`

  Registers an observer which will receive smartcard events.

- `isCardPresent`

  Returns whether a smartcard is currently inserted.

[7] we are using cgPKCS#11 for smartcard communication. PKCS#11 does not provide better information regarding PIN retry attempts.

The SmartCardExtension uses the SmartCardAdministrationFunctions- and SmartCardInfoFunctions-interfaces defined in the CORE SDK. Both interfaces need to be implemented at the smartcard layer.

In order to verify, change or overwrite / unlock PINs the smartcard is not allowed to be in private state. For this reason re-load a potential privatly loaded keystore to public mode and ensure that no pending private operation is ongoing.

## 2.4 SmartCardAuthProvider

A SmartCardAuthProvider is extended from the standard Java class AuthProvider and uses the concepts explained in 2.3. Nevertheless there are a few more concepts which are discussed within the next subchapters.

### 2.4.1 Reader

The CORE SDK provides an easy to use reader concept handled by the ReaderManager (see 3.1). As soon as a reader is reserved a SmartCardAuthProvider shall bind it i.e. when more readers are available potentially more than one SmartCardAuthProvider can be registered to the Java system (see 3.3).

During the lifetime of a SmartCardAuthProvider the reader never changes. The reader can be retrieved by calling `getReader` while calling `getReaderState` gives information about whether a smartcard is inserted currently or not.

A SmartCardAuthProvider further on listens only to that reader and reacts to ReaderEvents.

### 2.4.2 ReaderEvent

We designed SmartCardAuthProviders to react on three different reader events:

- smartcard removal

  When a smartcard is removed the SmartCardAuthProvider cancels all pending operations and unregisters it's services. Furthermore all returned SPI objects (i.e. previously returned services) e.g. a cipher and a keystore are locked and resetted (buffers & states). Potentially cashed PINs are also cleared. Trying to use such a locked object will always fail with an exception[8].

- smartcard insertion

  When a smartcard is inserted the SmartCardAuthProvider starts interacting. First all card supported features are fetched and registered as services. Then the provider transmits the smartcards serialnumber to all previously returned objects. The objects compare the serial with the one once used for creation and unlock when the serials match. Unlocked objects may need to be re-initialized (cashed PINs & states are gone). A potentially returned keystore re-loads all contained objects in public mode. See more about keystore modes in 4.2.

[8] when the SPI object's function is able to return an exception. Otherwise null is returned or nothing happens.

- reader removal

  When the reader is removed the provider unregisters itself at the SmartCardAuthProviderManager and releases it's services[9]. All objects returned by the provider are no more usable and locked forever.

Furthermore a SmartCardAuthProvider can forward these events to all observer classes. For receiving such an event register your observer via `addObserver`. For unregistering a previously registered observer call `removeObserver`. Observers (standard Java Interface) will receive `ReaderStateBundle`s which contain the new ReaderState and an instance of Smartcard (null when card or reader where removed) as soon as an event happens.

### 2.4.3 Handlers / Callbacks

Typically there are several types of standard callbacks where in fact just one plays a special role for our implementation – the password-callback. The password-callback is a standard Java class which is used to prompt a notification to - and return a password from - a user. The way how the user enters his password is chosen by the handler which is implementation specific.

For our implementation we distinguish between three handlers:

- The default callback handler

  This handler is implemented by us which outputs the prompt on System.out and reads a password via System.in.

- The application callback handler

  This handler is implementer specific and can be set via `setCallbackHandler`.

- The login callback handler

  This handler is implementer specific and is temporarily set via `login`. The handler stays active as long as `logout` has not been called.

All three handlers have different priorities. As long as `setCallbackHandler` has not been called (or was called with null) the default callback handler stays active. When a callback handler was set via `setCallbackHandler` the application callback handler stays active as long as no `login` occurs. When `login` was called (with handler != null) the login callback handler stays active as long as `logout` has not been called. When `logout` was called the application callback handler is active again or the default callbackhandler when no application callback handler was set. To make a long story short – the login callback handler has the highest priority where in contrary the default login handler has the lowest.

The password-callback is used to query the user for the smartcard's PIN. The PIN is needed as soon as the smartcard operates with private objects such as private- or secret keys. The PIN can also be cached.

[9] this only happens when the SmartCardAuthProvider was once registered to the ReaderManager. The registration is executed as soon as the SmartCardAuthProvider is used the very first time.

### 2.4.4 PIN caching

During a `login / logout` session the SmartCardAuthProvider will just ask once for the PIN and will reuse it for all private operations[10].

The same applies for the KeyStore. As soon as the keystore's `load` function is called with a valid PIN no password-callback will be triggered until load is called again or the card is removed and reinserted.

Additionally readers can be configured that the PIN is always cached as soon as it was once entered successfully. The cache is active as long as the smartcard stays within the reader. On removing the card all PIN caches are cleared. This also happens when the reader is removed or detached by ReaderManager. This cache is activated on reader-attachment. For this purpose set the appropriate flag to true when attaching the reader (see 3.1) enabling the PIN cache.

When PIN caching is disabled and no `login / logout` session is active the user is always prompted for the PIN as soon as a cryptographic operation shall be executed.

### 2.4.5 SmartCardService

As we already learned providers offer specific services which can be retrieved by calling the `getInstance` function of the several JCE's SPI classes. Usually a provider is pre-defined i.e. an implementer plans to support specific services / algorithms and adds them to his / her provider. Then the provider is shipped e.g. within a library. Someone who wants to make use of the provider creates an instance of it and registers the provider to the system[11]. Afterwards the defined services can be retrieved and used as described in 1.3.

We implemented our providers more generally i.e. we did not pre-define which algorithms we support. In fact our provider itself makes the desission which algorithms it supports during runtime. Therefore it uses the CORE SDK to communicate with the reader it's bound to and asks whether a smartcard is inserted. When a smartcard is inserted the CORE SDK returns all available algorithms supported by the smartcard. Afterwards these algorithms are registered as the providers services.

Usually the registration needs fixed classes where each one implements a single algorithm. Also our provider works this way but indeed it adds always the same SPI class type. The following example will give some clarity:

Someone plans to implement a provider offering a RSA - and an EC keygenerator. Therefore he / she would extend from Java KeyPairGeneratorSpi two times:

- One for the RSA keygenerator implementation called class A.
- Another for the EC keygenerator implementation called class B.

Both classes would be implemented algorithm specific and then added via

- provider.putService(new Service(this, "KeyPairGenerator", "RSA", **class A**, null, null));
- provider.putService(new Service(this, "KeyPairGenerator", "EC", **class B**, null, null));

[10] only when the entered PIN was correct.
[11] there is also a way to activate the provider statically for all programs. To do so the provider has to be installed into the Java archive. For more information see 5.1.1.

Our provider is working differently since it shall support dynamic algorithm loading. Lets use the given example in order to explain that.

For each supported SPI class we implemented a generalized class like **KeyPairGenSpi** in our JCE SDK. This class needs an instance of our **CryptoKeyPairGenerator** interface which is received during runtime.

So for this example someone would need to implement two classes implementing the **CryptoKeyPairGenerator** interface:

- One for the RSA keygenerator implementation called impl A.
- Another for the EC keygenerator implementation called impl B.

Both classes would be implemented algorithm specific and then added via

- provider.putService(new Service(this, "KeyPairGenerator", "RSA", **KeyPairGenSpi**, null, null));
- provider.putService(new Service(this, "KeyPairGenerator", "EC", **KeyPairGenSpi**, null, null));

The **KeyPairGenSpi** then has to decide during runtime which instance – impl A or impl B – shall be used. So another step would be to register the “impls” at **KeyPairGenSpi**.

For this reason we implemented the **SmartCardService** class and another interface called **SmartCardProviderSpi**. The SmartCardService receives an object type called **Algorithm** instead the algorithm name like “RSA” or “EC” and extends from Service. SmartCardService has an overloaded `newInstance` function which makes use of the SmartCardProviderSpi interface implemented in KeyPairGenSpi. Instead of putting a service we then put a SmartCardService like this way:

provider.putService(new SmartCardService(this, "KeyPairGenerator", **keyPairGen**, **KeyPairGenSpi**, null, null));

where **keyPairGen** is an instance of type Algorithm which is dynamically received from the smartcard via the CORE SDK. The `newInstance` function of SmartCardService then later uses the SmartCardProviderSpi’s declared `setSpiProperties` function implemented in KeyPairGenSpi which communicates with the **SmartCardCryptoFactory** in order to receive the correct algorithm instance of type **CryptoKeyPairGenerator** indicated in **keyPairGen**.

In fact we even more simplified the necessity to implement impl A and impl B. Actually we only need to implement one impl per interface! See 2.4.6 for more information.

The “**KeyPairGenSpi -uses-> CryptoKeyPairGenerator**”-concept applies to all of our supported SPIs (see 1.3).

### 2.4.6 Crypto Impls

So far we have learned most of our concepts which turn usual AuthProviders into SmartCardAuthProviders. Both SDKs – the CORE SDK and the JCE SDK – implement most of the logic used for provider creation / registration. On the top layer we implemented all necessary interfaces required for the JCE SDK. Besides the SmartCardExtension and the SmartCardCryptoFactory these are the Crypto-interfaces listed in Table 1.

We implemented a class – called the “impl” – for each of the interfaces. Each impl is able to communicate with the smartcard and uses whether the SmartCardCryptoFunctions- and / or the SmartCardPersistensMemoryFunctions-interface depicted in Figure 3[12]. Those two interfaces satisfy all the needs of the impls. Table 1 additionally shows how the interfaces are used by which impl.

| **Interface** | **Used Functions** |
|---|---|
| CryptoCipher | SmartCardCryptoFunctions->decrypt<br>SmartCardCryptoFunctions->encrypt<br>SmartCardCryptoFunctions->wrapKey<br>SmartCardCryptoFunctions->unwrapKey |
| CryptoKeyGenerator | SmartCardCryptoFunctions->generateKey |
| CryptoKeyPairGenerator | SmartCardCryptoFunctions->generateKeyPair |
| CryptoKeyStore | SmartCardCryptoFunctions->registerSmartCardKeyGenerationObserver<br>SmartCardPersistentMemoryFunctions->createObject<br>SmartCardPersistentMemoryFunctions->deleteObject<br>SmartCardPersistentMemoryFunctions->modifyObject<br>SmartCardPersistentMemoryFunctions->setCachedPIN<br>SmartCardPersistentMemoryFunctions->findPublicObjects<br>SmartCardPersistentMemoryFunctions->findPrivateObjects<br>SmartCardPersistentMemoryFunctions->loadPrivate<br>SmartCardPersistentMemoryFunctions->loadPublic |
| CryptoMessageDigest | SmartCardCryptoFunctions->hash |
| CryptoSecureRandom | SmartCardCryptoFunctions->generateRandom<br>SmartCardCryptoFunctions->seedRandom |
| CryptoSignature | SmartCardCryptoFunctions->sign<br>SmartCardCryptoFunctions->verify |

**Table 1: crypto interfaces - impls - mapping to smartcard interfaces**

Recently we mentioned that each algorithm – determined for being offered by the provider – should be implemented in an own impl. Since the SmartCardCryptoFunctions-interface func-

[12] the names of the interfaces where shortened there. SmartCardCryptoFunctions map to SmartCardCrypto, SmartCardPersistensMemoryFunctions to SmartCardMemory.

tions always carry the intended algorithm even the impls only need to be implemented once[13] per SPI class because they can potentially forward the demanded algorithm. This leads us to the point that the lowest layer needs to be able to switch between the algorithms since there is no algorithm specific implementation on the upper layers. More precisely this means that the bottom layer has to implement the algorithms used for the services[14].

We decided to use cgPKCS#11 at the bottom layer since this avoids implementing algorithm specific APDU commands for all different crypto features offered by the smartcard. PKCS#11 exactly does this job for us and switches automatically between the distinct algorithms by using pre-defined algorithm IDs. Also all the other CORE SDK features offered by our interfaces can be implemented by using PKCS#11.

[13] as we did this for the SPIs
[14] usualy – as we learned – this would be achieved by implementing serveral SPIs

## 3 Managing Classes

Now as we handled the basics of the JCE and our concepts to enable smartcard based providers we can discuss our manager instances which already implement the most of the logic to create, register and return our providers.

### 3.1 ReaderManager

The ReaderManager is the most important class within our framework. It communicates with the system via the ModuleManager and manages all available readers. It offers different services which are necessary to create a SmartCardAuthProvider. We now should exemplarily go through these services step by step:

1. First an instance of the ModuleManager interface has to be declared to the ReaderManager. See 3.2 for detailed information of the ModuleManager's tasks. The declaration is executed by calling `setModuleManager`.
2. This will lead the ReaderManager to receive the ModuleManagers default module and automatically register it by calling `registerNewModule`.

   Modules are kind of drivers responsible for communicating with a smartcard-reader / smartcard. Each module is able to communicate with a set of smartcard-readers connected to the system. The combination Module / Reader is a unique identifier which lets the ReaderManager know which module it shall use in order to communicate with a reader. Two distinct modules can also share a reader. The **ReaderModuleDescription** stores this unique identifier and is used by the ReaderManager to address the readers via different modules (drivers).

   For each module registered to the ReaderManager a new thread is started which listens to reader events. See 2.4.2 for details about reader events.

   A module can be also unregistered by calling `unregisterModule`. This would stop the module's listening thread and detach all the readers managed by the module. Readers shared with other modules would still be accessible by the other modules.

   As soon as a module is registered to the ReaderManager the readers adressed by the module are accessible via the ReaderManager.
3. Now a module is registerd and the ReaderManager is able to address some readers. Next we can call `getSystemReaders` which returns us all ReaderModuleDescriptions currently available.
4. With such a ReaderModuleDescription we can attach a reader by calling `attachReader`. This call will create a reader instance which is exclisivly reserved and can not be generated a second time. With this reader instance we can create a SmartCardAuthProvider.

   We can also detach a reader by calling `detachReader`. The reader then sends a signal to all its observers that the reader has been detached. The SmartCardAuthProvider listening to that reader then stops operating.

Other features offered by the ReaderManager are:

- `setSystemEventHandler`

  The systemEventHandler is able to react on reader-added and reader-removed events triggered by the ReaderManager. This function registers the handler at the ReaderManager. The interface is implemented by the SmartCardAuthProviderManager (see 3.3).

- `getDetachedReaders`

  This function returns all the readers that have not been reserved by `attachReader`.

- `getAttachedReaders`

  This function returns all the readers that have been previously reserved by `attachReader`.

- `unregisterAllModules`

  This function unregisters all modules currently registered. See 2.)

- `isSmartCardInserted`

  This function returns wheter a smartcard is inserted for the given ReaderModuleDescription.

- `getFirstAdditionallyReader`

  Additionally readers can be pre-configured. An additionally reader is a reader which is not directly avaible to the system but may will be accessible in the future such as a e.g. bluetooth reader. So even for those readers which are currently not present providers can be created.

  Additionally readers are linked to each module i.e. when two additionally readers are pre-configured and three modules registered there are at least six ReaderModuleDescriptions available at the ReaderManager.

  This function returns the very first of it where the order depends on the registration order of the modules and the configuration order of the additionally readers.

## 3.2 ModuleManager / P11ModuleManager

The ModuleManager shall[15] be responsible for managing different modules. The instance's tasks are:

- `getDefaultModules`

[15] the ModuleManager is an interface

The ModuleManager shall be able to at least return one default module i.e. it shall be shipped with at least one driver able for communicating with smartcard-readers.

- `getDefaultAdditionalReaders`

  See 3.1 for description of additionally readers. This function shall return the default ones.

- `getDefaultAdditionalReaderInfo`

  Any reader contains ReaderInfo. In order to support that feature for additional (unconnected) readers this function shall return the ReaderInfo for an additionally reader.

- `registerModule`

  Readers connected to the system shall be addressable by different modules / drivers. This functions registers a new module and shall enable communication with supported readers.

- `unregisterModule`

  This function shall (detach and) remove all readers addressable by that module.

- `getRegisteredModule`

  Shall return all registered modules.

- `getCompatibleReaders`

  Shall return all readers which are addressable by the given module.

- `existsReader`

  Shall check whether the reader exists / is addressable.

- `isAdditionallyReader`

  Shall check whether the reader is an additionally reader.

- `getReaderState`

  Shall check the state of the reader.

- `getReaderInfo`

  Shall return the reader info.

- `getModuleInfo`

  Shall return the module info.

- `getSmartcard`

  Shall create a smartcard instance.

- `waitForModuleEvent`

  Shall wait until an event happens like – reader added, reader removed, smartcard inserted into reader, smartcard removed out of reader.

The ModuleManager interface is implemented by the P11ModuleManager which uses the cgPKCS#11 in order to communicate with smartcards. The cgPKCS#11 can be seen as a single module and is used as default module by the P11ModuleManager.

## 3.3 SmartCardAuthProviderManager

The SmartCardAuthProviderManager offers the opportunity to do all the steps 1. – 5. shown in 3.1 automatically. The offered services are:

- `registerSmartCardProvider`

  A SmartCardAuthProvider is automatically registered to the system. First the provider is created and then registered by `Security.addProvider`.

- `unregisterSmartCardProvider`

  First the reader the provider is listening to is detached. Then the provider is unregisted at the system by calling `Security.removeProvider`. On detach all objects returned by the provider are signaled that the reader was detached. These objects are no more useable. Accessing such an object will fail with an exception.

- `registerAllAvailableProviders`

  Creates and registers SmartCardAuthProviders for each ReaderModuleDescription which is available and has not been attached in the system.

- `getRegisteredSmartCardProviders`

  Returns all available SmartCardAuthProviders which where registered by the Smart-CardAuthProviderManager. SmartCardAuthProviders created and registered manually are not returned.

- `getAdditionallyRegisteredSmartCardProviders`

  Returns a subset of providers which are based on additionally readers. This subset is al-so included in `getAdditionallyRegisteredSmartCardProviders`

- `getRegisteredSmartCardProvidersWithSmartCardInserted`

  Returns all SmartCardAuthProviders which are ready to use / have a smartcard insert-ed.

Furthermore the SmartCardAuthProviderManager implements the SystemEventHandler-interface and selfregisteres[16] at the ReaderManager which makes it capable of receiving read-er-added and reader-removed events.

- `onSystemEventReaderAdded`

  When a new reader is added to the system and the SmartCardAuthProviderManager was called once before a new provider is automatically added to the system.

[16] as soon as getInstance was called once. Otherwise the SmartCardAuthProviderManager stays uninitialized and does not react on events.

- `onSystemEventReaderRemoved`

  When a reader was removed and the SmartCardAuthProviderManager was called once before the provider is unregistered from the system.

# 4 Differences JCE

As we mentioned earlier the JCE is not intended for hardware usage. Even when we achieved the usecase in combining JCE providers with smartcards not all features of the standard JCE can be satisfied. Also there are some deviations regarding the expected behavior compared to the standard JCE. Within this chapter we clarify which functions / features do not behave like they would be expected.

## 4.1 Objects

Whenever a key is created or returned by one of our JCE classes actually this is not directly the key itself. Smartcards internally operate with keyIDs – or more general – handles and will not return the complete key but rather the handle. Private keys / Secret keys will even never leave the card since they are stored in the smartcard's protected area. Due to this reason our JCE classes always return a wrapped key handle and never the binary key data.

The same applies to certificates and data objects. Whenever an object is retrieved this is actually not the binary data of the object but rather a handle for accessing it.

In other words – objects received from our JCE classes are not useable with other JCE classes of different providers!

You even can **not** use objects from one SmartCardAuthProvider bilaterally with another one!

For instance someone has a private key handle received from SmartCardAuthProvider A's keystore. Furthermore there is a cipher-object created and returned from SmartCardAuthProvider B trying to use to encrypt with the key handle from A. This does not work since both objects operate on two different cards.

Certificates and public keys are able for being extracted in ASN1 raw data by `getEncoded` function.

## 4.2 KeyStore

As we learned in 4.1 all objects returned by our JCE classes are wrapped object handles. This concept also applies for outgoing objects of our keystore implementation while importing works as usual.

Before operating our keystore has to be loaded like each usual keystore by calling the `load` function. The first parameter *stream* is ignored while the second – *password* – plays a special role. When no password is given the keystore is loaded in public mode i.e. only public objects will be returned.

Lets clarify this within an example. First someone loads the keystore with both parameters set to null. Afterwards `getKey` is called with an alias usually storing a private and a public key. Since in public mode all private objects are invisible the function will return the public key.

Now we assume calling `load` carrying null as first parameter and the smartcard's PIN as the second one. This loads our keystore in private mode i.e. all objects are visible. The same `get-Key` call now results in returning the private key.

Now someone could have the idea retrieving both keys – the public and the private key – by calling `getKey` in different modes. This would of course work for usual keystores but not for ours. As we learned in 4.1 our keys are implemented via handles. As soon as the `load` function is called all previously received handles are no more guilty. In order to retrieve both keys someone first has to call the `load` function with the correct PIN. Afterwards the private key can be received by calling the `getKey` function. Getting the public key needs two steps. First call the `getCertificate` function. Then use the *certificate instance* and call it's `getPublicKey` function.

The keystore is the only object which permanently connects to the smartcard and must be loaded in private mode in order to use private cryptographic operations from other SPI objects like Cipher and Signature.

Before using the SmartCardExtension ensure unloading the keystore from private- to public mode! Otherwise the SmartCardExtension's functions being able to throw SmartCardInUseEx-ception will return with that error.

When a keystore was previously loaded and the smartcard is removed and reinserted the key-store will automatically reload in public mode.

### 4.3 KeyPairGenerator / KeyGenerators

As shown in 4.2 only the keystore opens a permanent connection to the smartcard. All other SPI objects generated from a SmartCardAuthProvider just open a temporary connection. This also applies for KeyGenerator / KeyPairGenerators.

When a Key- /KeyPairGenerator generates a key and the keystore was not yet loaded or loaded in public mode retrieved keys do not contain valid handles. In this case you have to load the keystore in private mode and receive the handle(s) again with the correct alias! You can extract the alias from invalid objects by calling `getID` after you casted them to SmartCardObject.

When a Key- /KeyPairGenerator generates a key and the keystore was loaded in private mode the handles retrieved by the generateKey / generateKeyPair stay valid as long as the keystore is reloaded or the smartcard is removed.

When the keystore is not loaded in private mode KeyGenrators or KeyPairGenerators do not return valid object handles!

### 4.4 Configuration

Our SmartCardAuthProviders and even some parts of the SDKs where designed to be configurable. Currently we support to disable specific algorithms and allow PIN caching. In the following we will discuss those features.

#### 4.4.1 Disable Algorithms

We learned that each SmartCardAuthProvider offers the cryptographic algorithms available on the currently inserted smartcard as services. For instance some of those algorithms could be outdated, broken or not recommended for some usecases. For this scenario we implemented the “DISABLED_ALGORITHMS” feature.

In order to disable some of the algorithms a provider would usually offer[17] someone can use the “DISABLED_ALGORITHMS”-property during the instantiation of the SmartCardAuthProvider. The property shall carry all the algorithms which shall not be offered by the provider separated by a “;”. The following example shows how some of the services can be disabled:

```
Properties prop = new Properties();
String disabledAlgorithms = "";
disabledAlgorithms += "RSA/NONE/PKCS1Padding;";
disabledAlgorithms += "DES3";

p.setProperty("DISABLED_ALGORITHMS", disabledAlgorithms);

SmartCardAuthProvider provider = new SmartCardAuthProvider(prop,
                                   reader, extension, factory);
```

#### 4.4.2 PIN Caching

So far we have learned that there are several situations where a prompt to the user is neccessary in order to get the smartcard’s PIN. Those situations are always processes doing private smartcard operations i.e. using private- or secret-keys. Since it would be quite annoying for the user to re-enter the PIN all the times we implemented PIN caching. In some situations PIN caching is mandatory since otherwise functional expectations would be violated. In some others it is optional and can be prohibited. Lets have a look at those situations:

- During a `login / logout` session[18] the PIN is asked once and always stays cached. This situation can’t be prohibited since it would violate the expectations a user might have from previous uses of the JCE.
- As soon as a keystore is successfully loaded in private mode by using the `load`-function the PIN stays cached as long as the load-function is called again or the smartcard is re-

[17] or to be more precise a smartcard would usually offer
[18] those functions are offerd by AuthProviders

moved. PIN caching within keystores is always independent from `login / logout sessions` and can not be prohibited.

- During the instantiation of a reader PIN caching can be allowed or not. When PIN caching was activated and the PIN was once entered correctly the PIN stays cached as long as the smartcard is removed. When a PIN was cached and `login` is called the user will not be asked for the PIN.

  When no PIN caching and no `login` session is active the user is always prompted for the PIN as soon as a private cryptographic operation shall be executed.

  A SmartCardAuthProvider instantiated with this reader will use PIN caching for it's complete lifetime!

## 5 Loading our JCE

After we have learned everything we need to know we now can focus on the concepts loading our JCE. Therefore we have to distinguish between standard Java platforms and special Java implementations like the one for Android.

On standard Java systems we ship a simple jar-file which can be used as library to dynamically load our JCE. There is also a static installation method that enables our JCE being present in the whole Java Runtime Environment.

Contrary to the standard Java system there is no static installation available for Android. Here developers would always have to link our library to their program in order to use our JCE. Due to this reason we developed our own loader concept for Android which enables our JCE being preinstalled once per device and avoids multi-linking for serveral apps. Lets have a deeper look at those concepts.

### 5.1 Loading our JCE on Windows / Linux – standard Oracle Java

On standard Java / Oracle platforms there are two ways of installing our SmartCardAuthProvider. We whether have the choice to install the SmartCardAuthProviders statically to our system or loading it dynamically.

#### 5.1.1 Static installation

In order to install the static provider there are two steps necessary. First we have to install[19] the jar-library to the “**Java**\lib\ext” folder where **Java** points to the location of your installed JRE. Afterwards we have to edit the “java.security” file stored in “**Java**\lib\security” and add the following line where **X** is a number starting at 1:

```
security.provider.X=com.certgate.jce.staticinstall.StaticSmartCardAuthProvider
```

Higher numbers mean lower priority. The priority can be seen as a order for asking the providers if they support a demanded algorithm when it is equal where the algorithm comes from. We suggest using 1 as priority and increment all other providers previously listed in that file. After a restart of the JRE each Java program will be able to use our provider. The provider - if necessary - can be retrieved by using the following lines of code:

```
AuthProvider p = Security.getProvider("CERTGATE SmartCard Provider");
```

When this line is executed the first time[20] the StaticSmartCardAuthProvider is initialized and will use the first reader which has a smartcard present. When no reader is found or no reader has a smartcard inserted the reader returned by `getFirstAdditionallyReader` is used (see 3.1 for more information).

When it is intended to load the providers dynamically to the system neither install the jar library to the “Java\lib\ext” folder nor insert the entry to the “java.security” file!

[19] just copy
[20] within a program

### 5.1.2 Dynamic loading with SmartCardAuthProviderManager

When our jar-library is linked to your code it is recommended to use the SmartCardAuthProviderManager to initialize all providers i.e. for each smartcard reader found within your system an own provider is created. This can be done with:

```
SmartCardAuthProviderManager.getInstance().registerAllAvailableProviders();
```

The registered SmartCardAuthProviders can e.g. be retrieved by:

```
SmartCardAuthProviderManager.getInstance().getRegisteredSmartCardProviders();
```

### 5.1.3 Dynamic loading with our SDK / library

The SmartCardAuthProviderManager gives the opportunity to register single or even all providers which are available[21]. This can be also done manually by our SDK. For doing so you have to execute the following steps:

1. Register the P11ModuleManager at the ReaderManager

```
ModuleManager moduleManager = P11ModuleManager.getInstance();
ReaderManager.getInstance().setModuleManager(moduleManager);
```

2. Cycle through all available ReaderModuleDescriptions, create a SmartCardAuthProvider and add it to the system

```
Vector<ReaderModuleDescription> descriptions = ReaderManag-
er.getInstance().getDetachedReaders();


for (ReaderModuleDescription d : descriptions)
{
        Reader r = ReaderManager.getInstance().attachReader(true, d);
        SmartCardExtension ext = new SmartCardExtensionImpl(r);
        SmartCardCryptoFactory fac = new SmartCardCryptoFactoryImpl(r);
        SmartCardAuthProvider provider = new SmartCardAuthProvider(null, r, ext,
        fac);
        Security.addProvider(provider);
}
```

## 5.2 Loading our JCE on Android

Since there is no static way for installing SmartCardAuthProviders to the whole[22] Android system usually a developer would need to link our library to his application binary. When we e.g. release a new bugfixed version of our library it can be a quite annoying task to build up the own application again just for re-linking the third party library. Due to this reasons and the Java sandbox limitations we developed an own loader concept allowing to install our cgJCE once per device and then use it from all applications. Indeed it is necessary to link another jar library but

[21] in fact this depends on the readers available
[22] i.e. that all applications can use the providers

this one is likely not intended for changing a lot and just a very small fraction regarding the filesize compared with our complete cgJCE library. This light weighted library is called the “loader”.

The loader is responsible for loading the pre-installed cgJCE into the context of the appropriate application and automatically[23] runs the necessary initialization procedures. Afterwards the SmartCardAuthProvider can be easily used as Java standard AuthProvider.

Furthermore the loader brings some wrapper classes. Since loading our cgJCE with the loader disables the use of our JCE SDK classes[24] we have written wrapper classes using reflection calls to nearly simulate our JCE SDK.

The SmartCardAuthProviderWrapper itself is no provider class but offers nearly the same functions as the SmartCardAuthProvider. For retreiving the provider – for e.g. direct getInstance calls as shown in 1.3 – `getProvider` has to be called.

SmartCardAuthProviders would usually offer to register observers. This is not available for the SmartCardAuthProviderWrapper but there is a replacement instead. The wrapper offers to register a SmartCardAuthProviderEventCallbackHandler which also forwards SmartCardEvents and can be registered or unregistered via `registerSmartCardEventHandler` and `unregisterSmartCardEventHandler`. As SmartCardExtension it returns a SmartCardExtension-Wrapper via `getSmartCardExtension`.

The SmartCardExtensionWrapper has almost the same functionality as the SmartCardExtension but has some minor deviations. Instead of a parameterized `getRemainingAttempts` function there are `getRemainingAttemptsUser` and `getRemainingAttemptsAdmin` instead. The same concept applies to the `verifyPIN` and `changePIN` functions.

When using the loader the JCE is initialized with the following steps:

1. Implement the LoaderCallback

   The LoaderCallback-interface contains a function which is called when the loader has finished its job in order to signal that the providers are ready to use.

2. Call ProviderLoader.load function

   This function starts our cgJCE providers being loaded asynchronously.

3. Wait until LoaderCallback is called before using the cgJCE providers.

   When the LoaderCallback receives the final `onLoaded` call our cgJCE providers were loaded and are now ready to use.

For easier integration we also implemented an AsyncTask called the InitJCETask. It offers to initialize the whole cgJCE with just one line of code by calling

`new InitJCETask(applicationContext, null, true).execute();`.

The above line of code does not supply a OnTaskFinishedCallback (null) whereas it would return the opertions result. For more information see the loader whitepaper. We recommend to al-

[23] if wished
[24] due to class loader issues instances of SmartCardAuthProvider can not be casted to the JCE SDK classes or interfaces

ways use the LoaderCallback or the OnTaskFinishedCallback since they signal that the loading process has finished.

When the loading-job is finished the providers are registered to the system and can be used as usual.

## 6 Examples