
**Space data and information transfer
systems — Space link extension —
Application program interface for
transfer services — Core specification**

*Systèmes de transfert des informations et données spatiales —
Extension de liaisons spatiales — Interface du programme
d'application pour les services de transfert — Spécification de base*





COPYRIGHT PROTECTED DOCUMENT

© ISO 2021

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of ISO documents should be noted (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html.

This document was prepared by the Consultative Committee for Space Data Systems (CCSDS) (as CCSDS 914.0-M-2 Cor.1, August 2016) and was adopted (without modifications) by Technical Committee ISO/TC 20, *Space vehicles*, Subcommittee SC 13, *Space data and information transfer systems*.

This third edition cancels and replaces the second edition (ISO 18441:2016), which has been technically revised.

The main changes compared to the previous edition are as follows:

- references CCSDS 913.1-B for one-way hash function algorithm and removes reference to Secure Hash Algorithm standard.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

CONTENTS

<u>Section</u>	<u>Page</u>
1 INTRODUCTION	1-1
1.1 PURPOSE OF THIS RECOMMENDED PRACTICE	1-1
1.2 SCOPE	1-1
1.3 APPLICABILITY	1-2
1.4 RATIONALE	1-3
1.5 DOCUMENT STRUCTURE	1-4
1.6 DEFINITIONS	1-7
1.7 REFERENCES	1-10
2 DESCRIPTION OF THE SLE API	2-1
2.1 INTRODUCTION	2-1
2.2 SPECIFICATION METHOD AND NOTATION	2-2
2.3 LOGICAL VIEW	2-7
2.4 SECURITY ASPECTS OF CORE SLE API CAPABILITIES	2-58
3 SPECIFICATION OF API COMPONENTS	3-1
3.1 INTRODUCTION	3-1
3.2 API PROXY	3-1
3.3 API SERVICE ELEMENT	3-27
3.4 SLE OPERATIONS	3-52
3.5 SLE UTILITIES	3-56
3.6 SLE APPLICATION	3-64
3.7 HANDLING OF IN PROCESS THREADS AND EXTERNAL EVENTS	3-71
4 STATE TABLES	4-1
4.1 INTRODUCTION	4-1
4.2 NOTATION	4-1
4.3 GENERAL ERROR HANDLING CONVENTIONS	4-2
4.4 STATE TABLE FOR ASSOCIATIONS	4-2
4.5 STATE TABLES FOR SERVICE INSTANCES	4-15
ANNEX A SPECIFICATION OF COMMON INTERFACES (NORMATIVE)	A-1
ANNEX B RESULT CODES (NORMATIVE)	B-1
ANNEX C STRUCTURE OF THE SERVICE INSTANCE IDENTIFIER FOR VERSION 1 OF THE SLE SERVICES RAF, RCF, AND CLTU (NORMATIVE)	C-1
ANNEX D SIMPLE COMPONENT MODEL (NORMATIVE)	D-1

API FOR SLE TRANSFER SERVICES—CORE SPECIFICATION

CONTENTS (continued)

<u>Section</u>	<u>Page</u>
ANNEX E CONFORMANCE (NORMATIVE)	E-1
ANNEX F INTERACTION OF COMPONENTS (INFORMATIVE)	F-1
ANNEX G INTERFACE CROSS REFERENCE (INFORMATIVE)	G-1
ANNEX H INDEX TO DEFINITIONS (INFORMATIVE)	H-1
ANNEX I ACRONYMS AND ABBREVIATIONS (INFORMATIVE)	I-1
ANNEX J INFORMATIVE REFERENCES (INFORMATIVE)	J-1

Figure

1-1 SLE Services and SLE API Documentation.....	1-6
2-1 UML Stereotypes Used in This Recommended Practice.....	2-3
2-2 Top Level Decomposition of the API.....	2-7
2-3 Structure of the Package API Proxy	2-9
2-4 Reporting and Tracing by the Proxy.....	2-10
2-5 Configuration Database of the Proxy.....	2-20
2-6 Structure of the Package API Service Element	2-23
2-7 Reporting and Tracing by the Service Element.....	2-24
2-8 Sequential Control Interface Component Class Controlled Component.....	2-39
2-9 Concurrent Control Interface.....	2-43
2-10 Structure of the Package SLE Application	2-44
2-11 Reporting and Tracing Interfaces Provided by the Application	2-45
2-12 Operation Objects	2-49
2-13 Operation Object Interfaces for Common Association Management.....	2-53
2-14 Common SLE Operation Objects	2-54
2-15 SLE Utilities	2-56
4-1 Processing Context for the Association State Table.....	4-3
4-2 Processing Context for the Service Instance State Table	4-16
B-1 Structure of Result Codes	B-1
F-1 Configuration of Components	F-3
F-2 Configuration of Interfaces for Service Provisioning.....	F-3
F-3 Interaction of API Components	F-4
F-4 Initialization and Shutdown.....	F-5
F-5 Collaboration Diagram for Use of Operation Objects.....	F-8
F-6 Sequence Diagram for Use of Operation Objects.....	F-9
F-7 User Side Binding (User Initiated Bind).....	F-12
F-8 User Side Unbinding (User Initiated Bind)	F-13
F-9 Provider Side Binding (User Initiated Bind)	F-14
F-10 Provider Side Unbinding (User Initiated Bind).....	F-16

CONTENTS (continued)

<u>Table</u>	<u>Page</u>
C-1 Identifiers and Abbreviations for Attributes.....	C-3
E-1 Optional Features for the API Proxy	E-3
E-2 Optional Features for the API Service Element.....	E-6
E-3 Parameters That May Be Constrained by a Proxy.....	E-9
E-4 Parameters That May Be Constrained by a Service Element	E-10

1 INTRODUCTION

1.1 PURPOSE OF THIS RECOMMENDED PRACTICE

The purpose of this Recommended Practice is to define a C++ Application Program Interface (API) for CCSDS Space Link Extension (SLE) Transfer Services, which is independent of any specific technology used for communications between an SLE service user and an SLE service provider.

This API is intended for use by application programs implementing SLE services. It can be configured to support SLE service user applications or SLE service provider applications.

This API is also intended to simplify the implementation of gateways that are required to achieve interoperability between SLE service provider and SLE service user applications using different communications technologies.

Using this Application Program Interface Recommended Practice, API implementations (software packages) able to run on specific platforms can be developed. Once developed, such a package can be supplied to new users of SLE services for integration with their user or production facilities, thus minimizing their investment to buy into SLE support.

1.2 SCOPE

1.2.1 ITEMS COVERED BY THIS RECOMMENDED PRACTICE

This Recommended Practice defines the Application Program Interface in terms of:

- a) the components that provide the services of the API;
- b) the functionality provided by each of the components;
- c) the interfaces provided by each of the components; and
- d) the externally visible behavior associated with the interfaces exported by the components.

It does not specify:

- a) individual implementations or products;
- b) the internal design of the components; and
- c) the technology used for communications.

This Recommended Practice defines those aspects of the Application Program Interface, which are common for all SLE service types or for a subset of the SLE service types, e.g., all return link services or all forward link services. It also defines a framework for specification of service type-specific elements of the API. Service-specific aspects of the API are defined by supplemental Recommended Practice documents for SLE return link services (references [10], [11], and [12]) and SLE forward link services (references [13] and [14]).

This Recommended Practice for the Application Program Interface responds to the requirements imposed on such an API by the CCSDS SLE transfer service Recommended Standards that were available when this Recommended Practice was released.

1.2.2 CONFORMANCE TO CCSDS RECOMMENDED STANDARDS

This version of the SLE API Recommended Practice conforms to the CCSDS Recommended Standards for Space Link Extension Services, referenced in 1.7, with the exception of the following optional features:

- a) The negotiation procedure for version numbers in the BIND operation is not supported. If the responder does not support the version number identified in the BIND Invocation, it responds with a BIND Return containing a negative result and the diagnostic ‘version number not supported’. The responder does not propose an alternative version number.
- b) Provider-initiated binding, specified by CCSDS Recommended Standards for return link services is not included in this Recommended Practice. The management parameters that specify the bind initiative are supported to simplify addition of this procedure in later versions.

1.3 APPLICABILITY

The Application Program Interface specified in this document supports three generations of SLE Transfer Service specifications, namely:

- a) Generation 1 covering the services RAF, RCF, and FCLTU identified by the version number 1 in the BIND operation, as specified by references [C1], [C2], and [C3];
- b) Generation 2 covering
 - 1) the services RAF, RCF, and FCLTU identified by the version number 2 in the BIND operation, as specified by references [J9], [J10], and [J12];
 - 2) the services ROCF and FSP identified by the version number 1 in the BIND operation, as specified by references [J11] and [J13];
- c) Generation 3 covering the services RAF, RCF, ROCF, FCLTU, and FSP identified by the version number 4 in the BIND operation, as specified by references [4], [5], [6], [7], and [8].

API FOR SLE TRANSFER SERVICES—CORE SPECIFICATION

Support for Generation 1 and Generation 2 of these services is included for backward compatibility purposes for a limited time and may not be continued in future versions of this specification. Support for Generation 1 (i.e., version 1 of the RAF, RCF and CLTU services) implies that SLE API implementations of this specification are able to interoperate with peer SLE systems that comply with the specification of the Transport Mapping Layer (TML) in ‘Specification of a SLE API Proxy for TCP/IP and ASN.1’, ESOC, SLES-SW-API-0002-TOS-GCI, Issue 1.1, February 2001. For Generation 2 and 3 of these services, SLE API implementations of this specification are able to interoperate with peer SLE systems that comply with the specification of the Transport Mapping Layer (TML) in reference [9].

Provisions within this Recommended Practice that are specific for one or more generations are marked as follows:

- [Gn:] for provisions specific to Generation n;
- [Gn,m:] for provisions specific to Generation n and Generation m.

Provisions that apply to all generations are not marked.

1.4 RATIONALE

This Recommended Practice describes the services provided by a software package implementing the API to application software using the API. It specifies the mapping of the SLE Transfer Services specifications to specific functions and parameters of the SLE API. It also specifies the distribution of responsibility for specific functions between SLE API software and application software. The distribution of responsibility has been defined with due consideration for reusability of software packages implementing the SLE API.

The goal of this Recommended Practice is to create a guide for interoperability between

- a) software packages implementing the SLE API; and
- b) application software using the SLE API.

This interoperability guide also allows exchangeability of different products implementing the SLE API, as long as they adhere to the interface specification of this Recommended Practice.

1.5 DOCUMENT STRUCTURE

1.5.1 OVERVIEW

This Recommended Practice is organized in two parts and a set of annexes.

1.5.1.1 Part I—The Descriptive Part

The descriptive part presents the API Model in section 2 using the Unified Modeling Language (UML) (see reference [J6]).

1.5.1.2 Part II—The Prescriptive Part

The prescriptive part contains the specification of the API. In case of any discrepancies between the descriptive part and the prescriptive part, the specifications in the latter shall apply.

Section 3 contains detailed specifications of the API components and of the interfaces that must be provided by the application.

Section 4 defines the state tables that must be implemented by the API.

1.5.1.3 Annexes

Annex A contains the detailed declaration of the C++ interfaces, which are common for all SLE service types.

Annex B lists the result codes that are used by the API.

[G1:] For version 1 of the services RAF, RCF, and CLTU, annex C defines a standard ASCII representation for the service instance identifier and lists the attribute identifiers and abbreviations that are valid for the service instance identifier.

[G2,3:] For later versions of these services and all other services, these specifications are provided by the applicable CCSDS Recommended Standards.

Annex D describes the design patterns and conventions that shall be applied to API components. The specifications in this annex are also relevant for the application software using the API.

Annex E defines requirements for software products claiming conformance with this Recommended Practice.

Annex F describes the interaction of API components, showing several use cases.

API FOR SLE TRANSFER SERVICES—CORE SPECIFICATION

Annex G provides cross-references between interfaces provided by API components and interfaces used by API components.

Annex H contains an index to definitions.

Annex I explains all acronyms used in this Recommended Practice.

Annex J lists informative reference documents.

1.5.2 DOCUMENTATION TREE FOR SLE SERVICES AND SLE API

This Recommended Practice is based on the cross support model defined in the SLE Reference Model (reference [3]). The SLE services constitute one of the three types of Cross Support Services:

- a) Part 1: SLE Services;
- b) Part 2: Ground Domain Services; and
- c) Part 3: Ground Communications Services.

The SLE services are further divided into SLE Service Management and SLE Transfer Services.

NOTE – In reference [3], SLE transfer services are identified; however, the complete service specifications will be provided in separate Recommended Standards.

This Recommended Practice describes how the functions of an SLE transfer service provider or user can be implemented in a software package for the purpose of providing or using one or several SLE transfer services. It is part of a suite of documents specifying the API for SLE transfer services:

- a) Core Specification of the Application Program Interface for Transfer Services (this Recommended Practice);
- b) a set of Application Program Interfaces for specific Transfer Services; and
- c) Internet Protocol for Transfer Services.

The basic organization of the SLE services and SLE API documentation is shown in figure 1-1. The various documents are described in the following paragraphs.

API FOR SLE TRANSFER SERVICES—CORE SPECIFICATION

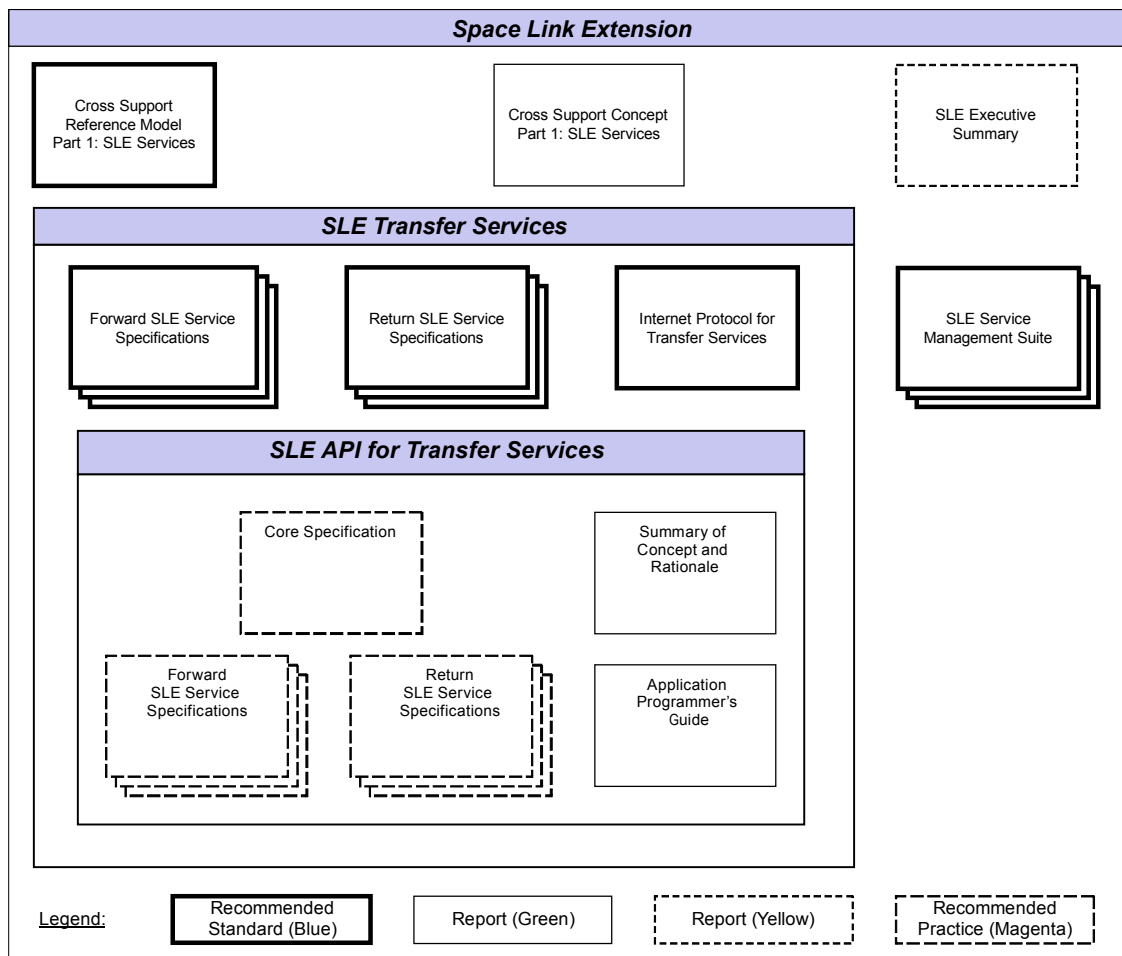


Figure 1-1: SLE Services and SLE API Documentation

- Cross Support Reference Model—Part 1: Space Link Extension Services*, a Recommended Standard that defines the framework and terminology for the specification of SLE services.
- Cross Support Concept—Part 1: Space Link Extension Services*, a Report introducing the concepts of cross support and the SLE services.
- Space Link Extension Services—Executive Summary*, an Administrative Report providing an overview of Space Link Extension (SLE) Services. It is designed to assist readers with their review of existing and future SLE documentation.
- Forward SLE Service Specifications*, a set of Recommended Standards that provide specifications of all forward link SLE services.
- Return SLE Service Specifications*, a set of Recommended Standards that provide specifications of all return link SLE services.

API FOR SLE TRANSFER SERVICES—CORE SPECIFICATION

- f) *Internet Protocol for Transfer Services*, a Recommended Standard providing the specification of the wire protocol used for SLE transfer services.
- g) *SLE Service Management Specifications*, a set of Recommended Standards that establish the basis of SLE service management.
- h) *Application Program Interface for Transfer Services—Core Specification*, this document.
- i) *Application Program Interface for Transfer Services—Summary of Concept and Rationale*, a Report describing the concept and rationale for specification and implementation of a Application Program Interface for SLE Transfer Services.
- j) *Application Program Interface for Return Services*, a set of Recommended Practice documents specifying the service-type specific extensions of the API for return link SLE services.
- k) *Application Program Interface for Forward Services*, a set of Recommended Practice documents specifying the service-type specific extensions of the API for forward link SLE services.
- l) *Application Program Interface for Transfer Services—Application Programmer's Guide*, a Report containing guidance material and software source code examples for software developers using the API.

1.6 DEFINITIONS

1.6.1 DEFINITION OF TERMS USED IN THIS DOCUMENT

1.6.1.1 Definitions from the SLE Reference Model

This Recommended Practice makes use of the following terms defined in reference [3]:

- a) invoker;
- b) offline delivery mode;
- c) online delivery mode;
- d) operation;
- e) performer;
- f) service provider (provider);
- g) service user (user);
- h) SLE protocol data unit (SLE-PDU);
- i) SLE transfer service instance (service instance);
- j) SLE transfer service production (service production);

- k) SLE transfer service provision (service provision);
- l) SLE transfer service provision period (provision period).

1.6.1.2 Definitions from the ISO Abstract Service Definitions and Conventions

This Recommended Practice makes use of the following terms defined in reference [19]:

- a) initiator;
- b) responder.

1.6.1.3 Definitions from SLE Transfer Service Specifications

This Recommended Practice makes use of the following terms defined in references [4], [5], [6], [7], and [8]:

- a) association;
- b) communications service;
- c) confirmed operation;
- d) invocation;
- e) parameter (of an operation);
- f) port identifier;
- g) return;
- h) unconfirmed operation.

1.6.1.4 Additional Definitions

1.6.1.4.1 General

For the purpose of this Recommended Practice, the following definitions also apply:

1.6.1.4.2 Component

A software module, providing a well-defined service via a set of interfaces. In this document the term component is used only to refer to the API components defined by this Recommended Practice.

API FOR SLE TRANSFER SERVICES—CORE SPECIFICATION

1.6.1.4.3 Component Interface

An interface exported by a component.

1.6.1.4.4 Component Object

An object within a component that can be accessed by one or more interfaces exported by the component. Objects providing more than one interface support navigation between these interfaces.

1.6.1.4.5 Client

A software entity that uses the services of a component or of an object by invocation of the methods of an interface provided by the component or object. In this Recommended Practice the qualified term ‘local client’ is used to stress the difference between an interface to a software entity on the same computer and the interface between an SLE service user and an SLE service provider.

1.6.1.4.6 Interface

The abstraction of a service that only defines the operations supported by that service, but not their implementations. The specification of an operation is referred to as a method.

1.6.2 NOMENCLATURE**1.6.2.1 Normative Text**

The following conventions apply for the normative specifications in this Recommended Standard:

- a) the words ‘shall’ and ‘must’ imply a binding and verifiable specification;
- b) the word ‘should’ implies an optional, but desirable, specification;
- c) the word ‘may’ implies an optional specification;
- d) the words ‘is’, ‘are’, and ‘will’ imply statements of fact.

NOTE – These conventions do not imply constraints on diction in text that is clearly informative in nature.

1.6.2.2 Informative Text

In the normative sections of this document, informative text is set off from the normative specifications either in notes or under one of the following subsection headings:

- Overview;
- Background;
- Rationale;
- Discussion.

1.6.2.3 Use of the Term ‘Client’

In a complete SLE API, the API Proxy interacts with the API Service Element in the same process and with a peer proxy across the network. However, the proxy might also be used in an environment where some other software entity interfaces locally to the proxy and supplies the interfaces defined for the API Service Element. An example for such a configuration is a gateway. Therefore this specification uses the term ‘client’ when referring to the entity with which the proxy interacts locally.

Where it is necessary to explicitly distinguish between interaction with the peer proxy across the network and interactions with the client on the local computer, the qualified term ‘local client’ is used.

1.6.2.4 Use of the Term ‘Release’

The term ‘release an object’ (or a resource) must be understood to mean that all actions shall be taken that are required to free the allocated memory or other operating system resources. For interfaces defined in this specification, this means that the method `Release()` must be called for every reference to an interface that a component holds. (See annex D for a description of the method `Release()` and of the reference counting scheme.)

1.7 REFERENCES

The following documents contain provisions which, through reference in this text, constitute provisions of this Recommended Practice. At the time of publication, the editions indicated were valid. All documents are subject to revision, and users of this Recommended Practice are encouraged to investigate the possibility of applying the most recent editions of the documents indicated below. The CCSDS Secretariat maintains a register of currently valid CCSDS documents.

NOTE – A list of informative references is provided in annex J.

API FOR SLE TRANSFER SERVICES—CORE SPECIFICATION

- [1] *Time Code Formats*. Issue 4. Recommendation for Space Data System Standards (Blue Book), CCSDS 301.0-B-4. Washington, D.C.: CCSDS, November 2010.
- [2] *Cross Support Concept—Part 1: Space Link Extension Services*. Issue 3. Report Concerning Space Data System Standards (Green Book), CCSDS 910.3-G-3. Washington, D.C.: CCSDS, March 2006.
- [3] *Cross Support Reference Model—Part 1: Space Link Extension Services*. Issue 2. Recommendation for Space Data System Standards (Blue Book), CCSDS 910.4-B-2. Washington, D.C.: CCSDS, October 2005.
- [4] *Space Link Extension—Return All Frames Service Specification*. Issue 3. Recommendation for Space Data System Standards (Blue Book), CCSDS 911.1-B-3. Washington, D.C.: CCSDS, January 2010.
- [5] *Space Link Extension—Return Channel Frames Service Specification*. Issue 2. Recommendation for Space Data System Standards (Blue Book), CCSDS 911.2-B-2. Washington, D.C.: CCSDS, January 2010.
- [6] *Space Link Extension—Return Operational Control Fields Service Specification*. Issue 2. Recommendation for Space Data System Standards (Blue Book), CCSDS 911.5-B-2. Washington, D.C.: CCSDS, January 2010.
- [7] *Space Link Extension—Forward CLTU Service Specification*. Issue 3. Recommendation for Space Data System Standards (Blue Book), CCSDS 912.1-B-3. Washington, D.C.: CCSDS, July 2010.
- [8] *Space Link Extension—Forward Space Packet Service Specification*. Issue 2. Recommendation for Space Data System Standards (Blue Book), CCSDS 912.3-B-2. Washington, D.C.: CCSDS, July 2010.
- [9] *Space Link Extension—Internet Protocol for Transfer Services*. Issue 2. Recommendation for Space Data System Standards (Blue Book), CCSDS 913.1-B-2. Washington, D.C.: CCSDS, September 2015.
- [10] *Space Link Extension—Application Program Interface for Return All Frames Service*. Issue 2. Recommendation for Space Data System Practices (Magenta Book), CCSDS 915.1-M-2. Washington, D.C.: CCSDS, September 2015.
- [11] *Space Link Extension—Application Program Interface for Return Channel Frames Service*. Issue 2. Recommendation for Space Data System Practices (Magenta Book), CCSDS 915.2-M-2. Washington, D.C.: CCSDS, September 2015.
- [12] *Space Link Extension—Application Program Interface for Return Operational Control Fields Service*. Issue 2. Recommendation for Space Data System Practices (Magenta Book), CCSDS 915.5-M-2. Washington, D.C.: CCSDS, September 2015.

- [13] *Space Link Extension—Application Program Interface for the Forward CLTU Service. Issue 2. Recommendation for Space Data System Practices (Magenta Book)*, CCSDS 916.1-M-2. Washington, D.C.: CCSDS, September 2015.
- [14] *Space Link Extension—Application Program Interface for the Forward Space Packet Service. Issue 2. Recommendation for Space Data System Practices (Magenta Book)*, CCSDS 916.3-M-2. Washington, D.C.: CCSDS, September 2015.
- [15] *Information Technology—Abstract Syntax Notation One (ASN.1): Specification of Basic Notation*. 4th ed. International Standard, ISO/IEC 8824-1:2008. Geneva: ISO, 2008.
- [16] *Information Technology—ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. 4th ed. International Standard, ISO/IEC 8825-1:2002. Geneva: ISO, 2008.
- [17] *Information Technology—Open Systems Interconnection—The Directory—Part 2: Models*. 7th ed. International Standard, ISO/IEC 9594-2:2014. Geneva: ISO, 2014.
- [18] *Information Technology—Open Systems Interconnection—The Directory—Part 8: Public-Key and Attribute Certificate Frameworks*. 7th ed. International Standard, ISO/IEC 9594-8:2014. Geneva: ISO, 2014.
- [19] *Information Technology—Text Communication—Message-Oriented Text Interchange Systems (MOTIS)—Part 3: Abstract Service Definition Conventions*. International Standard, ISO/IEC 10021-3:1990 [Withdrawn]. Geneva: ISO, 1990.
- [20] *Programming Languages—C++*. 3rd ed. International Standard, ISO/IEC 14882:2011. Geneva: ISO, 2011.

2 DESCRIPTION OF THE SLE API

2.1 INTRODUCTION

2.1.1 SCOPE OF THE MODEL

The intention of this section is to provide a high-level yet precise description of the API covering all API components and their interaction. For this purpose, the section uses an object model presented in the Unified Modeling Language (UML). Detailed specifications for each of the components are provided in section 3, which references the concepts, objects, and interfaces described by this model.

The material presented here is an API design, to the extent that the API is broken down into components and the interfaces and interactions of these components are specified. However, this model (i.e., design) is restricted to what must be defined to ensure co-operation between components and excludes specification of the internal design of components.

The model defines:

- a) the runtime components, from which the API is constructed;
- b) the externally visible logical architecture of the API in terms of:
 - 1) the interfaces that are exposed by the components;
 - 2) the functionality to which these interfaces provide access; and
 - 3) the behavior of the operations defined by the interfaces.

In order to specify the externally visible architecture, the model defines logical entities below the level of runtime components. These entities are to be understood as abstract analysis objects. It is not the intention to prescribe the structure defined by these objects for an implementation in any way. The only requirement for an implementation is to provide the interfaces specified with the functionality and the behavior described by the analysis objects.

Some minor semantic extensions to UML have been defined to highlight the difference between those aspects of the model that must be implemented as specified and those aspects that are required only for a complete and unambiguous description. Subsection 2.2 provides details of how UML is used in this model.

This section contains only a summary description of interfaces. A complete specification of the methods and types is provided in annex A for all interfaces that are not service type specific. Service type-specific interfaces are detailed in supplemental Recommended Practice documents defining service-specific APIs.

2.1.2 PRESENTATION OF THE MODEL

The API model is presented as follows:

Subsection 2.2 describes how UML is used for this model. It does not provide an introduction to UML. For a description of UML, the reader is asked to refer to the UML specification (see reference [J6]), or to one of the textbooks on the subject (see references [J7] and [J8]).

Subsection 2.3 describes the ‘logical view’. It contains a subsection for each of the API components:

- a) API Proxy (see 2.3.2);
- b) API Service Element (see 2.3.3);
- c) SLE Operations (see 2.3.6); and
- d) SLE Utilities (see 2.3.7).

Subsection 2.3.4 describes interfaces that must be implemented by more than one component and describes the application interface to the API. The logical view is complemented by annex F providing an overview of how the components interact.

2.2 SPECIFICATION METHOD AND NOTATION

2.2.1 INTRODUCTION

The architectural model for the SLE API is defined using the Unified Modeling Language (UML) as defined in reference [J6]. This subsection describes some specific aspects of how UML is used in this Recommended Practice.

A component in UML models a runtime object, e.g., an executable file, a dynamically linked library, or similar operating system objects. Therefore the relationships that can be defined for a component in UML are limited:

- a) a component can implement (‘realize’) and export an interface;
- b) a component can depend on another component (more precisely on the interface exported by another component).

In this Recommended Practice, the UML component is used to refer to a component that:

- a) is delivered as one or more linkable libraries;
- b) is instantiated by a global ‘creator function’ defined in annex D;
- c) is substitutable by a different component providing the same interfaces.

API FOR SLE TRANSFER SERVICES—CORE SPECIFICATION

This specification requires these characteristics only for the top-level components API Proxy, API Service Element, SLE Operations, and SLE Utilities. These components are considerably complex and provide a rather large number of interfaces. In order to specify these interfaces, additional model constructs are needed. Following the general UML approach, this Recommended Practice uses UML classes with specific stereotypes to define special model objects. The specialized model objects are:

- a) Interface;
- b) Component Class (CoClass);
- c) Component Internal Class; and
- d) Entity.

They are shown in figure 2-1 together with some important relationships addressed later in this section. In addition, the model uses the UML utility class to represent functions that are not bound to any specific class.

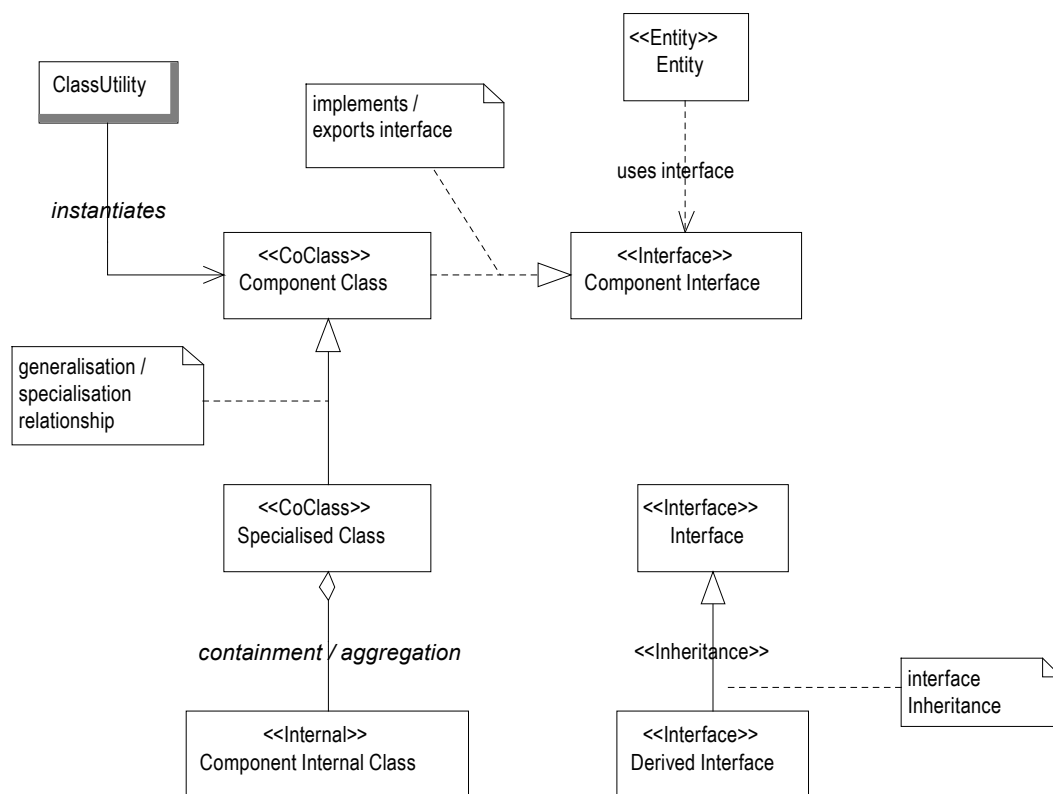


Figure 2-1: UML Stereotypes Used in This Recommended Practice

2.2.2 INTERFACE

The stereotype ‘Interface’ is defined by the UML specification. In this model it is used to identify a component interface. In C++ an interface is implemented as a class containing no data members and only public, pure virtual function members. According to the simple component model defined in annex D, all interfaces inherit the interface `IUnknown`. This fact is not explicitly shown in the diagrams.

An interface is displayed as a UML class with the stereotype `<<Interface>>`. The operations defined by the interface may or may not be shown, depending on the purpose of the specific diagram.

Where explicit public interface inheritance is required, this is indicated by the stereotype `<<Inheritance>>`. Generalization relationships that do not show this stereotype do not require an implementation using inheritance.

2.2.3 COMPONENT CLASS

A component class is a model object that specifies some functionality to be provided by a component. It is also used to describe navigational relationships between interfaces. The only implementation requirements related to component classes are the following. (For a description of the interface `IUnknown` and the method `QueryInterface()` see the ‘Simple Component Model’ in annex D.)

- a) A component must export all interfaces specified for a component class it implements.
- b) It must be possible to navigate between all interfaces specified for a component class and for component classes to which a generalization interface exists using `QueryInterface()`.
- c) For every non-abstract component class (except the ‘main’ class for a component) the model defines one (or more) interfaces by which a new instance can be obtained. These interfaces must be supported.
- d) When more than one instance of a component class exist, distinct references for the associated interfaces must be provided. The general requirement of the component model, that a query for the interface `IUnknown` on the same instance always returns the same pointer, applies.

Beyond these requirements, this Recommended Practice does not prescribe how the functionality defined for component classes is implemented. In particular, the generalization relationships shown in the model do not require implementation via inheritance. In fact, there need not be any equivalence between the classes within a component and the component classes shown in this model.

API FOR SLE TRANSFER SERVICES—CORE SPECIFICATION

A component class is defined as abstract, when no instances of the class are created. Such component classes define common functionality, behavior, and interfaces that are provided by more than one derived class.

A runtime component contains a single ‘main’ class and exposes a special ‘creator function’ that can create an instance of that class. This creator function must be a global symbol in the library that implements the component. In the diagrams of this model the creator function is represented by a UML Utility Class, which has an association ‘instantiates’ to the ‘main’ class.

The model uses the UML dependency (or ‘uses’) relationship between component classes and interfaces to describe how components are linked via their interfaces. The only requirement for an implementation is that the component implementing the functionality associated with the component class use the specified interfaces for the purpose identified in the model.

In a few cases, attributes are shown for component classes. Attributes are strictly analysis-model constructs to highlight characteristics of a class or options provided by a class. They are not to be understood to define data. Attributes shown in the model may not even be accessible at all.

A component class is displayed as a UML class with the stereotype <<CoClass>>. If the component class is abstract, its name is displayed in *italic typeface*.

2.2.4 COMPONENT INTERNAL CLASS

Component internal classes are used to describe features that are expected from a component, but which do not result in any externally visible interface. Component internal classes are pure model objects. This specification does not prescribe how the features presented by these objects are implemented.

A component internal class is presented as a UML class with the stereotype <<Internal>>. An internal class does not implement an interface. Beside this constraint, all relationships for classes can be used.

2.2.5 ENTITY

In some cases, it is necessary to identify use of an interface by some entity, which is otherwise unspecified. For this purpose, the model object ‘Entity’ is used. An entity is displayed as a UML class with the stereotype <<Entity>>. The only relationship an entity can have is a dependency relationship to an interface. No further semantics are associated with an entity.

2.2.6 NAMING CONVENTIONS

2.2.6.1 Component Classes

Because component classes are not expected to be visible in source code, their names do not adhere to the syntax of identifiers in programming languages.

The names of component classes that are independent of service types are not specifically prefixed. Names of classes for which a special version must be provided for every service type are prefixed with '<SRV>'.

2.2.6.2 Interfaces

Interfaces adhere to the syntax of C/C++ identifiers. Except for diagrams, interface names and method names are displayed in `mono-space font`.

Following Component Object Model (COM) conventions (see reference [J5]), the name of an interface always starts with a capital 'I'. Interfaces that are independent of specific SLE service types are prefixed with `ISLE_`. Names of interfaces, which are specific for service types, are prefixed with `I<SRV>`. These interfaces are defined in supplemental Recommended Practice documents for service-specific APIs, where `<SRV>` is replaced by the abbreviation for the service type. Readability of the name following the prefix can be improved using upper and lower case letters. The underscore character is reserved for separation of prefixes from the name. It is not used in the name itself.

Examples: `ISLE_ProxyAdmin`
 `ISLE_ServiceInform`
 `I<SRV>_SIAdmin` becomes, e.g., `IFSP_SIAdmin` or `IRAF_SIAdmin`

2.2.6.3 Entities and Component Internal Classes

Because the objects are pure modeling constructs and are not expected to be visible in source code, their names do not adhere to the syntax of identifiers in programming languages and no special naming conventions are applied.

2.2.7 DYNAMIC MODELING

The API requires implementation of a number of state machines. Because these comprise a considerable number of states and events, this Recommended Practice uses state tables instead of the state diagrams foreseen by UML. Because implementation of these state tables is mandatory, they have been placed in the prescriptive part of this Recommended Practice.

2.3 LOGICAL VIEW

2.3.1 STRUCTURE

The logical view of the API is partitioned into the packages shown in figure 2-2, which also displays the dependencies between these packages. Dependencies on the package SLE Utilities are not shown in order to avoid overloading of the diagram.

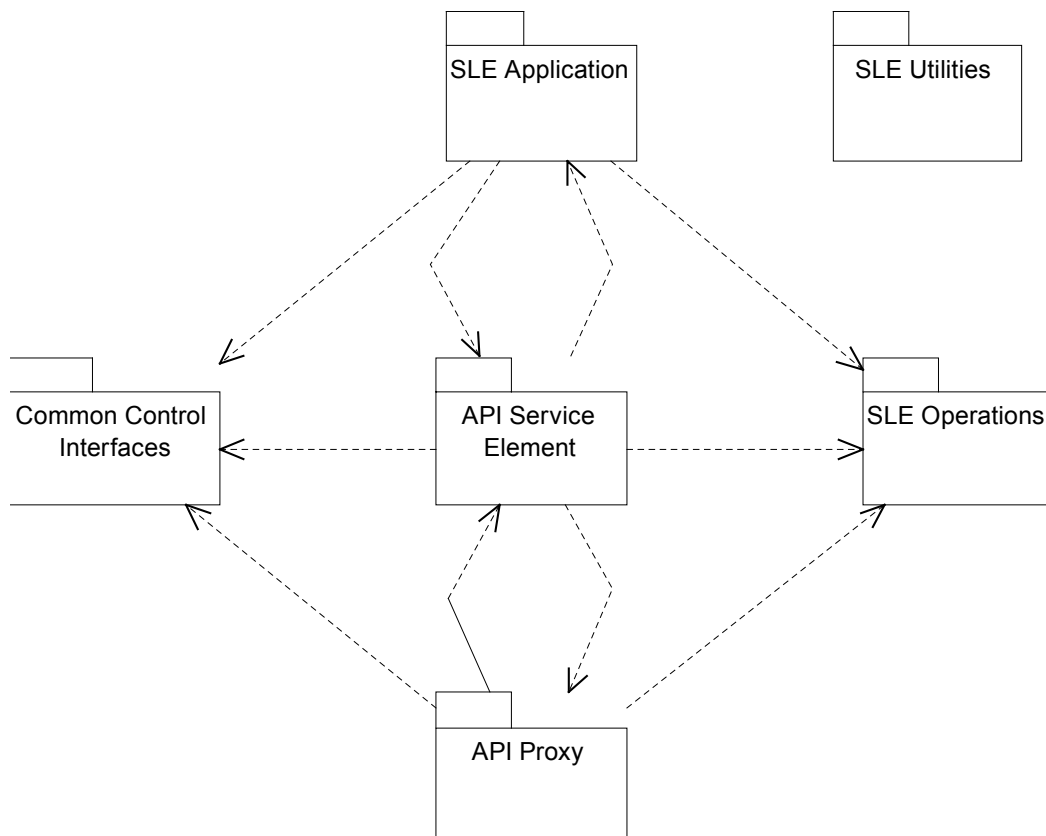


Figure 2-2: Top Level Decomposition of the API

The following packages exist:

- a) **API Proxy**
The package contains the component classes that define the component API Proxy as well as interfaces exported only by the Proxy.
- b) **API Service Element**
The package contains the component classes that define the component API Service Element as well as interfaces exported only by the API Service Element.
- c) **Common Control Interfaces**
The package specifies some interfaces that are supported by the API Proxy and the API Service Element.

- d) SLE Application
The package defines component classes that are assumed to be part of the application software. These component classes export the interfaces that must be made available by the application.
- e) SLE Operations
The package specifies the interfaces for operation objects implemented by the associated component and used for SLE transfer service interfaces.
- f) SLE Utilities
The package defines a small set of generally useful classes and their interfaces. The interfaces of these utility classes are used by interfaces throughout the model.

2.3.2 PACKAGE API PROXY

2.3.2.1 Overview

The API Proxy provides all functionality that must be implemented in a technology-specific manner and shields its clients from all technology-specific aspects. In addition, the Proxy implements access control on system level and authentication of the peer identity. Its structure is shown in figure 2-3.

The component class API Proxy is responsible for configuration, initialization, and management of the Proxy component and the data communication system. The configuration and initialization is performed using the interface `ISLE_ProxyAdmin`.

Communication between an SLE service user and an SLE service provider is handled by the class Association via the exported interface `ISLE_SrvProxyInitiate` and the complementary interface `ISLE_SrvProxyInform` supplied by the client. Associations can be created via the interface `ISLE_AssocFactory`.

Associations are distinguished according to the role they play in the BIND and UNBIND operation. Initiating associations invoke BIND and UNBIND operations, whereas responding associations accept incoming BIND and UNBIND invocations. These specialized classes differ in their behavior but do not expose any interfaces in addition to those inherited from the abstract class Association.

The PDU Translator is responsible for translation of the operation parameters between the syntax defined for the API and the syntax and encoding used for transfer. The abstract class PDU Translator handles common PDUs for association management, while a service-specific translator handles service-specific PDUs. It is the only element in the proxy that depends on the SLE service type.

The proxy and associations support logging and diagnostic traces using the interfaces `ISLE_Reporter` and `ISLE_Trace` provided by the application. Diagnostic traces can be switched on and off via the interface `ISLE_TraceControl` exported by the API Proxy and by the class Association.

API FOR SLE TRANSFER SERVICES—CORE SPECIFICATION

NOTE – All classes in the package API Proxy use the interfaces of operation objects and of utility objects. This fact is not specifically mentioned in the following description.

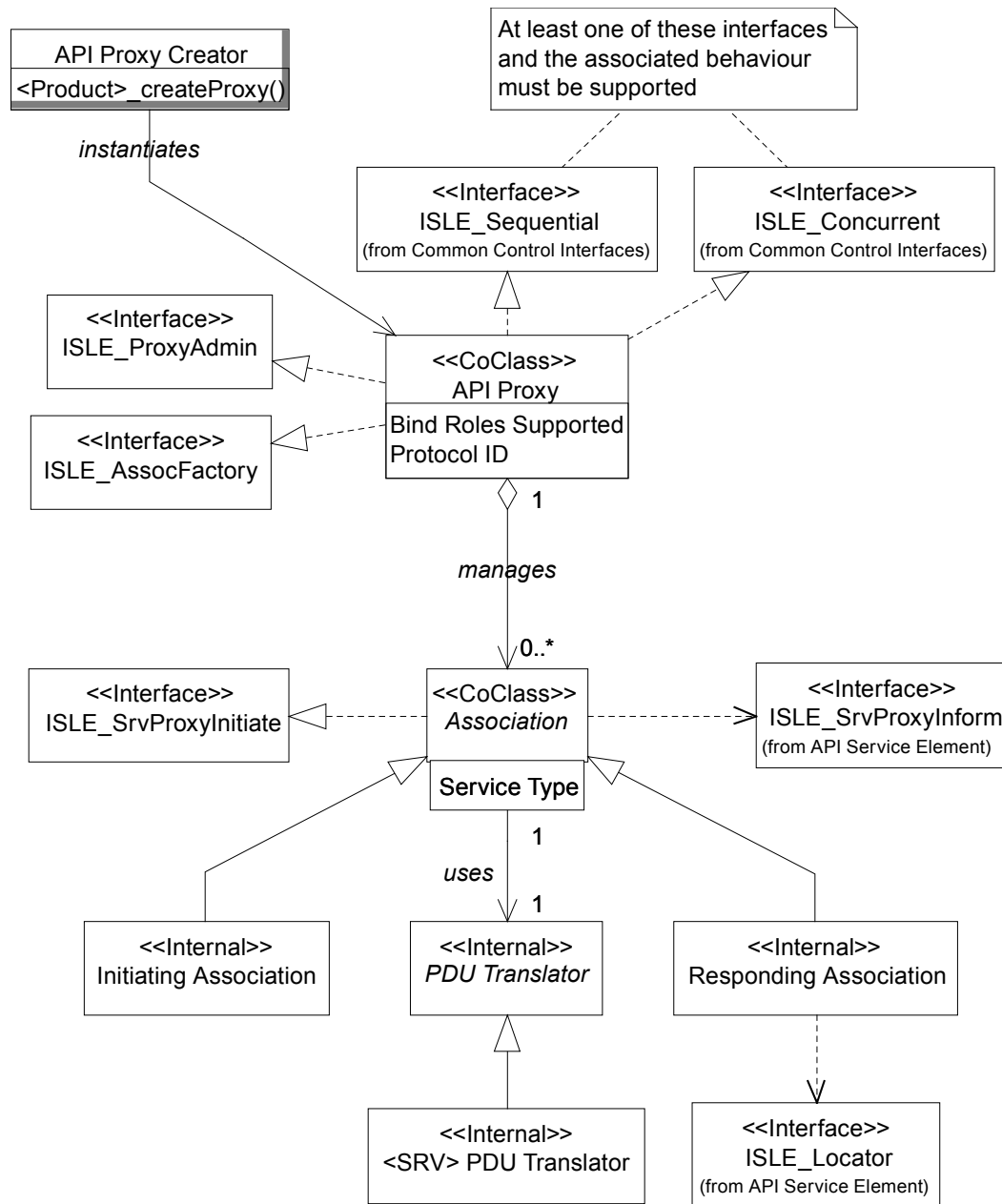


Figure 2-3: Structure of the Package API Proxy

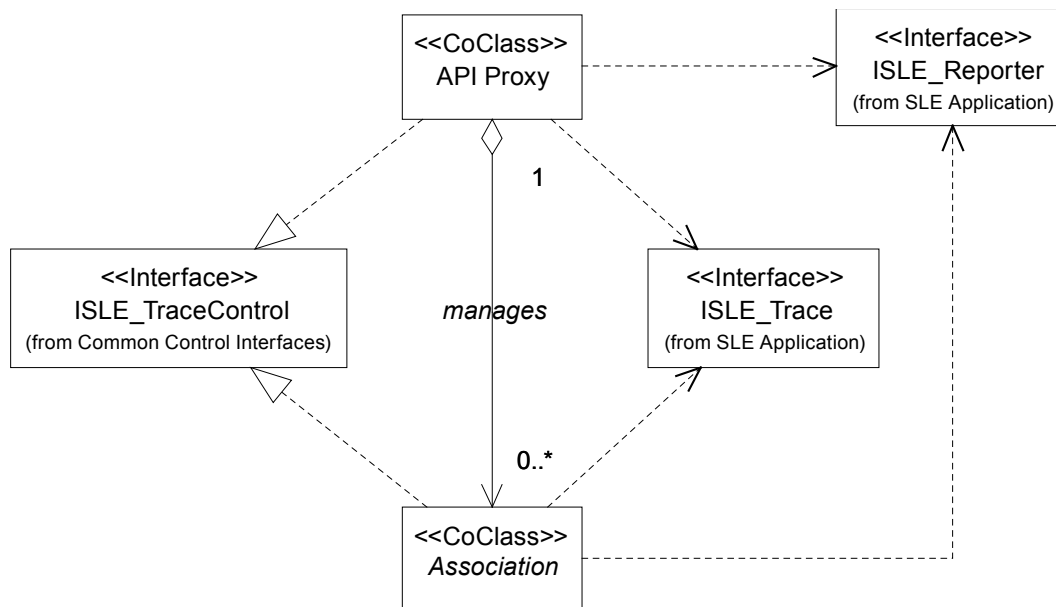


Figure 2-4: Reporting and Tracing by the Proxy

2.3.2.2 Component Class API Proxy

2.3.2.2.1 General

The API Proxy provides management of the communications infrastructure and of associations. The communications technology and the specific mapping of SLE transfer services to that technology by a proxy implementation is identified by a 'Protocol ID', available via the interface ISLE_ProxyAdmin.

In general, a proxy supports initiating associations and responding associations. However, an implementation may support only one of these roles. The 'bind roles' actually supported are defined by the attribute 'Bind Roles Supported'. A proxy supporting associations in the responder role listens for incoming connection requests on the network interface.

A single instance of this class exists within one instance of the component API Proxy.

2.3.2.2.2 Responsibilities

2.3.2.2.2.1 Configuration and Initialization of the Proxy Component

After creation the proxy must be configured and initialized using the interface ISLE_ProxyAdmin. This action includes configuration and initialization of the communications infrastructure. All static configuration parameters needed for this purpose are specified in the configuration database, defined in 2.3.2.8.

An implementation may require that part of the required infrastructure has already been initialized, e.g., at system start-up, or that a global infrastructure exists, e.g., for access to a directory system. Such prerequisites must be documented for every implementation.

2.3.2.2.2.2 Dynamic Port Registration and De-registration

The proxy performs dynamic registration of responder ports on request of a client via the interface `ISLE_ProxyAdmin`. Port registration includes all actions that may be required by a technology to register addresses, export address related information to a directory system, or publish a port by other means. Port registration has the effect that requests sent to the port are correctly routed to the proxy that registered it.

If port registration is not needed for the technology used, or for the current configuration of the proxy, the proxy ignores the request.

2.3.2.2.2.3 Management of Initiating Associations

The proxy creates and initializes initiating associations for a specified service type on request of its client via the interface `ISLE_AssocFactory`. If the proxy does not support the requested service type or does not support associations in the initiator role, it rejects the request.

The proxy keeps a reference to the associations until the client requests it to destroy the association. If the association is not in an unbound state, the proxy rejects this request. Otherwise, it releases all resources that may be allocated to the association and performs all actions required to delete the association.

2.3.2.2.2.4 Management of Responding Associations

A proxy that supports associations in the role of a BIND responder starts listening for incoming BIND invocations as soon as the start method of one of the control interfaces has been called, or when the port has been registered. When the proxy receives a BIND invocation, it creates a new responding association to process the BIND invocation.

When responding association terminates (following UNBIND, PEER-ABORT, or after a failure reported by the data communication service), the proxy releases all resources allocated to the association and deletes the association object.

2.3.2.2.2.5 Logging and Notification

The proxy generates log records for important events and enters them to the system log using the interface `ISLE_Reporter` provided by the application. For specific events that require immediate attention, the proxy notifies the application using the method `Notify()` in the interface `ISLE_Reporter`.

2.3.2.2.2.6 Diagnostic Traces

The proxy generates trace records for events that are not related to any particular association and passes them to the interface `ISLE_Trace` provided by the application. It supports the interface `ISLE_TraceControl` to switch tracing on and off. The proxy forwards all requests received via this interface to all associations currently managed.

2.3.2.2.3 Attributes

2.3.2.2.3.1 Protocol ID

Identifies the technology and specific mapping of SLE transfer services to that technology.

2.3.2.2.3.2 Bind Roles Supported

INITIATOR	the proxy is capable to support associations in the role of a bind initiator;
RESPONDER	the proxy is capable to support associations in the role of a bind responder;
ALL	the proxy is capable to support associations in the role of an initiator as well as associations in the role of a responder.

2.3.2.2.4 Behavior and Use

When the method `Configure()` is called on the interface `ISLE_ProxyAdmin`, the proxy checks the information passed and performs all actions required for configuration of the proxy and the communications service. Errors are logged and result in an error code returned to the caller. When the component has been configured successfully, the proxy returns a positive result code, indicating that it is ready for operation. However, it starts processing only when the start method is called on one of the control interfaces `ISLE_Sequential` or `ISLE_Concurrent` (see 2.3.4). This implies that a proxy supporting associations in the responder role starts listening for incoming BIND invocations only after call of the start method.

When the terminate method of the control interface is called, the proxy terminates all threads, if applicable, such that an orderly termination of the application is possible. If any associations are still active when termination is requested, the proxy aborts these associations. A proxy must expect that other proxies using the same communication infrastructure exist on the system and must make sure that their operation is not affected by termination activities.

NOTE – The terminate method is either `TerminateSequential()` of the interface `ISLE_Sequential`, or `TerminateConcurrent()` of the interface `ISLE_Concurrent`, depending on the behavior supported by the proxy.

API FOR SLE TRANSFER SERVICES—CORE SPECIFICATION

The proxy provides the method `ShutDown()` to shut it down on its administrative interface `ISLE_ProxyAdmin`. When that method is called it releases all interfaces of other components it still holds, frees all resources, and deletes all internal objects.

2.3.2.2.5 Exported Interfaces

Interface	Defined in Package	Purpose
<code>ISLE_ProxyAdmin</code>	API Proxy	configuration, initialization, dynamic port registration, and shutdown
<code>ISLE_Concurrent</code>	Common Control Interfaces	start and termination of operations for concurrent behavior
<code>ISLE_Sequential</code>	Common Control Interfaces	start and termination of operations for sequential behavior
<code>ISLE_AssocFactory</code>	API Proxy	creation and deletion of associations in the initiator role
<code>ISLE_TraceControl</code>	Common Control Interfaces	start and stop of diagnostic traces

2.3.2.2.6 Dependencies

Interface	Defined in Package	Purpose
<code>ISLE_Reporter</code>	SLE Application	logging and notification
<code>ISLE_Trace</code>	SLE Application	tracing
<code>ISLE_Locator</code>	API Service Element	indication of an incoming BIND invocation to the client (not shown in the diagram)
<code>ISLE_OperationFactory</code>	SLE Operations	creation of operation objects (not shown in the diagram)
<code>ISLE_UtilFactory</code>	SLE Utilities	creation of utility objects (not shown in the diagram)

2.3.2.3 Component Class Association

2.3.2.3.1 General

An object of a class derived from the abstract class `Association` handles a single data communication association between an SLE service user and an SLE service provider. The class `Association` defines those aspects of an association, which are independent of the role it plays in the BIND and UNBIND operation. An association is independent of the SLE service type. Service type-specific aspects are handled by the class `PDU Translator`, to which the association passes all operation invocations and returns for checking and for encoding and decoding.

An association object does not distinguish between the SLE service user role and the SLE service provider role and accepts any PDU that is defined for a given SLE service type. Checking for validity of PDUs for a given role must be performed by the client of the association.

2.3.2.3.2 Responsibilities

2.3.2.3.2.1 Mapping of Port Identifiers

The association maps the logical port identifiers defined by the CCSDS Recommended Standards for SLE transfer services to technology-specific addresses. The mapping is defined in the configuration database.

2.3.2.3.2.2 Processing of SLE Protocol Data Units

Associations accept operation objects holding SLE invocation and return parameters via the interface `ISLE_SrvProxyInitiate`, pass them to the PDU Translator for checking and encoding, and transfer the encoded PDU to the peer proxy. They receive SLE PDUs from the peer proxy, pass them to the PDU Translator for checking and decoding, and forward the resulting operation object to the client via the interface `ISLE_SrvProxyInform`.

2.3.2.3.2.3 Basic SLE Protocol Execution

Association objects implement a basic subset of the state tables defined for SLE services. The state table for associations is specified in section 4.

2.3.2.3.2.4 Authentication

For incoming PDUs, the association determines the required authentication mode defined in the configuration database for the peer application. If authentication is required it uses the interface `ISLE_SecAttributes` provided by the component SLE Utilities to check the credentials transmitted in the PDU. If authentication fails the association ignores the PDU. For outgoing PDUs, the association generates the credentials using the security attributes of the local application in its configuration database.

2.3.2.3.2.5 Monitoring of the State of the Data Communication Connection

The association monitors the state of the data communication connection it uses and informs its client if the connection breaks down. The maximum delay between the failure and the report is specified in the proxy configuration file.

2.3.2.3.2.6 Queuing of Outbound PDUs

The association queues a maximum number of PDUs if these cannot be transmitted immediately. A positive response to a transmission request by the client guarantees that the PDU has been queued for transmission. The maximum size of the queue is defined in the proxy configuration file. If requested by the client (see the interface `ISLE_SrvProxyInitiate`) the association notifies the client when a PDU has actually been transmitted. If the queue of outgoing PDUs is full, the association rejects further transmission requests.

2.3.2.3.2.7 Removal of Transfer Buffer PDUs

On request by the client, the association removes all PDUs of the type `TRANSFER-BUFFER` for which transmission has not yet started from the queue and releases associated resources. It informs the client whether such PDUs have been discarded.

2.3.2.3.2.8 Limiting Inbound Data Traffic

The association ensures that the number of PDUs received from the network and not yet passed to its client does not exceed a maximum number $N1$ defined in the proxy configuration file. Of these a maximum number $N2 \leq N1$ are allowed to be PDUs of the type `TRANSFER-DATA` invocation or `TRANSFER-BUFFER` invocation. The number $N2$ is also defined in the proxy configuration file. If either of these limits is exceeded the association does not accept further data from the network making sure that back-pressure is built up.

2.3.2.3.2.9 Logging and Notification

The association and its derived classes generate log records for important events and enter them to the system log using the interface `ISLE_Reporter` provided by the application. For specific events that require immediate attention, the association notifies the application using the method `Notify()` in the interface `ISLE_Reporter`.

2.3.2.3.2.10 Diagnostic Traces

The class `Association` and its derived classes generate trace records and pass them to the interface `ISLE_Trace` provided by the application. It supports the interface `ISLE_TraceControl` to switch tracing on and off.

2.3.2.3.3 Exported Interfaces

Interface	Defined in Package	Purpose
ISLE_SrvProxyInitiate	API Proxy	passing of SLE PDUs for transfer
ISLE_TraceControl	Common Control Interfaces	start and stop of diagnostic traces

2.3.2.3.4 Dependencies

Interface	Defined in Package	Purpose
ISLE_SrvProxyInform	API Service Element	passing of SLE PDUs received from the network
ISLE_Reporter	SLE Application	logging and notification
ISLE_Trace	SLE Application	tracing

2.3.2.4 Internal Class Initiating Association

2.3.2.4.1 General

An initiating association accepts requests to invoke the BIND operation and the UNBIND operation from its local client. If the association receives a BIND or UNBIND invocation PDU from the peer proxy, it aborts the data communication association with the diagnostic ‘protocol error’.

2.3.2.4.2 Responsibilities

2.3.2.4.2.1 Association Establishment

When receiving a BIND invocation from its local client, the initiating association establishes a data communication association with the peer proxy using technology-specific means and transmits the BIND invocation. It completes the association establishment procedure when it receives the BIND return from the peer proxy. If the BIND return PDU contains a positive result, the association is established and the state is set to ‘bound’. If the BIND return PDU carries a negative result, the association is not established and the state is set to ‘unbound’. The association informs its client by forwarding the operation object with the return parameters received from the peer proxy.

The association ensures that the BIND operation is not performed on an established association or during association release and is not re-invoked during association establishment. It also ensures that the BIND operation is performed according to the protocol defined by the CCSDS Recommended Standards for SLE transfer services.

2.3.2.4.2.2 Association Release

When receiving an UNBIND invocation from its local client, the association object forwards the invocation to the peer proxy and initiates termination of the data communication association by technology-specific means. It completes the association release procedure when receiving the UNBIND return, by setting its state to 'unbound' and forwarding the return to its client.

The association ensures that the UNBIND operation is performed only on an established association and is not re-invoked during association release. It also ensures that UNBIND operation is performed according to the protocol defined by the CCSDS Recommended Standards for SLE transfer services.

2.3.2.4.2.3 Access Control

As part of the BIND operation the initiating association locates the peer application in the configuration database of the proxy using the responder identifier in the BIND operation object. When receiving the BIND return it verifies that the responder identifier is the one expected and aborts if that is not the case. This test is performed before authentication, if authentication is required for the peer application.

2.3.2.5 Internal Class Responding Association

2.3.2.5.1 General

A responding association processes BIND invocations received from the peer proxy and responds to UNBIND invocations issued by the peer proxy. It rejects any BIND or UNBIND invocations that might be requested by its local client.

2.3.2.5.2 Responsibilities

2.3.2.5.2.1 Association Establishment

An object of the class Responding Association is created by the API Proxy in order to process a received BIND invocation (see 2.3.2.2.4). It performs all initial checks on the BIND invocation defined in this section. It then informs its client via the interface `ISLE_Locator` provided as part of the proxy configuration, passing it a reference to its interface `ISLE_SrvProxyInitiate` and to the operation object holding the BIND invocation parameters.

If the locator interface returns a reference to the interface `ISLE_SrvProxyInform`, the association forwards the BIND invocation via that interface. If the locator returns an error, the association generates a BIND return PDU with a negative result and a diagnostic corresponding to the error. It transmits the PDU to the peer proxy and terminates the data communication association.

The association completes the association establishment procedure when it receives the BIND return from its local client. If the BIND return PDU contains a positive result, the association is established and the state is set to 'bound'. If the BIND return PDU carries a negative result, the association is not established. In both cases, the association forwards the BIND return to the peer proxy.

If the association receives a BIND invocation from the peer proxy on the data communication association it handles, it aborts the association with the diagnostic 'protocol error'. The association also ensures that the BIND operation is performed according to the protocol defined by the CCSDS Recommended Standards for SLE transfer services.

2.3.2.5.2.2 Association Release

When receiving an UNBIND invocation from the peer proxy, the association forwards the invocation to its client. It completes the association release procedure when it receives the UNBIND return from its local client.

The association ensures that the UNBIND operation is performed only on an established association and is not re-invoked during association release. It also ensures that UNBIND operation is performed according to the protocol defined by the CCSDS Recommended Standards for SLE transfer services.

2.3.2.5.2.3 Access Control

When receiving a BIND invocation, the association verifies that the initiator is defined as a peer application in the configuration database of the proxy. If that is not the case, it responds with a BIND return containing a negative result and the diagnostic 'access denied'.

2.3.2.5.2.4 Handling of Service Types and Version Numbers

When receiving a BIND invocation, the association checks that the requested service type is defined in the configuration database of the proxy and that the version number for that type can be supported. If that is not the case, it responds with a BIND return containing a negative result and the appropriate diagnostic.

2.3.2.5.3 Dependencies

Interface	Defined in Package	Purpose
ISLE_Locator	API Service Element	indication of an incoming BIND invocation to the client

2.3.2.6 Internal Class PDU Translator

2.3.2.6.1 General

The class PDU Translator translates SLE operation parameters between the syntax used in the API and the syntax required for communication with the peer proxy. The base class handles the SLE operations BIND, UNBIND, and PEER-ABORT, which are identical for all service types. Service-specific operations are handled by the derived classes.

2.3.2.6.2 Responsibilities

2.3.2.6.2.1 Association of Returns with Invocations

The PDU Translator receives operation objects from the association. For invocations of confirmed operations by the local client, it stores a reference to the operation object until the return from the peer proxy arrives or the association is aborted. When receiving a PDU that contains a return, the PDU Translator locates the operation object holding the invocation by means of the invocation identifier. The PDU Translator verifies that the invocation and return are of the same operation type. If it cannot locate the invocation, it informs the association, which aborts with the diagnostic ‘unsolicited invocation identifier’.

NOTE – The processing of invocations of confirmed operations received from the peer proxy is described in 2.3.2.6.2.3.

It is noted that the confirmed operations BIND and UNBIND do not carry an invocation identifier. Because only a single return can be outstanding for these operations at any time, association of the return with the invocation is possible without the invocation identifier.

2.3.2.6.2.2 Encoding of PDUs

A PDU Translator extracts the invocation or return parameters from the operation objects passed by the association, encodes them as required by the technology mapping, builds the protocol data unit, and passes it back to the association for transmission.

2.3.2.6.2.3 Decoding of PDUs

The PDU Translator decodes PDUs received from the peer proxy and extracts the operation parameters. For invocations, the PDU Translator creates an operation object using the interface `ISLE_OperationFactory`, stores the invocation parameters to this object, and passes it to the association for further processing. For returns, it stores the return parameters to the operation object, which holds the associated invocation.

2.3.2.7 Internal Class <SRV> PDU Translator

A class <SRV> PDU Translator exists for every SLE service type supported by the proxy. For operation objects and PDUs passed to the PDU Translator, the class checks whether these are defined for the service type. If the operation is defined for the service type, the class performs encoding and decoding for these operations as described in 2.3.2.6.

2.3.2.8 Proxy Configuration Database

2.3.2.8.1 General

Operation of the API Proxy in a specific deployment environment is controlled by parameters in a configuration database. The structure of this database is implementation specific. It could consist of one or more files or could be implemented using directory systems or some management database. The configuration file passed to the proxy as part of the configuration can contain the complete database or only a reference that enables the proxy to access the database.

Also, the content of the database is largely implementation specific. Information, which must be part of the configuration database, is presented in figure 2-5. The objects shown in the figure are not complete. Information objects not shown in the figure are represented by the attribute 'Configuration Parameters' of the class Proxy Config Database. A complete list of required objects may be found in section 3.

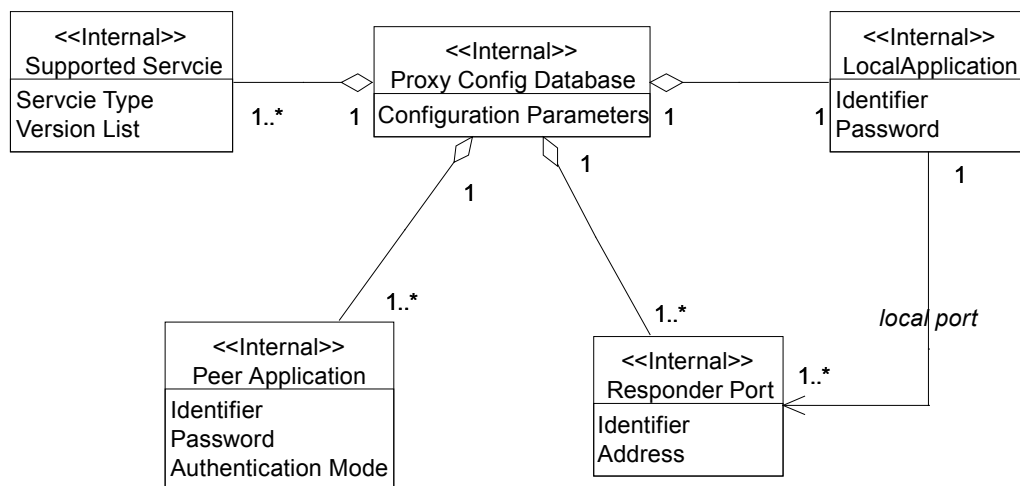


Figure 2-5: Configuration Database of the Proxy

2.3.2.8.2 Local Application

The database contains information concerning the local SLE application. The information includes the identifier (user name) and a password for authentication. For proxies supporting associations in the responder role, the database also holds information related to local ports, on which the proxy shall accept incoming BIND invocations. The number of ports that can be supported is implementation defined.

2.3.2.8.3 Peer Application

The database contains a list of peer applications, which are allowed to access the system. The list includes service users that may access a service provider and service providers, which the local application may use. For every peer application, the database contains the identifier, the required authentication mode (no authentication, authentication for BIND only, authentication of all PDUs) and a password for authentication.

2.3.2.8.4 Port

For all responder ports (local and remote) the database contains a logical port identifier and the technology-specific address information associated with that name.

2.3.2.8.5 Supported Services

The database contains a list of the service types that are supported by all API components in an installation, and for each type, the list of version numbers that are supported by all API components.

2.3.2.8.6 Interfaces Defined by the Package

Name	Description
ISLE_ProxyAdmin	The interface is provided to configure and initialize the proxy component passing it the pointers to interfaces of other components it needs operationally. In addition, the interface comprises the methods for port registration and de-registration and for shutting down of the proxy.
ISLE_AssocFactory	The interface supports creation of initiating association objects for a specified SLE transfer service type. A pointer to the client interface must be passed to the creation function. The interface also provides a method to request the proxy to destroy an association object that is no longer needed.
ISLE_SrvProxyInitiate	The interface provides methods to pass SLE operation invocations and returns for transmission. In addition, it supports the features to request reporting of actual transfer of a PDU, and to discard PDUs of the type TRANSFER-BUFFER. The interface is identical for all association roles (initiator and provider) and all SLE service types.

2.3.3 PACKAGE API SERVICE ELEMENT

2.3.3.1 Overview

The API Service Element implements functionality related to SLE transfer service provisioning, which can be clearly separated from service production. It provides support for SLE service provider applications and for SLE service user applications. The structure of the API Service Element is shown in figure 2-6.

The component class API Service Element is responsible for configuration, initialization, and management of the component. It provides an interface to the application to create and delete service instances (`ISLE_SIFactory`) and to the proxy to locate service instances when receiving a `BIND` invocation (`ISLE_Locator`).

API FOR SLE TRANSFER SERVICES—CORE SPECIFICATION

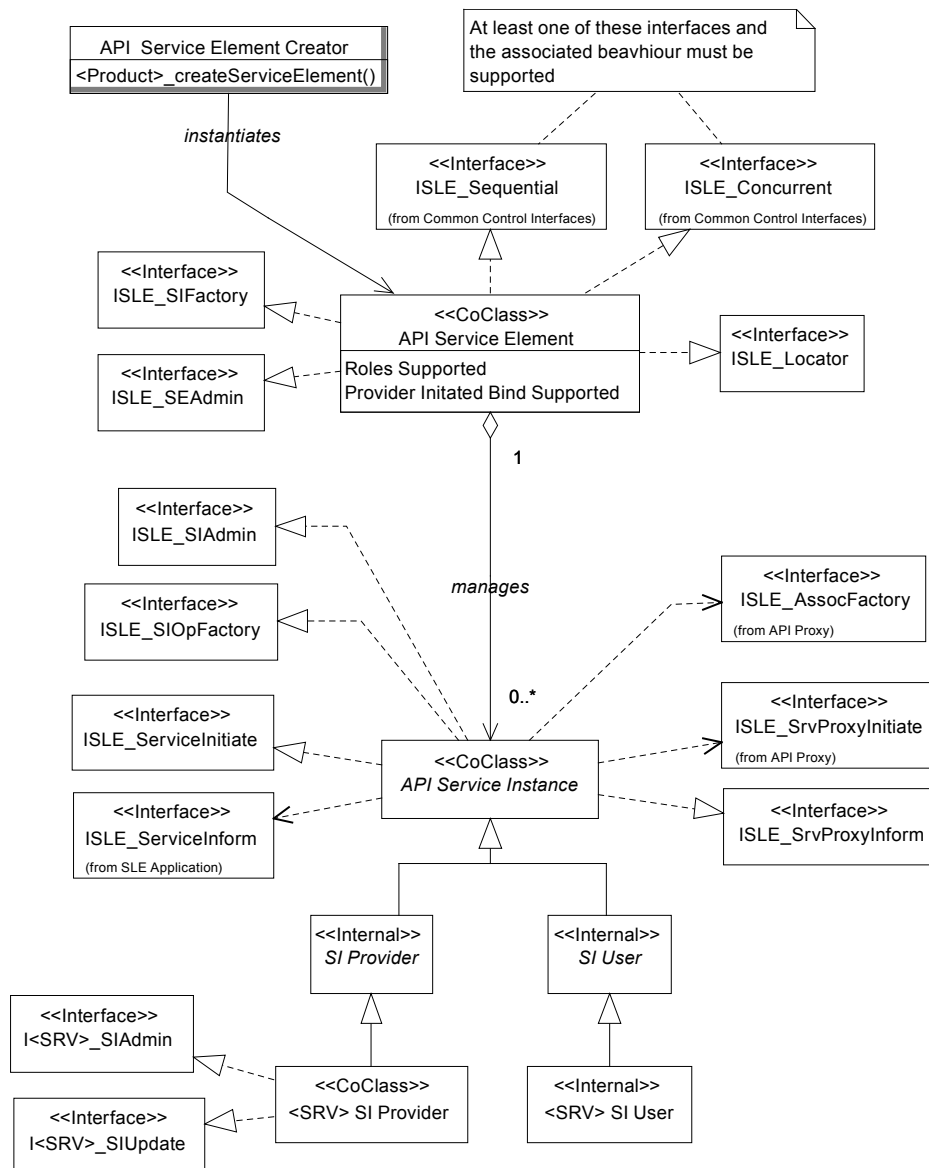


Figure 2-6: Structure of the Package API Service Element

Individual service instances are handled by the class API Service Instance. During periods in which an SLE service user and an SLE service provider communicate, the service instance is linked with an association object in the component API Proxy. It communicates with the association via the interface `ISLE_SrvProxyInitiate` and the complementary interface `ISLE_SrvProxyInform`. With the application it communicates via the interface `ISLE_ServiceInitiate` and the complementary interface `ISLE_ServiceInform`.

Service instances are distinguished according to the application role they support. The class `SI Provider` supports SLE service provider applications and the class `SI User` supports SLE

service user applications. While the behavior of these classes and the operations they support differ, the externally visible interfaces are identical.

The classes API Service Instance, SI Provider, and SI User are abstract. Service instance objects support a specific SLE service type represented by the classes <SRV> SI Provider and <SRV> SI User. The class <SRV> SI User does not export any new interfaces, while service instances in the provider role support additional service type-specific interfaces for configuration (I<SRV>_SIAdmin) and for update of service parameters (I<SRV>_SIUpdate).

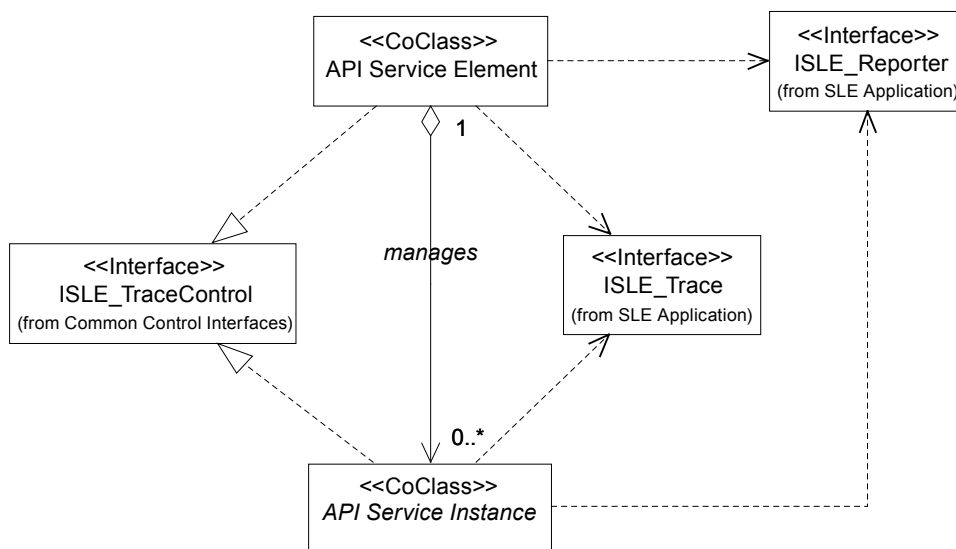


Figure 2-7: Reporting and Tracing by the Service Element

The service element and service instances support logging and diagnostic traces using the interfaces ISLE_Reporter and ISLE_Trace provided by the application. Diagnostic Traces can be switched on and off via the interface ISLE_TraceControl exported by the API Service Element and by the API Service Instance.

NOTE – All classes in the package API Service Element use the interfaces of operation objects and of utility objects. This fact is not specifically mentioned in the following description.

2.3.3.2 Component Class API Service Element

2.3.3.2.1 General

The component class API Service Element provides management of service instances. An implementation of the component can support service instances for the SLE service provider role and for the SLE service user role or only for one of the roles as indicated by the attribute 'Roles Supported'. Support for user-initiated binding of service instances is mandatory, while support for provider-initiated binding is an option. Its support is indicated by the attribute 'Provider Initiated Bind Supported'.

NOTE – This version of the Recommended Practice does not support provider-initiated binding (see 1.2.2 item b)). The option and the attribute 'Provider Initiated Bind Supported' are foreseen to allow later extension.

A single instance of this class exists within an instance of the API Service Element component.

2.3.3.2.2 Responsibilities

2.3.3.2.2.1 Configuration and Initialization of the API Service Element Component

After creation, the API Service Element must be configured and initialized using the interface `ISLE_SEAdmin`. All static configuration parameters needed for this purpose are specified in the configuration database defined in 2.3.3.8.

2.3.3.2.2.2 Control of Proxies

The service element can use several proxies distinguished by the 'Protocol ID' of the proxy. The interface `ISLE_SEAdmin` provides a method to link proxies to the service element after configuration. The service element starts and terminates operation of all linked proxies when its own operation is started or terminated.

For service instances that initiate the BIND operation, the service element selects the proxy to use by a table in its configuration database, which associates the peer port identifier with the Protocol ID supported by the proxy.

2.3.3.2.2.3 Management of Service Instances

The service element creates and initializes service instances for a specified service type and a specified role on request of the application via the interface `ISLE_SIFactory`. If the service element does not support the requested service type or role, it rejects the request. If the service instance shall initiate binding, the application must additionally specify the version number of the service type.

The service element keeps a reference to the service instances created until the application requests it to destroy the service instance. If the service instance is still bound at that time, the service element rejects the request. Otherwise, it releases all resources that are allocated to the service instance and performs all actions required to delete the service instance.

2.3.3.2.2.4 Location of Service Instances

The service element provides the interface `ISLE_Locator` to the proxy to locate requested service instances when the proxy receives a `BIND` invocation from the peer system. It uses the service instance identifier in the `BIND` invocation passed by the proxy to find the service instance. If the service instance has been created by the application, the service element verifies that this service instance is not already bound and that the `BIND` invocation parameters are consistent with the configuration of the service instance. If all checks are passed, it links the service instance with the association and returns a reference to the interface `ISLE_SrvProxyInform` of the service instance to the proxy. Otherwise, it returns an error, instructing the proxy to reject the `BIND` invocation.

2.3.3.2.2.5 Access Control

For incoming `BIND` invocations the service element verifies that the initiator identifier in the `BIND` invocation matches the peer identifier defined for this service instance. If that is not the case, it rejects the request and generates an access violation alarm.

2.3.3.2.2.6 Logging and Notification

The service element generates log records for important events and enters them to the system log using the interface `ISLE_Reporter` provided by the application. For events that require immediate attention, the service element notifies the application using the method `Notify()` in the interface `ISLE_Reporter`.

2.3.3.2.2.7 Diagnostic Traces

The service element generates trace records for events that are not related to any particular service instance and passes them to the interface `ISLE_Trace` provided by the application. It supports the interface `ISLE_TraceControl` to switch tracing on and off. The service element forwards all requests received via this interface to all service instances currently managed, and, if requested by the caller, to all proxies that it controls.

NOTE – The interface `ISLE_TraceControl` of the service element allows setting of the trace level on a global scope. Individual setting of the trace level of each service instance is possible using the interface `ISLE_TraceControl` of the service instance.

2.3.3.2.3 Attributes

2.3.3.2.3.1 Roles Supported

USER	the service element supports service instances in the role of an SLE service user;
PROVIDER	the service element supports service instances in the role of an SLE service provider;
ALL	the service element supports service instances in the role of a user as well as service instances in the role of a provider.

2.3.3.2.3.2 Provider Initiated Bind Supported

Indicates whether the service element supports provider-initiated binding of service instances.

NOTE – This version of the Recommended Practice does not support provider-initiated binding (see 1.2.2 item b)). The option and the attribute ‘Provider Initiated Bind Supported’ are foreseen to allow later extension.

2.3.3.2.4 Behavior and Use

When the method `Configure()` is called on the interface `ISLE_SEAdmin()`, the service element checks the information passed, and performs all actions required for configuration of the component. Errors are logged and result in an error code returned to the caller. When the component has been configured successfully, the service element returns a positive result code. Following configuration of the component, the service element must be linked with the proxies it will use. For this purpose, the service element provides the method `AddProxy()` in its administrative interface.

The service element starts processing when the start method is called on one of the control interfaces `ISLE_Sequential` or `ISLE_Concurrent` (see 2.3.4). It then also starts processing of all linked proxies using the interface selected for control of the proxy.

When the terminate method is called via the control interface, the service element terminates all threads, if applicable, such that an orderly termination of the application is possible. If any service instances are still active when termination is requested, the service element instructs them to abort the association. Finally the service element terminates processing of all linked proxies.

NOTE – The terminate method is either `TerminateSequential()` of the interface `ISLE_Sequential`, or `TerminateConcurrent()` of the interface `ISLE_Concurrent`, depending on the behavior supported by the service element.

The service element provides the method `ShutDown()` to shut it down on its administrative interface `ISLE_SEAdmin`. When that method is called it releases all interfaces of other components it still holds, frees all resources, and deletes all internal objects.

2.3.3.2.5 Exported Interfaces

Interface	Defined in Package	Purpose
<code>ISLE_SEAdmin</code>	API Service Element	configuration, initialization, and shutdown
<code>ISLE_Concurrent</code>	Common Control Interfaces	start and termination of operations for concurrent behavior
<code>ISLE_Sequential</code>	Common Control Interfaces	start and termination of operations for sequential behavior
<code>ISLE_SIFactory</code>	API Service Element	creation and deletion of service instances
<code>ISLE_Locator</code>	API Service Element	location of service instances requested by incoming BIND invocations
<code>ISLE_TraceControl</code>	Common Control Interfaces	start and stop of diagnostic traces

2.3.3.2.6 Dependencies

Interface	Defined in Package	Purpose
<code>ISLE_Reporter</code>	SLE Application	logging and notification
<code>ISLE_Trace</code>	SLE Application	tracing
<code>ISLE_OperationFactory</code>	SLE Operations	creation of operation objects (not shown in the diagram)
<code>ISLE_UtilFactory</code>	SLE Utilities	creation of utility objects (not shown in the diagram)
<code>ISLE_Concurrent</code>	Common Control Interfaces	start and termination of proxies for concurrent behavior (not shown in the diagram)
<code>ISLE_Sequential</code>	Common Control Interfaces	start and termination of proxies for sequential behavior (not shown in the diagram)
<code>ISLE_TraceControl</code>	Common Control Interfaces	start and stop of diagnostic traces of proxies (not shown in the diagram)

2.3.3.3 Component Class API Service Instance

2.3.3.3.1 General

An object of a class derived from the abstract class API Service Instance handles a single SLE transfer service instance. The class API Service Instance defines those aspects of a service instance, which are independent of the user and provider role and independent of a specific service type.

2.3.3.3.2 Responsibilities

2.3.3.3.2.1 Configuration of the Service Instance

The service instance exports the interface `ISLE_SIAAdmin` to set common configuration parameters after creation. When all parameters have been set, the method `ConfigCompleted()` must be called. The service instance then checks its configuration for completeness and consistency.

2.3.3.3.2.2 Control of Initiating Associations

If the service instance initiates binding, it creates the initiating association via the interface `ISLE_AssocFactory` exported by the component API Proxy. It selects the proxy instance from the mapping table in the configuration database of the service element, using the responder port identifier as a key. The service instance requests the proxy to destroy the association when it is no longer needed.

Implementations might create the association after configuration and keep it for the complete lifetime of the service instance or create a new association for every BIND invocation.

2.3.3.3.2.3 Processing of SLE Protocol Data Units

The service instance receives operation objects holding SLE PDUs from the application via the interface `ISLE_ServiceInitiate`. It verifies that the PDUs are valid in the current state and checks the parameters for completeness, consistency, and range. If all checks are passed, the service instance passes the operation objects to the association for transfer via the interface `ISLE_SrvProxyInitiate`. With a positive result code returned to the application, the service instance guarantees that the PDU has been accepted by the proxy.

The service instance receives operation objects holding SLE PDUs from the association via the interface `ISLE_SrvProxyInform`. It verifies that the PDUs are valid in the current state and checks the parameters for completeness, consistency, and range. If all checks are passed and the operation is not handled by the service instance itself, it passes the operation objects to the application via the interface `ISLE_ServiceInform`.

2.3.3.3.2.4 SLE Protocol Execution

The service instance enforces conformance to the state tables defined for SLE services to the extent that these are independent of service production. The state tables processed by service instances are specified in section 4.

2.3.3.3.2.5 Management of Invocation Identifiers

The service instance assigns unique invocation identifiers to operation objects for confirmed SLE operations. For invocations of confirmed operations received from the proxy, the service instance verifies that the invocation identifier is unique for all operations to which the application has not yet responded. If the service instance detects a duplicate invocation identifier, it responds with a return containing a negative response and the appropriate diagnostic.

It is noted that the confirmed operations BIND and UNBIND do not carry an invocation identifier and must be excluded from these checks.

2.3.3.3.2.6 Timeout Monitoring for Operation Returns

For confirmed operations invoked by the local application or by the service instance itself, the service instance ensures that a return is received within a timeout defined as a configuration parameter. If no return arrives within the specified timeout, it aborts the association.

2.3.3.3.2.7 Pre-setting of Operation Object Parameters

The service instance provides an interface for creation of operation objects for the service type supported. It uses the interface `ISLE_OperationFactory` exported by the component SLE Operations to create these objects and initializes the parameters of the operation objects according to its own configuration.

2.3.3.3.2.8 Logging and Notification

The class API Service Instance and its derived classes generate log records for important events and enter them to the system log using the interface `ISLE_Reporter` provided by the application. For events that require immediate attention, the service instance notifies the application using the method `Notify()` in the interface `ISLE_Reporter`.

2.3.3.3.2.9 Diagnostic Traces

The class API Service Instance and its derived classes generate trace records and pass them to the interface `ISLE_Trace` provided by the application. It supports the interface `ISLE_TraceControl` to switch tracing on and off. If requested by the caller, the service instance forwards the request to the associations, which it is using.

API FOR SLE TRANSFER SERVICES—CORE SPECIFICATION

2.3.3.3 Behavior

For handling of errors, the service instance applies the following rules:

- a) If SLE PDUs received from the application are not valid in the current state or fail to pass any of the other checks the service instance applies, it returns an error code to the method that passed the PDU.
- b) If a PDU received from the application is rejected by the association, the service instance returns the result code received from the association to the application. If the result code indicates that the queuing capacity of the association is exceeded, the service instance aborts the association. Because of the flow control mechanisms built into the API, queue overflow cannot be caused by transfer of space link data units. It can only happen because of excessive generation of other events related to the production process or excessively high status reporting frequencies. In these cases the application would have no other option to handle the problem.
- c) If SLE PDUs received from the association are not valid in the current state or fail to pass any of the other checks the service instance applies, the service instance proceeds as follows:
 - 1) if the problem is due to a misbehavior of the association, it returns an error code to the method that passed the PDU;
 - 2) if the PDU is an invocation of a confirmed operation, the service instance sets the result of the operation object to 'negative', inserts the appropriate diagnostic, and forwards it to the association for transfer;
 - 3) otherwise, the service instance aborts the association with the appropriate diagnostic.

2.3.3.3.4 Exported Interfaces

Interface	Defined in Package	Purpose
ISLE_SIAAdmin	API Service Element	configuration of the service instance
ISLE_SIOpFactory	API Service Element	creation and initialization of operation objects
ISLE_ServiceInitiate	API Service Element	passing of SLE PDUs from the application to the service instance
ISLE_SrvProxyInform	API Service Element	passing of SLE PDUs received from the association
ISLE_TraceControl	Common Control Interfaces	start and stop of diagnostic traces

2.3.3.3.5 Dependencies

Interface	Defined in Package	Purpose
ISLE_AssocFactory	API Proxy	creation and deletion of associations in the initiator role
ISLE_ServiceInform	SLE Application	passing of SLE PDUs to the application
ISLE_SrvProxyInitiate	API Proxy	passing of SLE PDUs to the association for transfer
ISLE_OperationFactory	SLE Operations	creation of operation objects (not shown in the diagram)
ISLE_TraceControl	Common Control Interfaces	start and stop of diagnostic traces of the association (not shown in the diagram)

2.3.3.4 Internal Class SI User

2.3.3.4.1 General

The class SI User defines those aspects of a service instance in the user role, which are independent of a specific SLE service type. This class does not export any additional interfaces.

2.3.3.4.2 Responsibilities

2.3.3.4.2.1 Processing of SLE Protocol Data Units

The service instance verifies that PDUs received from the association or from the application are compatible with the user role, the service type supported, and with the version number of the service.

2.3.3.4.2.2 Buffering for Return Services

For return services, the service instance accepts the TRANSFER-BUFFER operation object from the association, extracts the TRANSFER-DATA and SYNC-NOTIFY operation objects, and passes them to the application in the sequence they have been stored in the TRANSFER-BUFFER operation object. The service instance verifies that the buffer received only contains PDUs for which buffering shall be applied.

2.3.3.4.2.3 Flow Control for Forward Services

For forward services, the service instance provides flow control for TRANSFER-DATA invocations. When a maximum number of TRANSFER-DATA invocations have been queued by the association and not yet transmitted, the service instance returns a code to the application requesting it to suspend data transfer. It informs the application when data transmission can be resumed via the method `ResumeDataTransfer()` of the interface `ISLE_ServiceInform`. The number of invocations that can be queued is defined by the implementation or can be set in the configuration database of the service element.