An extensible client platform for eID, signatures and more

Tobias Wich[1] · Moritz Horsch[2] · Dirk Petrautzki[3] · Johannes Schmölz[1]
Detlef Hühnlein[1] · Thomas Wieland[3] · Simon Potzernheim[3]

[1] ecsec GmbH, Sudetenstraße 16, 96247 Michelau,
{tobias.wich,johannes.schmoelz,detlef.huehnlein}@ecsec.de

[2] TU Darmstadt, Hochschulstraße 10, 64289 Darmstadt,
horsch@cdc.informatik.tu-darmstadt.de

[3] Hochschule Coburg, Friedrich-Streib-Str. 2, 96450 Coburg
{petrautzki,thomas.wieland,potzernheim}@hs-coburg.de

Abstract: The present paper introduces an extensible client platform, which can be used for eID, electronic signatures and many more smart card enabled applications.

1 Introduction

Against the background of various electronic identity (eID) card projects around the globe there have been numerous initiatives in the area of research, development and standardization of eID cards, smart card middleware components and related services. Nevertheless, whenever a new eID project emerges, new software is often developed from scratch. This happens despite all similarities of the systems and requirements. The present paper introduces a modular and extensible client platform, which can be extended for the use with eID, electronic signatures and many other smart card related applications. The design of this extensible platform is a refinement of the architecture of the Open eCard App [HPS+12], which in turn is based on the eCard-API-Framework (BSI-TR-03112) and its integrated international standards, such as ISO/IEC 24727 [ISO08a, ISO08b] and OASIS Digital Signature Services [Dre07]. The design and implementation of the platform has been based on previous work [Hor11, Pet11] and realized as a joint effort of industrial and academic experts within different projects, such as ID4health[1], SkIDentity[2], FutureID[3], and Open eCard[4].

The remainder of the paper is structured as follows: Section 2 provides an overview of the proposed client platform. Section 3 describes the extension points of the client platform. Section 4 presents the design of the add-on framework and its mechanisms to dynamically load missing functionality. Section 5 closes the paper with an outlook on the next steps and future development.

[1] See http://www.id4health.de.
[2] See http://www.skidentity.de.
[3] See http://www.futureid.eu.
[4] See http://www.openecard.org.

2 Overview of the extensible architecture

The proposed client platform is aligned to the eCard-API-Framework (BSI-TR-03112) which integrates major international standards (e.g. [ISO08a, ISO08b, Dre07]) in order to provide a common and homogeneous interface for a standardized usage of different smart cards. The architecture depicted in figure 1 is designed to separate the overall functionality of an eID application in suitable components, reuse of common modules and to provide means for expandability. The modular approach and the platform-independent implementation of the core modules in Java allow the Open eCard App to be used on various computing platforms, such as desktop systems running on Windows, Linux and Mac OS X as well as mobile systems running Android for example.

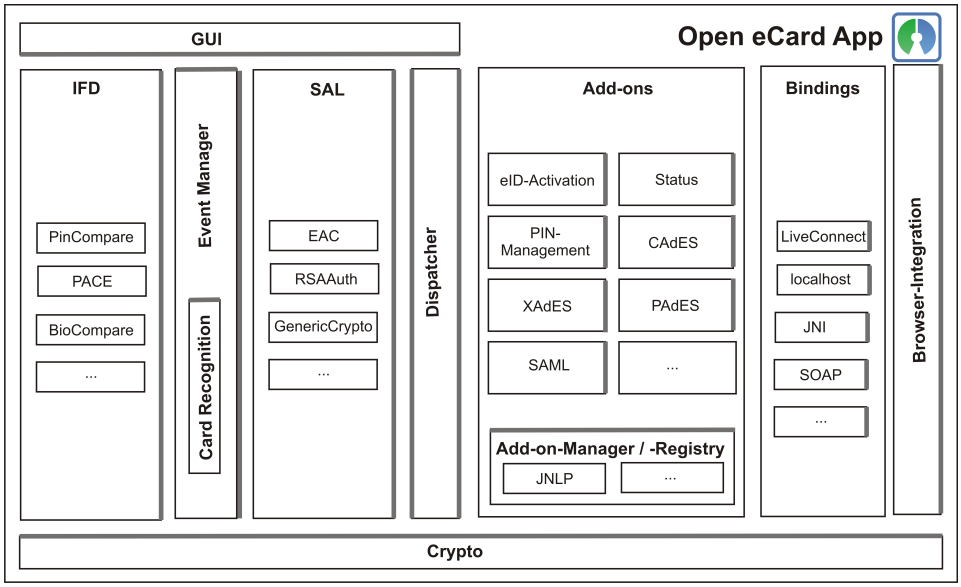

Figure 1: Extensible architecture of eID-client-platform

The components of the extensible eID platform are described in the following:

Interface Device (IFD) The IFD provides a generalized interface for communication with arbitrary card terminals and smart cards according to ISO/IEC 24727-4 [Fed12b, ISO08b]. It abstracts from specific interfaces and physical properties like contactless interfaces. Furthermore it provides expandability for the integration of secure channel establishment protocols which protect the communication between the eID client and the smart card.

Event Manager The Event Manager is responsible for managing card terminal and card events. It periodically asks the IFD for the current status of terminals and cards and determines changes like the connection and disconnection of card terminals and smart cards by comparing status reports over different time periods. Furthermore, the Event Manager performs the card recognition to determine the type and the functionality of the card as explained in section 3.2.

Service Access Layer (SAL) The SAL provides a generic interface for common smart card services according to ISO/IEC 24727-3 [ISO08a, Fed12c], which allows to manage data that is stored on the card for example. In detail, the SAL comprises Connection Services, Card Application Services, Named Data Services, Crypto Services, Differential Identity Services and means for accessing card application services in an authorized manner. Furthermore, the SAL provides an interface for integrating arbitrary authentication protocols, which provides expandability without changing other parts of the implementation (see section 3.3).

Dispatcher The Dispatcher provides a centralized communication component for handling incoming and outgoing messages.

Add-ons Add-ons provide additional functionality to the basic eID platform. Signature functionality and PIN Management, for instance, can be realised as an add-on to provide additional functionality and allow customisation. The Add-on Registry provides a service to search and retrieve add-ons. Such a registry can, e.g., be realised based on the Java Network Lauching Protocol (JNLP) [Her11]. After an add-on is loaded, the Add-on Manager takes over the management of the add-on instances and enforces the compliance with the defined security policy by a sandbox mechanism.

Bindings The Binding component comprises modules for message transport. The components implement a particular protocol like HTTP or SOAP to transmit messages from external applications to the client.

Crypto The Crypto component encapsulates common cryptographic functions, which are used by other components. It is based on the Bouncy Castle crypto library [The] which makes it easy to port it to platforms without support for the full Java Cryptography Architecture (JCA) [Orab], such as Android for example.

Graphical User Interface (GUI) The GUI component provides an abstract framework to develop user interfaces and interactions. This allows the exchange of GUI implementations and therefore providing platform-specific GUI implementations, while leaving the other components unchanged.

3 Extension Points

This section describes the extension mechanisms of the eID platform, which allows enhancing the application's functionality on different levels. In detail, it allows adding arbitrary protocols to the IFD and SAL component, supporting various card terminals and smart cards as well as enhancing the application functionality by add-ons.

In general, we use the term *add-on* to describe a software component which enhances the functionality of the basic eID platform. Furthermore, we distinguish between *plug-in* and *extension*.

Plug-ins depend on the context in which the user uses the application. Performing an authentication to a service using a particular smart card, for instance, requires a plug-in which is capable of providing such functionality. Subsequently, plug-ins require a communication with bindings to interact with external applications and services. Furthermore, we distinguish between IFD, SAL and application plug-ins, which are described in detail in the following sections.

Extensions are independent from the context. Moreover, they are directly integrated into the user interface and can be executed by the user. For instance, an add-on that provides a PIN change functionality for smart cards is classified as an extension.

3.1 IFD Plug-ins

The IFD provides a generalized interface for communication with arbitrary smart cards and card terminals. It also can be extended by plug-ins, i.e. protocols which perform a user authentication and/or establish a secure channel between a smart card and a card terminal to protect the communication from being eavesdropped.

Each protocol must have a unique identifier in form of a URI. The URI must be associated with the actual implementation as described in section 4.1. In addition, each protocol plug-in must implement the IFD Protocol Interface and must define protocol-specific `AuthenticationProtocolData` used in the `EstablishChannel` call[5] and corresponding response message.

The Password Authenticated Connections Establishment (PACE) protocol is one example of a protocol which is executed in the IFD layer. It is a password-based protocol that performs a user authentication, based on a PIN, and establishes a Secure Messaging channel (cf. [ISO]) to ensure that only the legitimate user can use the card and that the communication is encrypted and integrity protected. The details of the PACE-protocol are specified in BSI-TR-03110 [Fed12a].

ISO/IEC 24727-4 Interface An IFD-protocol will be executed by an `EstablishChannel` IFD API call. The function call includes a `SlotHandle` to address an established con-

[5] See `http://ws.openecard.org/schema/ISOIFD-Extension.wsdl`.

nection and a protocol-specific extended `AuthenticationProtocolData` element.

Java Interface The `IFDProtocol` interface defines functions for IFD protocols (cf. figure 2). Each protocol must implement the `establish` function that executes the protocol. The function gets as input an `EstablishChannel` request that includes protocol-specific data. The parts which are necessary to communicate with the eID application are handed over to the implementation in the `init` function. The context `ctx` contains the user consent implementation, which allows a protocol to perform user interaction, e.g. to receive PIN entries. In addition, the interface specifies the functions `applySM` and `removeSM` to apply and remove Secure Messaging. The `establish` function returns an `EstablishChannelResponse`. The `IFDProtocolFactory` provides a factory class which also proxies the protocol interface. The usage of the class name decouples the actual loading of the class and prevents execution of plug-in code outside of the sandbox.

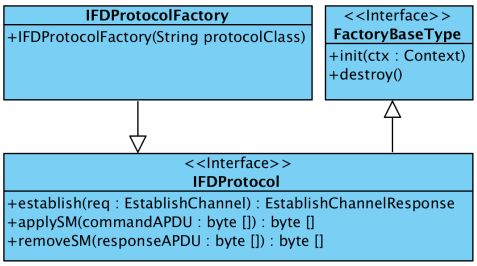

Figure 2: IFD-Protocol-Interface UML diagram

3.2 CardInfo Files

In order to support a broad range of smart cards, the eID platform supports CardInfo files (CIF) according to [ISO08a]. A CIF is an XML file that describes the data structure and the functionality of smart cards in a standardized way. Besides the abstract definition of the card, it also contains information how to recognize the specific card type.

To provide a sophisticated recognition of smart cards it is prudent engineering practice to construct a decision tree based on the set of available CIFs (cf. [Wic11]). While the construction of the tree could be performed by the eID application on demand, this task is better performed by a central CardInfo repository, which performs the construction and only distributes the decision tree (cf. [Fed12e]). To make the eID application capable of recognizing new smart cards, only the corresponding CIFs and an updated version of the decision tree have to be added.

3.3 SAL Plug-ins

The SAL provides a generic interface for common smart card services comprising different services, such as the Crypto Services and the Differential Identity Services. The SAL can be extended by plug-ins, which provide implementations of protocols for the Crypto Services and the Differential Identity Services [Fed12d, Section 4] as required for the use of specific signature cards and electronic identity cards for example.

The plug-in concept is quite similar to the one that is used in the IFD layer (cf. section 3.1). Each SAL protocol must define a unique identifier (URI). In contrast to the IFD, the SAL supports protocols with multiple steps and allows the definition of more sophisticated user interfaces including a sequence of interaction steps to represent information dialogues and general user consents.

One example of a SAL protocol is the Extended Access Control (EAC) protocol which is used for the authentication with the german eID card. The protocol-specific messages are specified in [Fed12d, Section 4.6].

ISO/IEC 24727-3 Interface A protocol execution is triggered by invoking an action within the Crypto Services or Differential Identity Services API (cf. [ISO08b, section 3.5 and 3.6]). The functions includes an `AuthenticationProtocolData` element, which is extended in a protocol-specific manner.

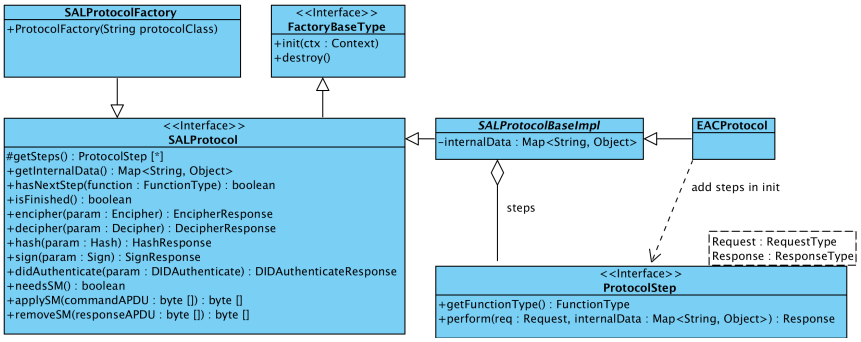

Figure 3: SAL Protocol Interface UML diagram

Java Interface Each protocol must implement the `SALProtocol` interface. A convenience abstraction which works for the common protocol flows is realized in the class `SALProtocolBaseImpl`. An internal data object is used for the exchange of data between the different protocol steps. A protocol step is represented by the `ProtocolStep` interface which defines a `FunctionType` defining a Crypto or Differential Identity Service and a `perform` function to execute the step. The control of the application flow is performed automatically after being triggered by incoming Crypto or Differential Identity Service requests. The instantiation is performed through the `SALProtocolFactory`

similar to the IFD protocols explained in section 3.1.

3.4 Application Plug-ins

Application plug-ins provide a mechanism to add additional functionality to the eID application with which external applications can communicate. Depending on the type of the underlying binding, this could be a browser, a PKCS#11 module or even a remote application.

Protocol bindings realize the connection to the external world. While a broad variety of transport protocols could be supported, the most obvious choices are HTTP and SOAP, as they are stipulated by [Fed12d, Section 3.2] for example. Given the properties of the activation mechanism, HTTP and SOAP, as well as similar transport protocols, the abstract requirements for a protocol binding are given as follows: A protocol binding must support

1. a request-response semantic,
2. a mapping mechanism to identify the appropriate plug-in for a request,
3. messages comprising a body, named parameters and attachments,
4. an error delivery mechanism, and
5. a redirect semantic.

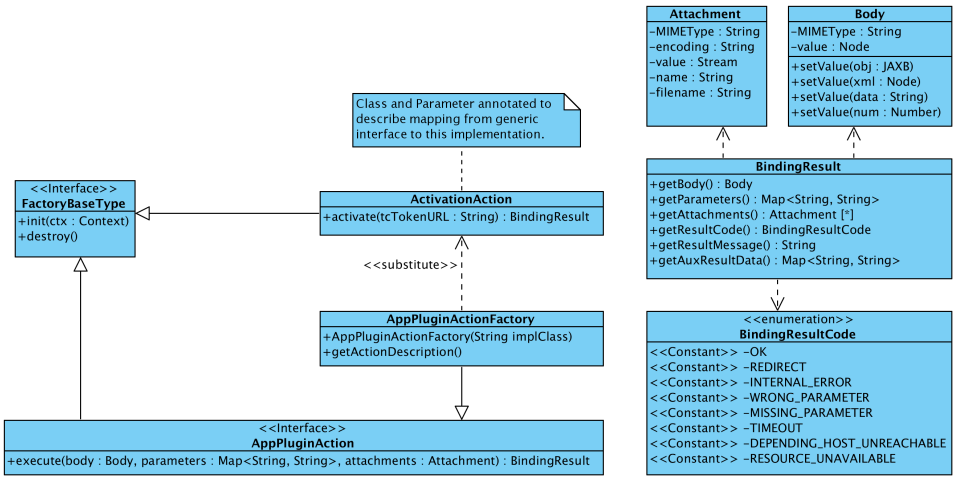

Figure 4: Application-Plug-in-Interface UML diagram

Figure 4 shows the interfaces and the data model of the application plug-ins. On the plug-in side it is easy to see that all properties are fulfilled. The interface `AppPluginAction` provides an `execute` function with a strict data oriented semantic, meaning no callback

code can be injected for asynchronous responses. The second property is fulfilled by a named identification of the action which is discussed in detail in section 4.1. The data structures for body and attachments can be seen on the right side of the diagram. Named parameters have no particular ordering and no special type so a string of characters can represent either key and value. These three elements form the input parameters of said `execute` function and are part of the result. The body element carries exactly one DOM node. This representation has the advantage that it can carry strings as well as more complex XML elements. That makes it suitable to provide the content of a SOAP body, JSON data converted to an XML representation or string based entities. Attachments are included to transport binary files. The data structure is modelled to support the most important features of MIME messages such as Multipart MIME messages (cf. [FB96, Section 5.1]). The fourth and fifth requirement are fulfilled by providing predefined response codes and auxiliary data for the specific type of action. In case of an error, a localized message may be attached to the result. A redirect needs a redirect target value in the auxiliary data. It is up to the receiving application how to interpret and perform the redirect. The open character of the auxiliary data makes it easy to add new capabilities for further use cases to the bindings without the need to change the Application Binary Interface (ABI) of the interface.

While different transport protocols (e.g. *HTTP on localhost*, *LiveConnect*, *SOAP*) may be used to realize bindings for the different add-ons (e.g. *eID Activation*, *Status*, *Signature PKCS#11*) we will explain the general concept using the example of a signature plug-in with the localhost binding according to BSI-TR-03112-7 [Fed12d] in the following.

Given the containers parameters, body and attachments, the plug-in can define its interface. A signature plug-in can be modelled in two ways. Either via an RPC-style interface where the properties of the plug-in are transported in parameters, or via an OASIS DSS [Dre07] like interface where the properties are transported as a structured object in the body.

Suppose the variant with the simple parameters is used, the following HTTP request (listing 1) and response (listing 2) messages can be modelled. The simple model might be desirable when the signature functionality is limited to a few base cases and thus the full OASIS DSS capabilities are not needed.

```
1  POST /signature?signatureType=XAdES&cardType=... HTTP/1.1
2
3  Content-Type: multipart/form-data; boundary=AaB03x
4
5  --AaB03x
6  Content-Disposition: form-data; name="files"; filename="data.xml"
7  Content-Type: text/xml
8
9  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
10 <Data xmlns="myns">to be signed</Data>
11 --AaB03x--
```

Listing 1: RPC-Style Sign Request

In order to sign to a document, at least the signature type and the data to be signed is required. To take away the responsibility of the user to select a signing entity, e.g. a specific

```
HTTP/1.1 200 OK

Content-Type: Multipart/mixed; boundary=AaB03x

--AaB03x
Content-Disposition: attachment; name="files"; filename="data.xml"
Content-Type: text/xml

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>...</SignedInfo>
  <SignatureValue>...</SignatureValue>
  <KeyInfo>...</KeyInfo>
  <Object Id="dataId">
    <Data xmlns="myns">to be signed</Data>
  </Object>
</Signature>
--AaB03x--
```

Listing 2: RPC-Style Sign Response

smart card, this information may be given as well. The parameters `signatureType` and `cardType` as given in listing 1 line 1 represent the latter choices. The document itself is included as a named part shown in line 5 ff. in the HTTP body. The representation as `multipart/form-data` according to [RLJ99, Section 17] has been chosen so that typical browsers can issue requests easily. Named parts can be matched to the attachment type of the interface as well as to the body. To resolve the ambiguity, the body can simply be an attachment with a special name value, but other schemes may be allowed as well to capture other communication patterns.

```
POST /signature?cardType=... HTTP/1.1

Content-Type: application/xml
Content-Length: ...

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<dss:SignRequest xmlns:dss="urn:oasis:names:tc:dss:1.0:core:schema">
  <dss:InputDocuments>...</dss:InputDocuments>
    <dss:Document>
      <Data xmlns="myns">to be signed</Data>
    </dss:Document>
  <dss:OptionalInputs>
    <dss:SignatureType>urn:ietf:rfc:3275</dss:SignatureType>
  </dss:OptionalInputs>
</dss:SignRequest>
```

Listing 3: OASIS DSS-Style Sign Request

A more sophisticated data exchange for a signature plug-in is shown in listing 3. The example uses OASIS DSS `SignRequest` messages to specify what kind of signature should be performed and what should be signed. The signing entity is chosen as in the previous example. The example also shows that the request is nearly identical to a SOAP

request, so the parameters can be mapped by either the localhost binding or a SOAP binding.

3.5 Application Extensions

Extensions enhance – similar to plug-ins – the basic eID platform and provide additional functionality, but they do not depend on the context in which the eID application is used. Further, extensions are included into the user interface and can be started directly by the user. Similar to application plug-ins, the `AppExtensionAction` interface, as shown in figure 5, contains an `execute` function. However, this function does not have any parameters nor does it have a result. Therefore, it cannot be used with a binding and only be triggered manually.

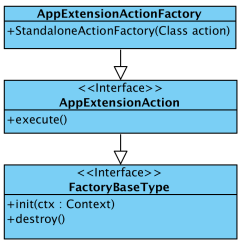

Figure 5: Application Extension Interface UML diagram

4 Add-on Framework

4.1 Add-on Anatomy

Add-ons are described by the data model shown in figure 6. This model is the representation of the XML structure of an add-on's manifest file. It contains general information such as the name, the textual description and configuration entries for changeable settings of the add-on, and its contained actions which represent the interfaces shown in section 3. The settings are saved in an add-on specific storage location and are loaded as Java properties by the add-on framework. Each action has one or more entries which identify it unambiguously. The IFD and SAL protocol plug-ins are identified by their protocol URI, whereas the application extensions and plug-ins are identified by the add-on id and action id, or resource name respectively. A reference to the action class makes it possible for the framework to find and load the implementation dynamically.

Based on the add-on manifest, bundles can be formed which can be integrated into the base application with zero configuration overhead on the user side. The structure of a bundle is largely dictated by the Java archive (JAR) file specification [Oraa]. A single JAR file

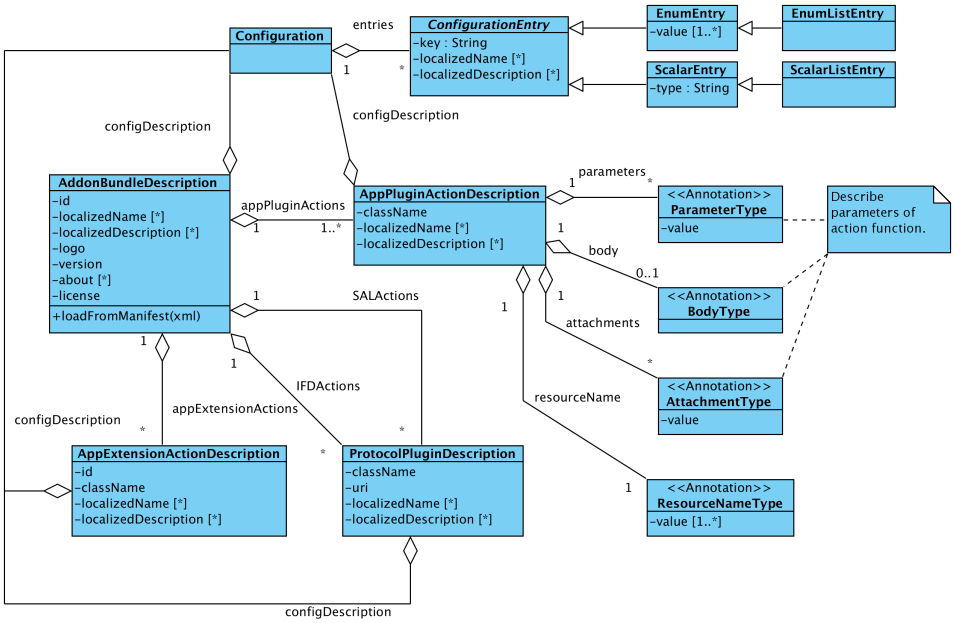

Figure 6: Add-on Description data model UML diagram

bundles the add-on and all dependent libraries. The manifest describing the add-on must be present in the `META-INF` directory with the name `addon.xml`.

4.2 Secure Retrieval and Execution

When a request message is received, the `AddonRegistry` (cf. figure 7) can be consulted to retrieve an applicable add-on for the requested resource. If an applicable add-on is found, it's JAR file will then be downloaded and a `ClassLoader` for subsequently loading the plug-in in a secure manner is returned. The `ClassLoader` will then be used in the factory responsible for the plug-in's type to load the class files.

Furthermore, a custom security policy implementation is set in the JRE and will therefore automatically be consulted every time a security relevant operation (e.g. reflection, classloader creation, filesystem access etc.) is performed. This policy allows to differentiate between signed add-ons, add-ons from a trusted origin and add-ons from an untrusted origin. Depending on the trust level, the add-ons may be granted different privileges.

By the use of privileged actions in a `AccessControler.doPrivileged()` call, trusted add-ons are permitted to call functions of the eID application that themselves do security relevant operations which the add-on would otherwise not have the appropriate rights for and therefore would fail.

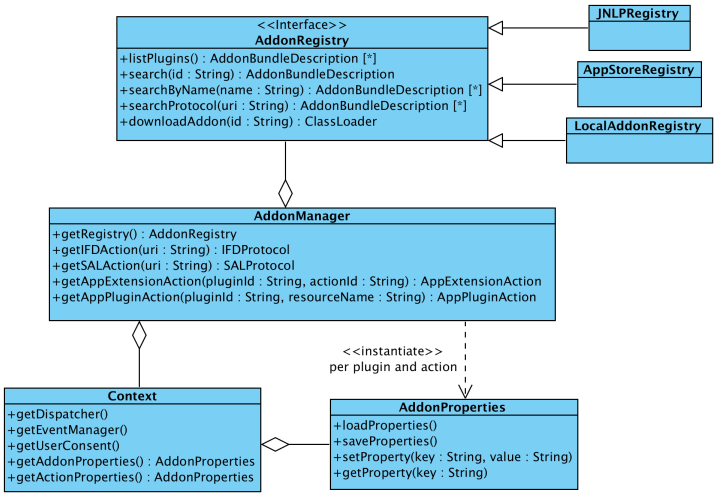

Figure 7: Plug-in Manager and Registry UML diagram

5 Conclusion

The new add-on mechanism of the eID application proposed in the present paper provides an extensible framework which makes it easy to build tailormade eID and similar smart card based applications without re-developing basic functionality again from scratch. The proposed platform provides a set of well defined extension points and the initially provided modules ensure that existing installations can be utilized without modifications. With an App-Store like distribution method, it will be easy for third party vendors to provide their own add-ons. Paired with restrictive security measures, the App-Store model does not sacrifice the security and privacy of the user.

References

[Dre07] Stefan Drees. Digital Signature Service Core Protocols, Elements, and Bindings, Version 1.0. OASIS Standard, 2007. http://docs.oasis-open.org/dss/v1.0/oasis-dss-core-spec-v1.0-os.pdf.

[FB96] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046, November 1996. https://www.ietf.org/rfc/rfc2046.txt.

[Fed12a] Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik). Advanced Security Mechanism for Machine Readable Travel Documents - Extended Access Control (EAC), Password Authenticated Connection Establishment (PACE), and Restricted Identification (RI). Technical Directive (BSI-TR-03110), Version 2.10, 2012. http://docs.ecsec.de/BSI-TR-03110.

[Fed12b] Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik). eCard-API-Framework – IFD-Interface. Technical Directive (BSI-TR-03112), Version 1.1.2, Part 6, 2012. http://docs.ecsec.de/BSI-TR-03112-6.

[Fed12c] Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik). eCard-API-Framework – ISO24727-3-Interface. Technical Directive (BSI-TR-03112), Version 1.1.2, Part 4, 2012. http://docs.ecsec.de/BSI-TR-03112-4.

[Fed12d] Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik). eCard-API-Framework – Protocols. Technical Directive (BSI-TR-03112), Version 1.1.2, Part 7, 2012. http://docs.ecsec.de/BSI-TR-03112-7.

[Fed12e] Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik). eCard-API-Framework – Support-Interface. Technical Directive (BSI-TR-03112), Version 1.1.2, Part 5, 2012. http://docs.ecsec.de/BSI-TR-03112-5.

[Her11] A. Herrick. JSR 56: Java Network Launching Protocol and API. Maintenance Release 6, 2011. http://jcp.org/en/jsr/detail?id=56.

[Hor11] Moritz Horsch. MONA – Mobile Authentication with the new German eID-card (in German). Master-Thesis, Technische Universität Darmstadt, 2011. http://www.cdc.informatik.tu-darmstadt.de/mona/pubs/201107_MA_Mobile%20Authentisierung%20mit%20dem%20neuen%20Personalausweis%20(MONA).pdf.

[HPS+12] Detlef Hühnlein, Dirk Petrautzki, Johannes Schmölz, Tobias Wich, Moritz Horsch, Thomas Wieland, Jan Eichholz, Alexander Wiesmaier, Johannes Braun, Florian Feldmann, Simon Potzernheim, Jörg Schwenk, Christian Kahlo, Andreas Kühne, and Heiko Veit. On the design and implementation of the Open eCard App. In *Sicherheit 2012*, GI-LNI, 2012. http://subs.emis.de/LNI/Proceedings/Proceedings195/95.pdf.

[ISO] ISO/IEC 7816. Identification cards – Integrated circuit cards – Part 1-15. International Standard.

[ISO08a] ISO/IEC. Identification cards – Integrated circuit cards programming interfaces – Part 3: Application programming interface, ISO/IEC 24727-3. International Standard, 2008.

[ISO08b] ISO/IEC. Identification cards – Integrated circuit cards programming interfaces – Part 4: API Administration, ISO/IEC 24727-4. International Standard, 2008.

[Oraa] Oracle Inc. JAR File Specification. http://docs.oracle.com/javase/6/docs/technotes/guides/jar/jar.html.

[Orab] Oracle Inc. Java Cryptographic Architecture (JCA). http://www.javasoft.com/products/jdk/1.2/docs/guide/security/CryptoSpec.hml.

[Pet11] Dirk Petrautzki. Security of Authentication Procedures for Mobile Devices (in German). Master-Thesis, Hochschule Coburg, 2011.

[RLJ99] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. HTML 4.01 Specification. W3C Recommendation 24 December 1999, 1999. http://www.w3.org/TR/html401/.

[The] The Legion of the Bouncy Castle. Bouncy Castle API. http://www.bouncycastle.org/.

[Wic11] Tobias Wich. Tools for automated utilisation of Smart-Card Descriptions. Master-Thesis, Hochschule Coburg, 2011.