WP31 - Interface Device Service

D31.4 – Test Report

Document Identification	
Date	14.10.2014
Status	Final
Version	1.2

Related SP / WP	SP3 / WP31	**Document Reference**	D31.4
Related Deliverable(s)	D31.1, D31.2, D31.3	**Dissemination Level**	PU
Lead Participant	G&D	**Lead Author**	Frank-Michael Kamm
Contributors	Moritz Horsch (TUD), Frank-Michael Kamm (G&D), David Derler (TUG) Tobias Wich (ECS)	**Reviewers**	Antonio de la Piedra (RU), Paolo Modesti (UNEW)

This document is issued within the frame and for the purpose of the FutureID project. This project has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 318424

Abstract

This deliverable documents the integration and testing of the Interface Device (IFD) Service into the FutureID client. It is built upon the specification done in task 31.2 and the implementation work of task 31.3. The purpose of this deliverable is to document the scope and high-level goals of the unit and system tests for IFD integration and to verify if the specified requirements are covered by these tests. The actual testing will then be done within Work Package (WP) 37 Client Testbed.

The deliverable is structured as follows: In Section 2, the scope of the unit tests are documented while section 3 describes the scope of the system tests. As specific test cases the German Identity Card (Section 3.1) and the Austrian Citizen Card (Section 3.2) have been chosen. These test cases cover the most important aspects of the IFD functionality and can be verified with test cards that are available for the project. Section 4 covers the requirements verification while Section 5 summarizes the main results and conclusions. The bibliography can be found in Section 6.

Unit tests are a tool to assert the functionality of a given module or unit under test. In contrast to integration and system tests, unit tests do not incorporate external elements such as hardware or other modules. In our case, system tests verify the successful integration of the IFD into the entire FutureID client application. The first system test uses the German Identity Card to test multiple functions of the IFD. The second test is based on the Austrian Citizen Card.

Coming back to the initial requirements of the IFD layer which have been specified in D31.1 section 4 focuses on the verification of the IFD requirements. Based on the unit and systems test it is verified whether the specified requirements are fulfilled.

In summary, all mandatory requirements for the IFD layer are covered by the specified unit and system test cases. Concerning the optional requirements, those requirements that are needed for the demo implementation are covered by the test cases as well. The support of a Trusted Platform Module (TPM) is not tested since it has been decided not to pursue TPM support any further within this project. Also, the actual implementation of the IFD is concentrated on Java and thus a C interface is not tested.

Overall, the unit and system tests provide a broad verification of the IFD requirements and demonstrate the functionality of the fundamental design principles of it.

Document Information

History

Version	Date	Author	Changes
0.1	01.07.2014	Moritz Horsch	Created document structure
0.2	28.08.2014	Moritz Horsch	Added section system test German Identity Card
0.3	01.09.2014	Frank-Michael Kamm	Update Introduction, chapter 4.
0.4	04.09.2014	David Derler	Added section system test Austrian Citizen Card
0.5	08.09.2014	Frank-Michael Kamm	Chapter 4 updated.
0.6	11.09.2014	Frank-Michael Kamm	Chapter 4 updated.
0.7	12.09.2014	Tobias Wich	Added Chapter 2
1.0	12.09.2014	Frank-Michael Kamm	Added chapter 5, abstract. Finalised reviewer version.
1.1	22.09.2014	F.-M. Kamm, D. Derler, T. Wich	Integration of reviewer comments (RU)
1.2	14.10.2014	F.-M. Kamm	Integration of reviewer comments (UNEW)

Table of Contents

1. Introduction

1.1 Project Scope

The FutureID project builds a comprehensive, flexible, privacy-aware and ubiquitously usable identity management infrastructure for Europe, which integrates existing eID technology and trust infrastructures, emerging federated identity management services and modern credential technologies to provide a user-centric system for the trustworthy and accountable management of identity claims.

The FutureID infrastructure will provide great benefits to all stakeholders involved in the eID value chain. Users will benefit from the availability of a ubiquitously usable open source eID client that is capable of running on arbitrary desktop PCs, tablets and modern smart phones. FutureID will allow application and service providers to easily integrate their existing services with the FutureID infrastructure, providing them with the benefits from the strong security offered by eIDs without requiring them to make substantial investments.

This will enable service providers to offer this technology to users as an alternative to username/password based systems, providing them with a choice for a more trustworthy, usable and innovative technology. For existing and emerging trust service providers and card issuers FutureID will provide an integrative framework, which eases using their authentication and signature related products across Europe and beyond.

To demonstrate the applicability of the developed technologies and the feasibility of the overall approach, FutureID will develop two pilot applications and is open for additional application services who want to use the innovative FutureID technology

Future ID is a three-year duration project funded by the European Commission Seventh Framework Program (FP7/2007-2013) under grant agreement no. 318424

1.2 Deliverable Scope and Outline

This deliverable documents the integration and testing of the Interface Device (IFD) Service into the FutureID client. It is built upon the specification done in D31.2 and the implementation work of D31.3. The IFD layer is the common interface for external applications to use arbitrary card terminals, smart cards and other hardware tokens. It abstracts from platform dependent interfaces like the Java Smart Card I/O and the transport layer of the OpenMobile API [3]. In case that multiple IFDs are present on one device, the IFD also provides a proxy layer.

The purpose of this deliverable is to document the scope and high-level goals of the unit and system tests for IFD integration and to verify if the specified requirements are covered by these tests. The actual testing will then be done within WP 37. The deliverable is structured accordingly: in Section2 the scope of the unit tests are documented, while Section3 describes the scope of the system tests. As specific test cases the German Identity Card

(Section 3.1) and the Austrian Citizen Card (Section 3.2) have been chosen. These test cases cover the most important aspects of the IFD functionality and can be verified with available test cards. Section4 covers the requirements verification while Section5 summarizes the main results and conclusions. The bibliography can be found in Section 6.

1.3 Terminology

1.3.1 Key Words

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [1].

1.3.2 Abbreviations and Notations

APDU	Application Protocol Data Unit
API	Application Programming Interface
eID	Electronic Identity
IFD	Interface Device
nPA	“neuer Personalausweis”, German eID card
PACE	Password assisted connection establishment protocol
SICCT	Secure Interoperable Chip Card Terminal
SOAP	Simple Object Access Protocol
TPM	Trusted Platform Module

2. Unit Tests

Unit tests are a tool to assert the functionality of a given module or unit under test. In contrast to integration and system tests, unit tests do not incorporate external elements such as hardware or other modules in the test. This is in general a problem with complex pieces of software, because of a possibly high level of coupling between the modules. Besides the general design principle to counter strong coupling it is often unavoidable. In order to test coupled modules, different techniques have emerged such as the utilization of mock objects. Even though mocking widens the test coverage achievable by unit tests, certain tests, especially the ones including hardware and third party libraries, exceed the cost-benefit ratio and are therefore typically implemented in the integration or system tests.

The following sections are structured as follows. Section 2.1 introduces the tools and the environment of the testbed. Section 2.2 contains sections for the different groups of functionality in the IFD code. Each section contains a list of classes which are to be tested and detailed test case descriptions.

2.1 Setting

The main tool for writing unit tests is the TestNG unit test framework [4]. It provides methods for controlling when and how tests are executed. For the unit tests only the `@Test` annotation is needed which marks a single function as a test case. It marks the test for automatic execution in the test phase of Maven [5].

As already mentioned in the previous section, mocking is used to test units which are otherwise coupled with other modules or need hardware to execute properly. The JMockit framework is provided in this setting to ease the creation of mock objects [6].

2.2 Test Cases

2.2.1 Utilities

The Utilities group contains classes with supporting data structures und utility classes especially for the work with PACE.

The table below contains all classes for which test cases exist in the Utilities group and are described in this section.

Class	Group
`org.openecard.common.ifd.PACECapabilities`	Utilities
`org.openecard.common.ifd.anytype.PACEInputType`	Utilities
`org.openecard.common.ifd.anytype.PACEOutputType`	Utilities
`org.openecard.ifd.scio.reader.EstablishPACERequest`	Utilities
`org.openecard.ifd.scio.reader.EstablishPACEResponse`	Utilities
`org.openecard.ifd.scio.reader.ExecutePACERequest`	Utilities
`org.openecard.ifd.scio.reader.ExecutePACEResponse`	Utilities
`org.openecard.ifd.scio.reader.PCSCFeatures`	Utilities
`org.openecard.ifd.scio.reader.PCSCPinModify`	Utilities
`org.openecard.ifd.scio.reader.PCSCPinVerify`	Utilities
`org.openecard.ifd.scio.wrapper.ByteArrayComperator`	Utilities

TEST NO.	U_U1 – PACE Capabilities initialization
DESCRIPTION	An instance of the class `PACECapabilities` is initialized with a byte array capturing the following cases: • Containing all possible PACE features • Containing no features • Null value • Invalid length • Invalid feature codes
TEST SUBJECT	`PACECapabilities (byte[])`
PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	An initialized object is available.
ERROR CASES	A given null value throws a `NullpointerException.` An invalid object raises an `InvalidArgumentException.`

TEST NO.	U_U2 – PACE Capabilities extraction
DESCRIPTION	The `PACECapabilities` class must extract the correct values from a given byte array.

TEST SUBJECT	`PACECapabilities.getCapabilitiy(long)`
	`PACECapabilities.getFeatures()`
	`PACECapabilities.getFeaturesEnum()`

PRECONDITION	Objects from test case U_U1
POSTCONDITION	-
EXPECTED RESULT	The functions under test must yield the correct values.
ERROR CASES	-

TEST NO.	U_U3 – PACE Input Type initialization
DESCRIPTION	An instance of the class `PACEInputType` is initialized with a `DIDAuthenticationDataType` Variable.
TEST SUBJECT	`PACEInputType(DIDAuthenticationDataType)`
PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	An initialized object is available.
ERROR CASES	

TEST NO.	U_U4 – PACE Input Type extraction
DESCRIPTION	The `PACECapabilities` class must extract the correct values from the initialized Object.

TEST SUBJECT	`PACEInputType.getPINID()`
	`PACEInputType.getCHAT()`
	`PACEInputType.getPIN()`
	`PACEInputType.getCertificateDescription()`
	`PACEInputType.getOutputType()`

PRECONDITION	Objects from test case U_U3
POSTCONDITION	-
EXPECTED RESULT	The functions under test must yield the correct values.
ERROR CASES	-

TEST NO.	U_U5 – PACE Output Type initialization
DESCRIPTION	An instance of the class `PACEOutputType` is initialized with an `AuthDataMap` Variable.
TEST SUBJECT	`PACEOutputType(AuthDataMap)`

PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	An initialized object is available.
ERROR CASES	-

TEST NO.	U_U6 – PACE Output Type extraction
DESCRIPTION	The `PACEOutputType` class must extract the correct values from the initialized Object. Use the setter methods to insert values into the object.

TEST SUBJECT	`PACEOutputType.setCurrentCAR(byte[])`
	`PACEOutputType.setEFCardAcces(byte[])`
	`PACEOutputType.setPreviousCAR(byte[])`
	`PACEOutputType.setIDPICC(byte[])`
	`PACEOutputType.setRetryCounter(byte)`
	`PACEOutputType.getAuthDataType()`

PRECONDITION	Objects from test case U_U5
POSTCONDITION	-
EXPECTED RESULT	A `DIDAuthenticationDataType` with the correct values of the Objects from test case U – U5.
ERROR CASES	-

TEST NO.	U_U7 – Establish PACE Request initialization
DESCRIPTION	Instances of the class `EstablishPACERequest` are initialized with: the parameters `passwordType`, `chat`, `password`, `certDesc`. Three types have to be supported: generic PACE, germane ID and QES.
TEST SUBJECT	`EstablishPACERequest(byte, byte[], byte[], byte[])`
PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	Initialized objects are available.
ERROR CASES	-

TEST NO.	U_U8 – Establish PACE Request type check
DESCRIPTION	The created types are checked if they are compatible with the three supported types generic PACE, German ID and QES.
TEST SUBJECT	Establish `PACERequest.isSupportedType(List<PACECapabilities.PACECapability>)`
PRECONDITION	Objects from test case U_U7

POSTCONDITION	-
EXPECTED RESULT	True if a positive result is expected, false if the type should not be supported.
ERROR CASES	-

TEST NO.	U_U9 – Establish PACE Request extraction
DESCRIPTION	The `EstablishPACERequest` class must create valid structures according to the PC/SC-10 Amd. 1 specification [7].
TEST SUBJECT	`EstablishPACERequest.toBytes()`
PRECONDITION	Objects from test case U_U7
POSTCONDITION	-
EXPECTED RESULT	The function under test must yield the correct data structure.
ERROR CASES	-

TEST NO.	U_U10 – Establish PACE Response initialization
DESCRIPTION	Instance of the class `EstablishPACEResponse` are initialized with a structure according to PC/SC specifications (Amd. 1) [7].
TEST SUBJECT	`EstablishPACEResponse(byte[])`
PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	An initialized object is available.
ERROR CASES	-

TEST NO.	U_U11 – Establish PACE Response extraction
DESCRIPTION	The Establish PACE Response objects must extract the correct values from the given structures.

TEST SUBJECT	`EstablishPACEResponse.hasEFCardAccess()`
	`EstablishPACEResponse.hasCurrentCAR()`
	`EstablishPACEResponse.hastPreviousCAR()`
	`EstablishPACEResponse.hasIDICC()`
	`EstablishPACEResponse.getStatus()`
	`EstablishPACEResponse.getRetryCounter`
	`EstablishPACEResponse.getEFCardAccess()`
	`EstablishPACEResponse.getCurrentCAR()`
	`EstablishPACEResponse.getPreviousCAR()`
	`EstablishPACEResponse.getIDICC()`

PRECONDITION	Objects from test case U_U10
POSTCONDITION	-
EXPECTED RESULT	The functions under test must yield the correct values.

ERROR CASES	-

TEST NO.	U_U12 – Execute PACE Request initialization
DESCRIPTION	An instance of the class `EstablishPACEResponse` can be initialized in two ways. It can be initialized with a Function or a Function and a byte array.
TEST SUBJECT	`ExecutePaceRequest(Function)` `ExecutePaceRequest(Function, byte[])`
PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	Initialized objects are available.
ERROR CASES	`NullpointerException` in case the data field is null.

TEST NO.	U_U13 – Execute PACE Request extraction
DESCRIPTION	The `EstablishPACERequest` class must return the correct data.
TEST SUBJECT	`EstablishPACERequest.toBytes()`
PRECONDITION	Objects from test case U_U12
POSTCONDITION	-
EXPECTED RESULT	The returned value must encode the function, length and the data field if it exists.
ERROR CASES	-

TEST NO.	U_U14 – Execute PACE Response initialization
DESCRIPTION	Instance of the class `PACECapabilities` are initialized with a data structure according to PCSC-10 Amd. 1.
TEST SUBJECT	`ExecutePACEResponse(byte[])`
PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	Initialized objects are available.
ERROR CASES	`ArrayIndexOutOfBoundsException` in case the data structure is not well formed.

TEST NO.	U_U15 – Execute PACE Response extraction
DESCRIPTION	The Execute PACE Response class must extract the correct values from a given data structure.
TEST SUBJECT	`ExecutePACEResponse.getResultCode()` `ExecutePACEResponse.isError()`

	`ExecutePACEResponse.getResult()` `ExecutePACEResponse.getData()`
PRECONDITION	Objects from test case U_U14
POSTCONDITION	-
EXPECTED RESULT	The functions under test must yield the correct values. The Result object must be only tested for error and good status.
ERROR CASES	-

TEST NO.	U_U16 – PCSC Features response
DESCRIPTION	Converts a feature response from a reader to a Map containing the system specific control codes for a terminal.
TEST SUBJECT	`PCSCFeatures.featureMapFromRequest(byte[])`
PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	The Map must contain exactly those codes that are in the input. The result values are not checked as they are system dependent.
ERROR CASES	`NullpointerException` when the given input is null.

TEST NO.	U_U17 – PCSC Pin Modify initialization
DESCRIPTION	Instances of the class `PCSCPinModify` must be created for the password types , bcd, ascii-numeric and utf8.
TEST SUBJECT	`PCSCPinModify(PasswordAttributesType, byte[])`
PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	Initialized objects are available.
ERROR CASES	`IFDException` when the supplied values contradict each other.

TEST NO.	U_U18 – PCSC Pin Modify extraction
DESCRIPTION	The PCSC Pin Modify class must extract the correct values.

TEST SUBJECT	`PCSCPinModify.getMinPINSize()`
	`PCSCPinModify.getMaxPINSize()`
	`PCSCPinModify.toBytes()`

PRECONDITION	Objects from test case U_U17
POSTCONDITION	-
EXPECTED RESULT	The functions under test must yield the correct value.

ERROR CASES	-

TEST NO.	U_U19 – PCSC Pin Modify manipulation
DESCRIPTION	The `PCSCPinModify` class provides functions to modify values.

TEST SUBJECT	`PCSCPinModify.setMinPINSize(byte)`
	`PCSCPinModify.setMaxPINSize(byte)`
	`PCSCPinModify.setData()`
	`PCSCPinModify.toBytes()`

PRECONDITION	Objects from test case U_U17
POSTCONDITION	-
EXPECTED RESULT	When values are set, they must be reflected in the output of `toBytes()`.
ERROR CASES	-

TEST NO.	U_U20 – PCSC Pin Verify initialization
DESCRIPTION	Instances of the class `PCSCPinModify` must be created for the password types , bcd, ascii-numeric and utf8.
TEST SUBJECT	`PCSCPinVerify(PasswordAttributesType, byte[])`
PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	Initialized objects are available.
ERROR CASES	IFDException when the supplied values contradict each other.

TEST NO.	U_U21 – PCSC Pin Verify extraction
DESCRIPTION	The `PCSCPinVerify` class must extract the correct values.

TEST SUBJECT	`PCSCPinVerify.getMinPINSize()`
	`PCSCPinVerify.getMaxPINSize()`
	`PCSCPinVerify.toBytes()`

PRECONDITION	Objects from test case U_U20
POSTCONDITION	-
EXPECTED RESULT	The functions under test must yield the correct value.
ERROR CASES	-

TEST NO.	U_U22 – PCSC Pin Verify insertion
DESCRIPTION	The `PCSCPinVerfiy` class provides functions to modify values.
TEST SUBJECT	`PCSCPinVerfiy.setMinPINSize(byte)` `PCSCPinVerify.setMaxPINSize(byte)`

	`PCSCPinVerfiy.setData()` `PCSCPinVerify.toBytes()`
PRECONDITION	Objects from test case U_U20
POSTCONDITION	-
EXPECTED RESULT	When values are set, they must be reflected in the output of `toBytes()`.
ERROR CASES	-

TEST NO.	U_U23 – Byte Array Comparator
DESCRIPTION	The `ByteArrayComparator` compares two given byte arrays. The ordering is lexicographic. If both arrays are equal up to the same length, then the shorter array yields a negative sorting value
TEST SUBJECT	`ByteArrayComperator.compare(byte[], byte[])`
PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	Ordering as in the description.
ERROR CASES	`NullpointerException` when any value is null.

2.2.2 Terminal

The Terminal group contains classes related to card terminals. The classes which form a similar API to the Java smartcard IO are not part of this group as this would require the simulation of complex hardware which is out of scope of the unit test.

The table below contains all classes for which test cases exist in the Terminal group and are described in this section.

Class	Group
`org.openecard.ifd.scio.AbstractTerminal`	Terminal
`org.openecard.ifd.scio.EventListener`	Terminal

TEST NO.	U_T1 – Abstract Terminal verifyUser
DESCRIPTION	Create instances of the `AbstractTerminal` class simulating terminals with PIN pad and without PIN pad (i.e. IFD capabilities). The `verifyUser` method is used to verify the proper selection of the required commands in the IFD.
TEST SUBJECT	`AbstractTerminal(IFD, SCWrapper, UserConsent, byte[], BigInteger)`

	`AbstractTerminal.verifyUser(VerifyUser)`
PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	The mocked IFD is called appropriately depending on the reader capabilities.
ERROR CASES	-

TEST NO.	U_T2 - Event Listener status difference
DESCRIPTION	An instance of the class `EventListener` is initialized with mocked objects simulating the following cases: • No terminal present → Terminal present • Terminal Present → No terminal present • No card present → Card present • Card present → No card present • No terminal present → Card present • Card present → No terminal present Only the synchronous mode is tested.
TEST SUBJECT	`EventListener(IFD, SCWrapper, ExecutorService, byte[], long, ChannelHandleType, List, Boolean)` `EventListener.call()`
PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	An initialized object is available.
ERROR CASES	-

2.2.3 APDU

The classes in the APDU group form the basic building blocks for the creation of specific APDU classes according to ISO/IEC 7816-4 such as SELECT and READ.

The table below contains all classes for which test cases exist in the APDU group and are described in this section.

Class	Group
`CardCommandAPDU`	APDU
`CardResponseAPDU`	APDU

TEST NO.	U_A1 – Card Command APDU initialization
DESCRIPTION	Create instances of the class `CardCommandAPDU` and check if the serialization equals to the expected APDU.

	For the tests the actual values do not matter. That means INS, P1, P2 can be any value and the data does not have to be a sensible APDU The following APDUs should be tested: • No Lc, no Le • No Lc, short Le • No Lc, long Le • Short Lc, no Le • Long Lc, no Le • Short Lc, short Le • Short Lc, long Le • Long Lc, short Le • Long Lc, long Le • Invalid combination 1 byte Le with long Lc • Invalid combination 1 byte Lc with long Le • Invalid combination data does not match Lc

TEST SUBJECT	`CardCommandAPDU(byte[])`
	`CardCommandAPDU(byte, byte, byte, byte)`
	`CardCommandAPDU(byte, byte, byte, byte, byte)`
	`CardCommandAPDU(byte, byte, byte, byte, short)`
	`CardCommandAPDU(byte, byte, byte, byte, int)`
	`CardCommandAPDU(byte, byte, byte ,byte, byte [])`
	`CardCommandAPDU(byte, byte, byte ,byte, byte [], byte)`
	`CardCommandAPDU(byte, byte, byte ,byte, byte [], short)`
	`CardCommandAPDU(byte, byte, byte ,byte, byte [], int)`
	`CardCommandAPDU.toByteArray()`

PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	Initialized objects are available and the output of toByteArray matches the expected APDU value.
ERROR CASES	IllegalArgumentException when the given input does not satisfy the APDU syntax.

TEST NO.	U_A2 – Card Command APDU change
DESCRIPTION	Change the fields of a given instance and observe the output via the get functions.

TEST SUBJECT	`CardCommandAPDU.setINS(byte)`
	`CardCommandAPDU.setP1(byte)`
	`CardCommandAPDU.setP2(byte)`
	`CardCommandAPDU.setP1P2(byte[])`
	`CardCommandAPDU.setLC(byte)`
	`CardCommandAPDU.setLC(short)`
	`CardCommandAPDU.setLC(int)`
	`CardCommandAPDU.setData(byte[])`
	`CardCommandAPDU.setBody(byte[])`
	`CardCommandAPDU.setLE(byte)`
	`CardCommandAPDU.setLE(short)`
	`CardCommandAPDU.setLE(int)`

	`CardCommandAPDU.setChaining()`
	`CardCommandAPDU.setSecureMessaging()`
	`CardCommandAPDU.getCLA()`
	`CardCommandAPDU.getINS()`
	`CardCommandAPDU.getP1()`
	`CardCommandAPDU.getP2()`
	`CardCommandAPDU.getP1P2()`
	`CardCommandAPDU.getHeader()`
	`CardCommandAPDU.getHeader(byte[])`
	`CardCommandAPDU.getLC()`
	`CardCommandAPDU.getLE()`
	`CardCommandAPDU.isSecureMessaging()`

PRECONDITION	Objects from test case U_A1.
POSTCONDITION	-
EXPECTED RESULT	Each set function sets the expected values in the fields which is checked by evaluating the result of the respective get function. If a set function manipulates multiple fields, each get function belonging to the respective field is checked.
ERROR CASES	-

TEST NO.	U_A3 – Card Command APDU transmit message
DESCRIPTION	Create a `Transmit` object based on any APDU. Test with no, one and two expected responses.
TEST SUBJECT	`CardCommandAPDU.makeTransmit(byte[], list)`
PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	The transmit message must contain the values of the `CardCommandAPDU` and the given response codes.
ERROR CASES	`NullpointerException` when the response list is null or contains null values.

TEST NO.	U_A4 – Card Response APDU initialization
DESCRIPTION	Create instances of the class `CardResponseAPDU` and check if the given values are returned by the get functions.

TEST SUBJECT	`CardResponseAPDU(byte[])`
	`CardResponseAPDU(byte[], byte[])`
	`CardResponseAPDU.getData(byte[])`
	`CardResponseAPDU.getTrailer()`
	`CardResponseAPDU.getSW1()`
	`CardResponseAPDU.getSW2()`
	`CardResponseAPDU.getSW()`
	`CardResponseAPDU.getStatusBytes()`
	`CardResponseAPDU.getStatusMessage()`
	`CardResponseAPDU.isNormalProcessed()`

	`CardResponseAPDU.isWarningProcessed()`
	`CardResponseAPDU.isExecutionError()`
	`CardResponseAPDU.isCheckingError()`

PRECONDITION	-
POSTCONDITION	-
EXPECTED RESULT	The response APDUs are reflected accordingly in the get functions.
ERROR CASES	`NullpointerException` when a given trailer is null.

TEST NO.	U_A5 – Card Response APDU output
DESCRIPTION	Check the serialization functions of the `CardResponseAPDU` class.
TEST SUBJECT	`CardResponseAPDU.toByteArray()` `CardResponseAPDU.toHexString()`
PRECONDITION	Objects from test case U_A4.
POSTCONDITION	-
EXPECTED RESULT	The response APDU represented by the instance.
ERROR CASES	-

TEST NO.	U_A6 – Card Response APDU manipulation
DESCRIPTION	Check if the set functions yield the appropriate values in the get function.

TEST SUBJECT	`CardResponseAPDU.setTrailer(byte[])`
	`CardResponseAPDU.setSW1(byte)`
	`CardResponseAPDU.setSW2(byte)`
	`CardResponseAPDU.setStatusBytes(byte[])`
	`CardResponseAPDU.getTrailer()`
	`CardResponseAPDU.getTrailer(byte[])`
	`CardResponseAPDU.getSW1()`
	`CardResponseAPDU.getSW2()`
	`CardResponseAPDU.getSW()`
	`CardResponseAPDU.getStatusBytes()`
	`CardResponseAPDU.getStatusMessage()`
	`CardResponseAPDU.isNormalProcessed()`
	`CardResponseAPDU.isWarningProcessed()`
	`CardResponseAPDU.isExecutionError()`
	`CardResponseAPDU.isCheckingError()`

PRECONDITION	Objects from test case U_A4
POSTCONDITION	-
EXPECTED RESULT	The response APDUs are reflected accordingly in the get functions.
ERROR CASES	-

3. System Tests

System tests verify the successful integration of the IFD into theFutureID client application. The system tests of the IFD will be developed within WP 3.7 (Client Testbed). Nevertheless this section provides a description of two tests which shall be conducted in WP 3.7. The first system test relies on the German Identity Card to test multiple functions of the IFD. The second test is based on the Austrian Citizen Card. Please note that not all functions of the IFD are covered by both tests and that some functionality of the IFD is indirectly tested. For instance, the functions `ListIFDs` and `GetIFDCapabilities` are used by the Event Manager and not directly by the system tests, but they are part of the whole test run.

3.1 German Identity Card

The German Identity Card (nPA) provides electronic authentication and signature capabilities. This test performs an authentication using the nPA to test the integration of the IFD. In detail, we simulate the process of browsing a website and performing an authentication in order to access a certain service. This includes the activation of the client as well as all steps of the user interaction for authentication.

3.1.1 Setting

The test environment to run the system tests requires a contactless card reader and an nPA. Furthermore, the test needs an eID server to perform the authentication protocols of the German ID card, as specified in BSI TR-03110[1].

3.1.2 Coverage

The following list includes all IFD functions and describes which ones are covered by the system test using the nPA. As the list shows, the test nearly covers all functions of the IFD. The missing functions should be verified by other system or unit tests.

FUNCTION	DESCRIPTION	COVERED
`EstablishContext`	Opens a session with the IFD service and returns a `ContextHandle` to address the IFD instance in future.	■[1]
`ReleaseContext`	Terminates a session with the IFD service.	■[1]
`ListIFDs`	Returns a list of available card terminals.	■[2]
`GetIFDCapabilities`	Returns information of a card terminal.	■[2]
`GetStatus`	Determines the current status of the card terminal.	■[2]
`Wait`	Registers an event callback (e.g., new smart card is inserted).	■[2]
`Cancel`	Cancels a `Wait` call.	■[2]
`ControlIFD`	Sends (proprietary) commands to a connected card terminal.	■[3]
`Connect`	Establishes a connection to a card and returns a `SlotHandle`to address the connection in future.	■
`Disconnect`	Destroys a connection to a smart card.	■
`BeginTransaction`	Starts a transaction through which several commands can be sent to the card.	□
`EndTransaction`	Ends a transaction.	□
`Transmit`	Sends a set of APDUs.	■
`VerifyUser`	Verifies the user by means of a PIN.	□
`ModifyVerificationData`	Modifies the verification data (i.e. changes a PIN).	□
`Output`	Can be used to control the output units of a card terminal.	□
`SignalEvent`	Can be used to inform applications about card terminal events.	■[2]

[1] Used during the IFD initialization
[2] Used by the Event Manager (card detection)
[3] Used in case of card terminal with PIN pad

3.2 Austrian Citizen Card

The Austrian Citizen Card allows creating signatures which can either be based on qualified or non-qualified certificates, and, thus, two different keys. This test shall simulate an OASIS DSS `SignRequest` to the eSign services in order to test the IFD commands listed below using both of these keys. For further details on OASIS DSS we refer the reader to [9], whereas a detailed description of the eSign services can be found in the deliverable document for WP33.3 [1].

3.2.1 Setting

The test environment consists of a card reader and the FutureID client with activated eSign add-on (cf. [9]). In order to send OASIS DSS requests, the `dssreq` tool (a tool provided for the demos in WP33.3 [1]) can be used.

3.2.2 Coverage

FUNCTION	DESCRIPTION	COVERED
`EstablishContext`	Opens a session with the IFD service and returns a `ContextHandle` to address the IFD instance in future.	■
`ReleaseContext`	Terminates a session with the IFD service.	■
`ListIFDs`	Returns a list of available card terminals.	■
`GetIFDCapabilities`	Returns information of a card terminal.	■
`GetStatus`	Determines the current status of the card terminal.	■
`Wait`	Registers an event callback (e.g., new smart card is inserted).	■
`Cancel`	Cancels a `Wait` call.	■
`ControlIFD`	Sends (proprietary) commands to a connected card terminal.	■
`Connect`	Establishes a connection to a card and returns a `SlotHandle`to address the connection in future.	■
`Disconnect`	Destroys a connection to a smart card.	■
`BeginTransaction`	Starts a transaction through which several commands can be sent to the card.	□
`EndTransaction`	Ends a transaction.	□
`Transmit`	Sends a set of APDUs.	■
`VerifyUser`	Verifies the user by means of a PIN.	■
`ModifyVerificationData`	Modifies the verification data (i.e. changes a PIN).	□
`Output`	Can be used to control the output units of a card terminal.	□
`SignalEvent`	Can be used to inform applications about card terminal events.	■

4. Requirements Verification

Coming back to the initial requirements of the IFD layer which have been specified in D31.1, this section focusses on the verification of the IFD requirements. Based on the unit and systems test as described in the previous sections it is verified whether the specified requirements are fulfilled.

For reading convenience, the following tables list all requirements as specified in D31.1, provide references to tests that were made to verify these requirements and comments the results or any open topics that could not be covered by the integration tests.

NO.	R1 – Smart Cards and Card Terminals (mandatory)
DESCRIPTION	The IFD service MUST provide a generalized interface for communication with arbitrary card terminals and smart cards.
TEST	Covered by all unit and system tests
COMMENT	This requirement covers the basic functionality of the IFD which is tested by all tests described above. Since the IFD specifications are based on ISO/IEC 24727-4 and TR-03112-6 API (cf. R10 – ISO/IEC 24727-4) the interface can be regarded as generalized.

NO.	R1_1 – Smart Cards and Card Terminals (optional)
DESCRIPTION	The IFD SHOULD contain functions for card terminals like establish and destroy a session, status and capability information and control commands.
TEST	System tests
COMMENT	This requirement is verified by the system tests in both use cases (see especially the first 8 commands in sections 3.1.2 and 3.2.2).

NO.	R1_2 – Smart Cards and Card Terminals (optional)
DESCRIPTION	The IFD SHOULD contain commands to connect, disconnect and transmit data to a smart card.
TEST	System tests
COMMENT	This requirement is verified by the system tests in both use cases, especially by the commands `Connect`, `Disconnect`and `Transmit`.

NO.	R2 – Secure Elements (optional)
DESCRIPTION	The IFD service SHOULD provide a generalized interface for communication with arbitrary secure elements.
TEST	System tests

COMMENT	This requirement is addressed by testing the `Transmit` command with both system test cases. This command can be used for sending APDUs to arbitrary secure elements. Within the APDU group of the unit tests (Section 2.2.3) this functionality is also tested on unit level. In addition, the capability of the IFD to integrate the OpenMobile API allows addressing arbitrary secure elements on mobile devices as well. The commands`BeginTransaction` and `EndTransaction`are not covered by the system test cases.However, these commands are neither provided by the OpenMobile API nor by the Android NFC interface. Therefore, they are not available on mobile platforms anyway.

NO.	R3 – Protocols (mandatory)
DESCRIPTION	The IFD service MUST support protocols to establish trusted channels.
TEST	System test (German Identity Card)
COMMENT	The IFD provides the Protocol API to provide means for protocols. The system test with the German ID card includes the PACE protocol which uses the Protocol API of the IFD service.

NO.	R3_1 – Protocols (optional)
DESCRIPTION	The IFD SHOULD provide commands to establish and destroy trusted channels. Input data, which is required to establish the channel, SHOULD be defined in a flexible and transparent way.
TEST	System test (German Identity Card)
COMMENT	The system test with the German ID card includes the PACE protocol which uses the Protocol API of the IFD service.

NO.	R3_2 – Protocols (optional)
DESCRIPTION	The IFD SHOULD consider protocol specific requirements like the functionality of connected card terminals, for instance, whether a keypad or display is present or not.
TEST	System test (German Identity Card)
COMMENT	This requirement is verified by the system tests with the German ID card. Especially the commands `GetIFDCapabilities`, `GetIFDStatus` and `ControlIFD` support this requirement.

NO.	R4 – NFC (mandatory)
DESCRIPTION	The IFD service MUST support Near Field Communication (NFC).
TEST	System test (German Identity Card)
COMMENT	When the system test with the German ID card is done on a mobile device with NFC interface, then this requirement will be verified by this test.

NO.	R5 – Open Mobile API (optional)
DESCRIPTION	The IFD service SHOULD support the Open Mobile API.
TEST	System test (German Identity Card)
COMMENT	When the system test with the German ID card is done on a mobile device with the Open Mobile API installed, then this requirement will be verified by this test. The test can also be combined with the previous requirement by using a mobile device with Open Mobile API including an NFC plugin-terminal. The card is the addressed via the NFC terminal of the Open Mobile API.

NO.	R6 – PC/SC (mandatory)
DESCRIPTION	The IFD service MUST support PC/SC.
TEST	Unit and system tests
COMMENT	PC/SC is used in any tests including smartcards and performed on desktop computers. Thus, this requirement is covered by all system and unit tests described above.

NO.	R7 – SICCT (optional)
DESCRIPTION	The IFD service SHOULD support SICCT.
TEST	Unit and system tests
COMMENT	This requirement can be verified by performing all system and unit tests on a desktop computer using the SICCT interface [8].

NO.	R8 – TPM (optional)
DESCRIPTION	The IFD service SHOULD support Trusted Platform Modules (TPM).
TEST	Not covered
COMMENT	As it has been decided not to address TPMs in the current version of the client, this optional requirement is not covered by the above tests.

NO.	R9 – Proxy (optional)
DESCRIPTION	Mobile and computer-based platforms MAY support multiple interfaces. Consider a notebook that is equipped with an NFC and a USB interface. Thus, smart cards can be accessed via NFC or via a USB-connected card reader. Therefore, the IFD service SHOULD support multiple interfaces as defined in R4 – R8 simultaneously.
TEST	System tests
COMMENT	The IFD specification contains a proxy layer that supports this requirement. It can be verified by performing the system tests on a device with multiple interfaces.

NO.	R10 – ISO/IEC 24727-4 (mandatory)
DESCRIPTION	The IFD service MUST support the ISO/IEC 24727-4 specification.
TEST	Unit and system tests
COMMENT	When the unit and system tests are passed, then this requirement is fulfilled by definition since the IFD specification is based on ISO/IEC 24727-4.

NO.	R11 – Bindings (mandatory)
DESCRIPTION	The IFD service MUST provide a common interface for external application.
TEST	System and unit tests
COMMENT	By definition, the IFD provides a standardized interface which can be called by external applications. All system and unit tests use this interface and therefore verify its functionality.

NO.	R11_1 – Bindings (optional)
DESCRIPTION	The IFD service SHOULD support a language-specific Programming Interface, for instance for Java and C.
TEST	System and unit tests
COMMENT	The IFD has been implemented in Java and therefore all unit and system tests verify the functionality of the Java interface. An implementation in C is not planned during this project and therefore will not be verified by the tests.

NO.	R11_2 – Bindings (optional)
DESCRIPTION	The IFD service SHOULD support a SOAP binding.
TEST	Not covered
COMMENT	A SOAP binding is not needed for the current demo cases and therefore not covered. The general functionality of the bindings concept is demonstrated by the PAOS binding (see R11_3). Thus a future implementation of a SOAP binding will be straightforward.

NO.	R11_3 – Bindings (optional)
DESCRIPTION	The IFD service SHOULD support a PAOS binding.
TEST	System tests (German ID card)
COMMENT	The PAOS protocol is used by the German ID card and therefore covered by this system test case. Accordingly, the functionality of the PAOS binding is verified by this test.

In summary, all mandatory requirements for the IFD layer are covered by the specified unit and system test cases. Concerning the optional requirements, those requirements that are needed for the demo implementation are covered by the test cases as well. The support of a TPM is not tested since it has been decided not to pursue TPM support any further within this project. Also, the actual implementation of the IFD is concentrated on Java and thus a C interface is not tested.

Overall, the unit and system tests provide a broad verification of the IFD requirements and demonstrate the functionality of the fundamental design principles of the IFD.

5. Conclusion

The unit and system tests described in Sections 2 and 3 provide a broad basis for the actual test implementation of WP 37 and allow verifying the IFD functionality with a high trust level. While the unit tests provide verification of the individual modules independent of any terminal hardware or smart cards, the system tests demonstrate the hardware interaction for representative test cases (nPA, Austrian Citizen Card).

With the system tests and unit tests combined, all mandatory requirements of the IFD specification can be verified. All optional requirements that are relevant for the demo implementation within FutureID are verified as well. The only optional requirements which are not covered by the tests are not relevant for the implementation within FutureID (TPM support, SOAP binding).

6. Bibliography

[1] FutureID, *WP33 - eSign Services, D33.3 - Implementation of the Framework,* 2014.

[2] Bundesamt für Sicherheit in der Informationstechnik (BSI), *Advanced Security Mechanisms for Machine Readable Travel Documents,* Technical Guideline TR-03110, Version 2.10, Part 1 - 3, https://www.bsi.bund.de/EN/Publications/TechnicalGuidelines/TR03110/BSITR03110.html, 2012.

[3] Simalliance, Open Mobile API Specifications, V 2.04, http://www.simalliance.org/en?t=/documentManager/sfdoc.file.supply&fileID=1383153790991

[4] TesNG Framework, http://testng.org/doc/index.html

[5] http://maven.apache.org/

[6] JMockit, http://jmockit.github.io/

[7] PC/SC specifications, http://www.pcscworkgroup.com/specifications/overview.php

[8] SICCT Interface, https://www.teletrust.de/projekte/sicct/

[9] OASIS Digital Signature Services TC, Digital Signature Services - DSS Core Protocols, Elements, and Bindings v1.0, 2007