Document Title	Explanation of Adaptive Platform Software Architecture
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	982

Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R25-11

Document Change History			
Date	**Release**	**Changed by**	**Description**
2025-11-27	R25-11	AUTOSAR Release Management	• Added the Functional Clusters Remote Persistency and Safe Hardware Acceleration. • Added Suspend-to-RAM functionality to State Management. • Reworked use cases and scenarios for Platform Health Management.
2024-11-27	R24-11	AUTOSAR Release Management	• Added the Functional Cluster Automotive API Gateway. • Extended the Use Case View and Runtime View with use cases and scenarios. • Added information about supported bus types and protocols. • Described design principles and patterns used in the Adaptive Platform.

▽

△

2023-11-23	R23-11	AUTOSAR Release Management	• Extended the Use Case View and Runtime View with use cases and scenarios for the Functional Clusters Time Synchronization and Platform Health Management. • Removed Functional Cluster Identity and Access Management. • Added Functional Clusters Vehicle Update and Configuration Management and Raw Data Stream.
2022-11-24	R22-11	AUTOSAR Release Management	• Added an initial version of the Use Case View that shows the main use cases for the Functional Clusters of the AUTOSAR Adaptive Platform. • Reworked the Runtime View to show a proposed internal realization of the use cases in the Use Case View by means of interaction diagrams. • Added functional cluster Firewall
2021-11-25	R21-11	AUTOSAR Release Management	• Applied a more fine-grained description schema for functional clusters and interfaces in the Building Block View. • Removed functional cluster RESTful Communication • Added functional cluster Adaptive Intrusion Detection System Manager • Added section for clarification of diagnostic deployment options
2020-11-30	R20-11	AUTOSAR Release Management	• Initial release

Table of Contents

1 Introduction

This explanatory document provides detailed technical description of the software architecture of the AUTOSAR Adaptive Platform standard with the main focus on the architecture model.

1.1 Objectives

This document is an architecture description of the AUTOSAR Adaptive Platform in accordance to [1, ISO/IEC 42010] and has the following main objectives:

- Identify the **stakeholders** of the AUTOSAR Adaptive Platform and their **concerns**.
- Identify the **system scope** and provide **overview information** of the AUTOSAR Adaptive Platform.
- Provide definitions for all used **architecture viewpoints** and a **mapping of all stakeholder concerns to those viewpoints**.
- Provide an **architecture view** and its **architecture models** for each architecture viewpoint used in this architecture description.
- Provide **correspondence rules** and **correspondences** among the contents of this architecture description.
- Provide an **architecture rationale** (explanation, justification, reasoning for decisions made) on a high level. A more in-depth documentation of decisions is provided in [2, EXP_SWArchitecturalDecisions].
- Provide a **record of known inconsistencies and gaps** among the architecture description.

There is some potential for ambiguity about the term "architecture". Association with this term is quite different e.g., for a mass production project in contrast with Adaptive Platform standardization. For system development of an automotive embedded computer the software architecture usually defines the details of the structural and the behavioral architecture views down to module level. In contrast the architecture of AUTOSAR Adaptive Platform lacks such details deliberately to provide more degrees of freedom for stack vendors in their solution design.

Beyond the specification of APIs the term "architecture" for Adaptive Platform refers to guidelines how to apply the standard to concrete development projects.

This document describes the original architectural design of the AUTOSAR Adaptive Platform including details how the building blocks should interact with each other. It is an example how an implementation of the standard should work internally. However, a stack vendor is free to choose another design as long as it complies with the binding AUTOSAR Adaptive Platform standard.

1.2 Scope

This explanatory document applies to the AUTOSAR Adaptive Platform. It is recommended to get an overview of the AUTOSAR Adaptive Platform for all members of the working groups, stack vendors, and application developers.

1.3 Document Structure

This document is organized as follows. Section 4 provides an overview of the main requirements for the AUTOSAR Adaptive Platform, the top quality goals of its architecture, and a list of stakeholders that are affected by it. Section 5 lists requirements that constrain design and implementation decisions or decisions about the development process.

Section 6 is the base for discovering trade-offs and sensitivity points in the architecture of the AUTOSAR Adaptive Platform by introducing a quality attribute tree followed by the most important quality scenarios. The system context in which the AUTOSAR Adaptive Platform is intended to be used is outlined in section 7. Section 8 summarizes the fundamental decisions and solution strategies, that shape the architecture of the AUTOSAR Adaptive Platform such as technology decisions or architectural patterns to be used.

Sections 9 to 12 explain the software architecture from different view points. First, section 9 explains the decomposition of the AUTOSAR Adaptive Platform into Functional Clusters and their interdependencies. Then, section 11 demonstrates how the main use cases are realized using the Functional Clusters in the AUTOSAR Adaptive Platform. Section 12 shows different scenarios how applications based on the AUTOSAR Adaptive Platform may be deployed.

Section 13 provides an overview of concepts and patterns used by the AUTOSAR Adaptive Platform.

2 Definition of Terms and Acronyms

This chapter lists abbreviations and terms that are specific to this document. A list of general terms for AUTOSAR is provided in the [3, glossary].

2.1 Acronyms and Abbreviations

Abbreviation	Description
SecOC	AUTOSAR Secure Onboard Communication

Table 2.1: Definition of Acronyms and Abbreviations

2.2 Definition of Terms

Term	Description
Thread	The smallest sequence of instructions the can be managed independently by a scheduler. Multiple `Thread`s can be executed concurrently within one `Process` sharing resources such as memory.
Watchdog	An external component that supervises execution of the AUTOSAR Adaptive Platform. See Section 7.2.3 for details.

Table 2.2: Definition of Terms

3 Related Documentation

This document provides a high-level overview of the AUTOSAR Adaptive Platform architecture. It is closely related to general requirements for AUTOSAR Adaptive Platform specified in [4, RS_General], and the architectural decisions documented in [2, EXP_SWArchitecturalDecisions].

The individual building blocks of the architecture (`Functional Cluster`s) are specified in separate documents. Each `Functional Cluster` defines one or more requirements specification(s) (RS document), one or more software specification(s) (SWS document) and one or more explanatory document(s) (EXP document). Please refer to these documents for any details on the AUTOSAR Adaptive Platform standard.

3.1 References

[1] ISO 42010:2011 – Systems and software engineering – Architecture description
https://www.iso.org

[2] Explanation of Adaptive and Classic Platform Software Architectural Decisions
AUTOSAR_FO_EXP_SWArchitecturalDecisions

[3] Glossary
AUTOSAR_FO_TR_Glossary

[4] General Requirements specific to Adaptive Platform
AUTOSAR_AP_RS_General

[5] ATAM[SM]: Method for Architecture Evaluation
https://resources.sei.cmu.edu/asset_files/TechnicalReport/2000_005_001_13706.pdf

[6] Agile Software Development: Principles, Patterns, and Practices

[7] Guide to the Software Engineering Body of Knowledge, Version 3.0
www.swebok.org

[8] Specification of Communication Management
AUTOSAR_AP_SWS_CommunicationManagement

[9] Specification of Manifest
AUTOSAR_AP_TPS_ManifestSpecification

[10] Explanation of Adaptive Platform Design
AUTOSAR_AP_EXP_PlatformDesign

[11] Explanation of MACsec and MKA Protocols implementation and configuration guidelines
AUTOSAR_AP_EXP_MACsec

[12] Time Synchronization over Ethernet Protocol Specification

AUTOSAR_FO_PRS_TimeSyncOverEthernetProtocol

[13] SOME/IP Protocol Specification
AUTOSAR_FO_PRS_SOMEIPProtocol

[14] SOME/IP Service Discovery Protocol Specification
AUTOSAR_FO_PRS_SOMEIPServiceDiscoveryProtocol

[15] Specification of the AUTOSAR Network Management Protocol
AUTOSAR_FO_PRS_NetworkManagementProtocol

[16] Log and Trace Protocol Specification
AUTOSAR_FO_PRS_LogAndTraceProtocol

[17] Specification of Secure Onboard Communication Protocol
AUTOSAR_FO_PRS_SecOCProtocol

[18] Specification of Intrusion Detection System Protocol
AUTOSAR_FO_PRS_IntrusionDetectionSystem

[19] E2E Protocol Specification
AUTOSAR_FO_PRS_E2EProtocol

[20] Explanation of Service-Oriented Vehicle Diagnostics
AUTOSAR_AP_EXP_SOVD

[21] Requirements on Data Distribution Service
AUTOSAR_FO_RS_DataDistributionService

[22] Specification of Diagnostics
AUTOSAR_AP_SWS_Diagnostics

[23] Specification of Automotive API Gateway
AUTOSAR_AP_SWS_AutomotiveAPIGateway

[24] Specification of Raw Data Stream
AUTOSAR_AP_SWS_RawDataStream

[25] Specification of Sensor Interfaces
AUTOSAR_AP_SWS_SensorInterfaces

4 Overview and Goals

In conventional automotive systems ECUs are used to replace or augment electro-mechanical systems. Those resource constrained, deeply-embedded ECUs typically perform basic control functions by creating electrical output signals (e.g. for actors) based on input signals (e.g. from sensors) and information from other ECUs connected to the vehicle network. Much of the control software is specifically designed and implemented for the target vehicle and does not change significantly during vehicle lifetime. The AUTOSAR Classic Platform standard addresses the needs of these deeply-embedded systems.

Recent and future vehicle functions, such as highly automated driving, will introduce complex and computing resource demanding software that shall fulfill strict safety, integrity and security requirements. Such software performs for example, environment perception and behavior planning, and interacts with external backend and infrastructure systems. The software in the vehicle regularly needs to be updated during the life-cycle of the vehicle, due to evolving external systems, improved or added functionality, or security problems. The AUTOSAR Classic Platform standard cannot fulfill the needs of such systems. Therefore, AUTOSAR specifies a second software platform, the AUTOSAR Adaptive Platform. It provides high-performance computing and communication mechanisms as well as a flexible software configuration, for example, to support software update over-the-air. Features that are specifically defined for the AUTOSAR Classic Platform, such as access to electrical signals and automotive specific bus systems, can be integrated into the AUTOSAR Adaptive Platform but is not in the focus of standardization.

4.1 Requirements Overview

This section provides an overview of the basic requirements for the AUTOSAR Adaptive Platform that impact its architecture. The corresponding requirement identifiers are provided in square brackets. Please refer to [4, RS_General] for any details, rationale or intended use-cases of these requirements.

Support of state-of-the-art Technology

The AUTOSAR Adaptive Platform aims to support resource-intensive (memory, cpu) applications on state-of-the-art hardware. Therefore, the AUTOSAR Adaptive Platform shall support high performance computing platforms [RS_Main_00002] as well as virtualized environments [RS_Main_00511]. The AUTOSAR Adaptive Platform shall be able to run multiple applications in parallel [RS_Main_00049], each with concurrent application internal control flows [RS_Main_00050].

Software Update and Configuration

The AUTOSAR Adaptive Platform shall support a flexible (configuration) data and software update. Hereby, AUTOSAR Adaptive Platform shall support up- and download of such update packages [RS_Main_00650] and change of communication and application software at runtime [RS_Main_00503].

AUTOSAR shall provide a unified way to describe software systems deployed to Adaptive and / or Classic platforms [RS_Main_00161]. That kind of description shall also support the deployment and reallocation of AUTOSAR Application Software [RS_Main_00150], and shall provide means to describe interfaces of the entire system [RS_Main_00160].

Security

The AUTOSAR Adaptive Platform shall support the development of secure systems [RS_Main_00514] with secure access to ECU data and services [RS_Main_00170], and secure onboard communication [RS_Main_00510].

Safety

The AUTOSAR Adaptive Platform shall support the development of safety related systems [RS_Main_00010] that are reliable [RS_Main_00011] and highly available [RS_Main_00012]. The AUTOSAR Adaptive Platform specifications shall be analyzable and support methods to demonstrate the achievement of safety related properties accordingly [RS_Main_00350].

Reuse and Interoperability

The AUTOSAR Adaptive Platform shall support standardized interoperability with non-AUTOSAR software [RS_Main_00190] as well as (source code) portability for AUTOSAR Adaptive Applications across different implementations of the platform [RS_AP_00111]. Hereby, the AUTOSAR Adaptive Platform shall provide means to describe a component model for application software [RS_Main_00080], and support bindings for different programming languages [RS_Main_00513].

Communication

The AUTOSAR Adaptive Platform shall support standardized automotive communication protocols [RS_Main_00280] for intra ECU communication [RS_Main_01001] with different network topologies [RS_Main_00230].

Diagnostics

The AUTOSAR Adaptive Platform shall provide diagnostics means during runtime for production and services purposes [RS_Main_00260].

4.2 Quality Goals

This section will list the top quality goals for the architecture whose fulfillment is of highest importance to the major stakeholders in a future version of this document. Please refer to the currently un-prioritized list of Quality Attributes in Section 6.1.

4.3 Stakeholders

This section lists the stakeholders of the AUTOSAR Adaptive Platform architecture and their main expectations.

Role	Expectation
Project Leader	Overview of technical risks and technical debt in the AUTOSAR Adaptive Platform.
Working Group Architecture	Concise yet thorough documentation of the goals and driving forces of the AUTOSAR Adaptive Platform. Documentation of the original architectural design of the AUTOSAR Adaptive Platform standard. Documentation of identified technical risks and technical debt in the AUTOSAR Adaptive Platform.
Working Group	Consolidated overview of the AUTOSAR Adaptive Platform architecture. Realization of use-cases that span multiple `Functional Cluster`s. Usage of interfaces within the AUTOSAR Adaptive Platform. Guidelines and patterns for `Functional Cluster` and interface design.
Stack Developer	Consolidated overview of the original architectural design of the AUTOSAR Adaptive Platform. Realization of use-cases that span multiple `Functional Cluster`s. Usage of interfaces within the AUTOSAR Adaptive Platform.
Application Developer	Overview of the building blocks of the AUTOSAR Adaptive Platform and their purpose and provided functionality. Explanation of the concepts used in the AUTOSAR Adaptive Platform.
Integrator	Overview and explanation of the original architectural design of the AUTOSAR Adaptive Platform. Overview of extension points of the AUTOSAR Adaptive Platform.

Table 4.1: Stakeholder table with roles and expectations

5 Architecture Constraints

AUTOSAR is a worldwide development partnership of vehicle manufacturers, suppliers, service providers and companies from the automotive electronics, semiconductor and software industry. AUTOSAR standardizes the AUTOSAR Adaptive Platform automotive middleware. The AUTOSAR Adaptive Platform is not a concrete implementation. The AUTOSAR Adaptive Platform standard leaves a certain degree of freedom to implementers of the standard, as most standards do. On the one hand, more freedom enables competition among the different implementations and a broader choice of properties for users of the AUTOSAR Adaptive Platform. On the other hand, a more strict standardization makes the different implementations compatible and exchangeable (within the standardized scope). Naturally, those attributes are in conflict. It is usually a choice of the standardization organization to evaluate the attributes and define the desired level of strictness.

The AUTOSAR Classic Platform is rather strict in that sense by specifying a detailed layered software architecture imposing many constraints on its implementations. The AUTOSAR Adaptive Platform launched with a less strict approach. That less strict approach puts constraints on the AUTOSAR Adaptive Platform architecture as outlined below.

5.1 Internal Interfaces

An important architectural constraint is that only interfaces that are intended to be used by applications or interfaces that are used to extend the functionality of the AUTOSAR Adaptive Platform shall be standardized. Internal interfaces between the building blocks of the AUTOSAR Adaptive Platform shall not be standardized. This approach leaves a lot of freedom to design and optimize the internals of an AUTOSAR Adaptive Platform stack. However, it also imposes constraints on how the AUTOSAR Adaptive Platform architecture can be defined and described in this document. Also, this means that it might not be possible to use different functional clusters from different AUTOSAR Adaptive Platform stack vendors.

First, the existence of internal interfaces and their usage by other building blocks is in most cases a recommendation and reflects the original design approach of the authors of the standard. The same applies to any interactions described in this document that make use of such internal interfaces.

Second, some quality attributes may be hard to ensure in general by the architecture of the standard. Additional measures like security or safety considerations lack well-defined inputs such as data flows or specifications of interdependencies. Consequently, a more thorough design phase is required when an AUTOSAR Adaptive Platform stack is implemented.

5.2 Distributed Work

Standardization of the AUTOSAR Adaptive Platform is a worldwide distributed effort. The individual building blocks are specified by dedicated working groups in separate documents to be able to scale in that distributed setup. This impacts the way the AUTOSAR Adaptive Platform architecture is described in this document.

First, this document shows interfaces on an architectural level only. This document does not specify details of interfaces such as individual operations. This keeps redundancies and thus dependencies between this document and the documents actually specifying the individual building blocks manageable. Another consequence is that the interactions shown in this document are not based on actual operations specified in the interfaces but rather on an architectural level as well.

Second, this document aims to provide guidance for the working groups in specifying the individual building blocks by defining patterns and concepts to solve common problems. This guidance should help to build a uniform and consistent standard from ground up.

6 Quality Requirements

Quality requirements define the expectations of AUTOSAR Adaptive Platform stakeholders for the quality and the attributes of the AUTOSAR Adaptive Platform standard that indicate whether the quality factors are satisfied or not. Section 6.1 starts by listing the quality attributes that, in the end, are used to assess whether the AUTOSAR Adaptive Platform and its software architecture satisfies the expected quality factors or not. Section 6.2 then provides quality scenarios that operationalize quality attributes and turn them into measurable quantities by describing the reaction of the system to a stimulus in a certain situation.

6.1 Quality Attributes

The AUTOSAR Adaptive Platform has many stakeholders with different concerns. Thus, this document uses the following three quality attribute categories that correspond to the three main stakeholder groups in order to make the requirements and their impact on the architecture more comprehensible:

- **AUTOSAR Adaptive Platform Standard:** Quality requirements for the AUTOSAR standard itself. These requirements may directly affect the architecture of the AUTOSAR Adaptive Platform.
- **AUTOSAR Adaptive Platform Stack:** Quality requirements for an implementation of the AUTOSAR standard as an AUTOSAR stack. These requirements may indirectly affect the architecture of the AUTOSAR Adaptive Platform.
- **AUTOSAR Adaptive Application:** Quality requirements for an application based on an AUTOSAR stack. These requirements may transitively affect the architecture of the AUTOSAR Adaptive Platform.

The quality attributes are organized according to the Architecture Tradeoff Analysis Method (ATAM) [5] as a tree, one for each of the quality attribute categories. The leafs of those trees are the individual quality attributes.

6.1.1 AUTOSAR Adaptive Platform Standard

- Functional suitability
  - The software architecture shall reflect the project objectives (POs) and be the consistent source for all specifications (here: completeness with respect to the PO; see also usability below).
  - The standard shall not contain elements that are not traceable to POs, change requests (CRs), or concepts.
  - The standard shall contain at least one element derived from each PO, CR, or concept.

- Performance efficiency
  - The specification shall allow for a run-time efficient implementation. Run-time efficiency refers to all resource consumption, CPU, RAM, ROM.
- Compatibility
  - The standard shall retain older versions of its elements in the face of change.
  - The standard shall be interoperable with pre-existing standards, especially the AUTOSAR Classic Platform. Pre-existing standards means network protocols and the like.
  - The standard shall adopt new versions of pre-existing standards only after an impact analysis.
- Usability
  - The use of the standard shall be as easy as possible for suppliers and application developers. Easy means: not much material and resources required.
  - The holistic approach shall not be broken (avoid different approaches in one standard).
  - The standard shall contain application sample code for all its elements.
  - The standard shall contain documentation of the use cases for its elements.
  - The standard shall document the semantics of its elements.
  - The standard shall document its decisions, consequences, and implementation restrictions (both for stack & apps) including their rationale.
  - The standards elements shall be easy to use and hard to misuse.
  - The standard shall stick to pre-existing standards, as far as no functional requirements are compromised.
  - The standard shall be as stable as possible.
  - AUTOSAR standards shall not change disruptive but rather evolve evolutionary (for example, backward-compatibility can be a help).
  - The software architecture shall reflect the PO and be the consistent source for all specifications (here: consistency; see also functional suitability above).
- Reliability
  - The standard shall classify its elements with respect to safety relevance (that is, functional clusters shall be marked if they participate in safety critical operations of the platform).
  - The standard shall specify control flow restrictions between its elements in order to achieve freedom from interference.

- The standard shall contain use case driven argumentation for safety scenarios that can be used to build a safety case. (This should help the stack implementers in getting a certification, if they follow the standard.)

- Security
  - The standard shall specify data flow restrictions between its elements, and between applications.
  - The standard shall classify its elements with respect to security sensitivity (that is, functional clusters shall be marked if they handle sensitive data.)
  - The standard shall contain use case driven argumentation for security scenarios that can be used to build a security case. (This should help the stack implementers in getting a certification, if they follow the standard.)

- Maintainability
  - It shall be possible in an efficient way to maintain AUTOSAR Adaptive Platform without preventing the introduction of new technologies (efficient in terms of effort on the modification of the standard).
  - The impact set of a change shall be available.
  - The standard shall be structured in a way that minimizes change impact.
  - It shall be possible to drop/deprecate elements of the standard.
  - It shall be easy to add new features/needs without breaking the maintainability or the need to redesign the software architecture. Easy means quick, with low effort, local changes only and no heavy side effects.
  - The maturity of parts of the standard shall be visible.
  - The process shall enforce an architectural impact analysis in a very early stage of the change process.
  - The process shall enforce minimizing changes, that is not adding similar functionality multiple times.

- Portability
  - Applications shall be portable between different stack implementations and different machines.
  - It shall be possible to scale the software architecture to the given project needs.

6.1.2 AUTOSAR Adaptive Platform Stack

- Compatibility

  - An AUTOSAR Adaptive Platform stack implementation shall be capable to offer multiple versions of the same service.
  - An instance of an AUTOSAR Adaptive Platform stack implementation shall be able to co-exist with other instances on different machines, within the same vehicle.
- Usability
  - An AUTOSAR Adaptive Platform stack implementation shall explicitly document restrictions on the application development that go beyond the standard.
- Maintainability
  - An AUTOSAR Adaptive Platform stack implementation shall be traceable to the contents of the standard.
  - An AUTOSAR Adaptive Platform stack implementation shall support multiple versions of the same service.
- Portability
  - An AUTOSAR Adaptive Platform stack shall be portable to a different custom hardware.
  - An AUTOSAR Adaptive Platform stack shall provide mechanisms to replace parts.

6.1.3 AUTOSAR Adaptive Application

- Usability
  - *No Goal:* An application developer shall be able to supply custom implementation for pre-defined platform functionality.
- Maintainability
  - An application shall explicitly state which parts of the standard it uses.
- Portability
  - An application entirely based on AUTOSAR Adaptive Platform (i.e. without custom extensions) shall be portable to another AUTOSAR Adaptive Platform stack of the same version without modifications to the application source code itself (source code compatibility).

6.2 Quality Scenarios

There are currently no quality scenarios defined for the AUTOSAR Adaptive Platform.

7 System Scope and Context

This chapter provides an overview of the AUTOSAR Adaptive Platform system context by separating the AUTOSAR Adaptive Platform and its communication partners (e.g., external systems). Considering Figure 7.1, there are three categories of communication partners for the AUTOSAR Adaptive Platform.

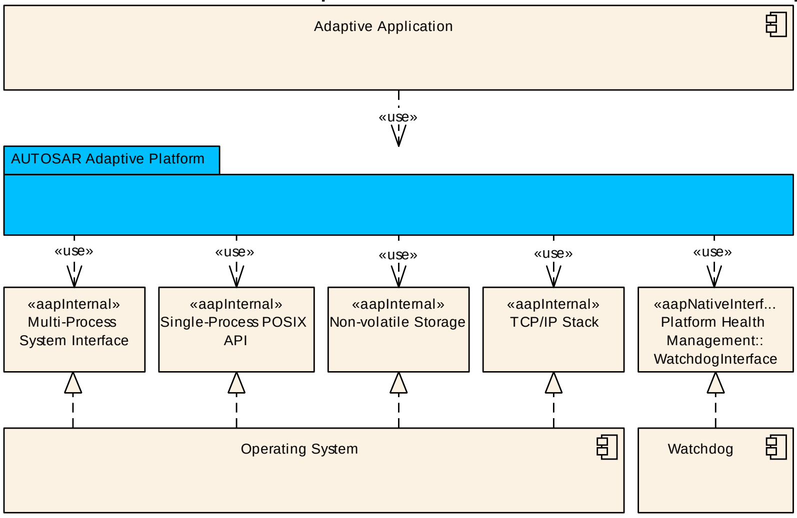

Figure 7.1: Overview of AUTOSAR Adaptive Platform and its context

The AUTOSAR Adaptive Platform is conceptually a middleware. AUTOSAR Adaptive Platform provides services to `Adaptive Application`s beyond those available from the underlying operating system, drivers, and extensions (cf. Section 7.2). Section 7.3 describes the third category that are external systems communicating with (an `Adaptive Application` via) AUTOSAR Adaptive Platform.

7.1 Adaptive Application

`Adaptive Application`s are built on the functionality provided by the AUTOSAR Adaptive Platform. They directly use the various interfaces provided by the individual building blocks of AUTOSAR Adaptive Platform described in more detail in chapter 9.

7.2 Dependencies

7.2.1 Crypto Provider

`Crypto Provider` is a component that provides implementations of cryptographic routines and hash functions to the AUTOSAR Adaptive Platform.

7.2.2 Operating System

The Operating System is the main component that AUTOSAR Adaptive Platform uses to provide its services. The Operating System controls processes and threads, and provides inter-process communication facilities. The Operating System also provides access to network interfaces, protocols like TCP/IP, and access to non-volatile storage.

7.2.3 Watchdog

The Watchdog is a component to control the hardware watchdog of the machine an AUTOSAR Adaptive Platform runs on.

7.3 External Systems

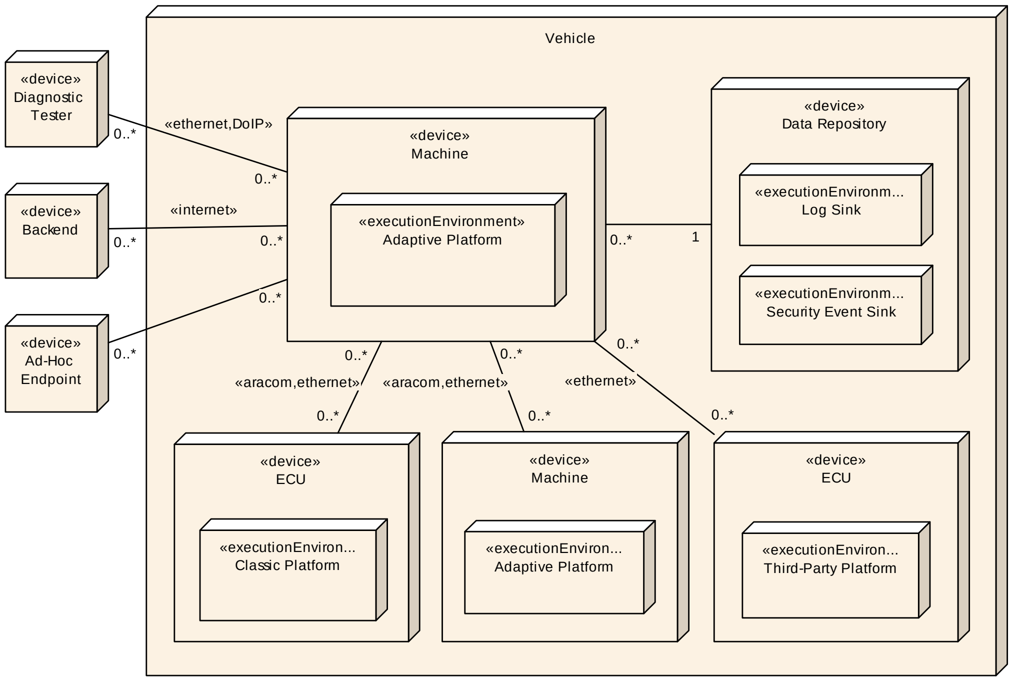

Figure 7.2: Overview of the AUTOSAR Adaptive Platform and external systems

The AUTOSAR Adaptive Platform supports applications that are operated in heterogeneous environments. This section lists the external systems that AUTOSAR Adaptive Platform is intended to interface.

7.3.1 Diagnostic Tester

A tester is considered a physical or virtual device, which is located external to the Vehicle Network. It may request diagnostic services to be handled by the AUTOSAR Adaptive Platform, for instance a UDS request that is processed by `Diagnostic Management`.

7.3.2 Backend

Network node or network gateway that exchanges data with the AUTOSAR Adaptive Platform usually via TCP/IP protocol suite and is reachable via a public or a private IP address. Examples include an OEM cloud service node, a node in a private, e.g. home network, or a node connected to the public internet.

7.3.3 AdHoc Endpoint

A node which communicates via wireless links (single hop or multi-hop connection) with an AUTOSAR Adaptive Platform instance. Therefore, availability of services is constrained to regional proximity (distance usually from few meters up to a few hundred meters). Examples include a stationary roadside infrastructure or a road side user (e.g. device attached to pedestrian).

7.3.4 AUTOSAR Adaptive Platform

An AUTOSAR Adaptive Platform instance may need to exchange information with another `AUTOSAR Adaptive Platform` instance. Examples include the communication between `Vehicle Update and Configuration Management` and one or more `Update and Configuration Management` instances to distribute and install `Software Packages` in the vehicle.

7.3.5 AUTOSAR Classic Platform

An AUTOSAR Adaptive Platform instance may also need to exchange information with an `AUTOSAR Classic Platform` instance. Examples include sending/receiving controller signals (e.g. transmitted via CAN bus) that are translated to ara::com services connected via Ethernet network by an `AUTOSAR Classic Platform` instance.

7.3.6 Third-party Platform

An objective of AUTOSAR is to enable service oriented communication between an AUTOSAR Adaptive Platform instance and a 3rd-party Platform instance. Even though a gateway function to map protocol data on both endpoints has not yet been specified, this possibility is considered for the modelling of external actors.

7.3.7 Data Repository

A data repository may host/store data associated with events generated within the AUTOSAR Adaptive Platform, such as log events by `Log and Trace` or security events by `Intrusion Detection System Manager` functional clusters. Such a repository is located external to the AUTOSAR Adaptive Platform.

8 Solution Strategy

The AUTOSAR Adaptive Platform is a standard for an automotive middleware. It is not a concrete implementation. The AUTOSAR Adaptive Platform standard leaves a certain degree of freedom to its implementers by defining requirements and software specifications that need to be fulfilled without specifying how.

8.1 Architectural Approach

To support the complex applications, while allowing maximum flexibility and scalability in processing distribution and compute resource allocations, AUTOSAR Adaptive Platform follows the concept of a service-oriented architecture (SOA). In a service-oriented architecture a system consists of a set of services, in which one may use another in turn, and applications that use one or more of the services depending on its needs. Often service-oriented architectures exhibit system-of-system characteristics, which AUTOSAR Adaptive Platform also has. A service, for instance, may reside on a local ECU that an application also runs, or it can be on a remote ECU, which is also running another instance of AP. The application code is the same in both cases - the communication infrastructure will take care of the difference providing transparent communication. Another way to look at this architecture is that of distributed computing, communicating over some form of message passing. At large, all these represent the same concept. This message passing, communication-based architecture can also benefit from the rise of fast and high-bandwidth communication such as Ethernet.

8.2 Decomposition Strategy

The building blocks of the AUTOSAR Adaptive Platform architecture are refined step-by-step in this document according to the model depicted in figure 8.1. The top-level categories are chosen to give an overview from a users perspective what kind of functionality the AUTOSAR Adaptive Platform provides. A category contains one or more `Functional Cluster`s. The `Functional Cluster`s of the AUTOSAR Adaptive Platform are defined to group a specific coherent technical functionality. `Functional Cluster`s themselves specify a set of interfaces and components to provide and realize that technical functionality. The building block view also contains information of the `Functional Cluster`s interdependencies based on interfaces from other `Functional Cluster`s they use. However, note that these interdependencies are recommendations rather than strict specifications because they would constrain implementations.

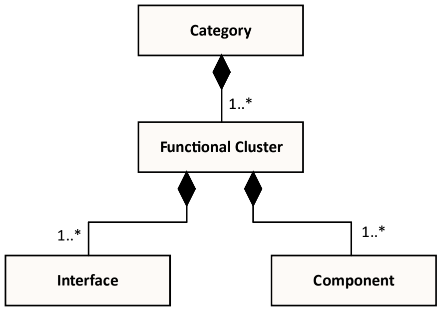

Figure 8.1: Type model of building blocks

8.3 UML Profile

The UML diagrams presented in this document use a UML profile to provide a more precise semantics of the elements and relationships. Table 8.1 provides an overview of the stereotypes in that profile and their semantics. The names of all stereotypes that are specific to the AUTOSAR architecture are prefixed with *aap* (short for AUTOSAR Architecture Profile) to make them easily distinguishable from standard UML stereotypes and keywords.

Stereotype	***UML Meta-Class***	***Semantics***
`aapInternal`	Interface	Internal interfaces shall be used only by Functional Clusters within the platform. Internal interfaces shall be realized by components that are part of the respective stack implementation (i.e., another Functional Cluster, additional middleware, drivers, or the operating system).
`aapPlatformExtension`	Interface	Platform extension interfaces shall be used only by Functional Clusters within the platform. Platform extension interfaces shall be realized either by third-party components (including application-level components) or components are part of the respective stack implementation (i.e., another Functional Cluster, additional middleware, drivers, or the operating system).
`aapAPI`	Interface	An interface of the public API of the platform. Such interfaces may be used by Adaptive Applications and other Functional Clusters within the platform.

▽

△

Stereotype	***UML Meta-Class***	***Semantics***
aapNativeInterface	Interface	An interface defined in the respective programming language of the stack implementation.
aapPortInterface	Interface	An aapPortInterface relates to an element in the Manifest using the InstanceSpecifier pattern. Such interfaces are either defined in the respective programming language of the stack implementation or they are generated in that language by the stack tooling. The property Generated specifies if they are generated.
aapAraComServiceInterface	Interface	An ara::com service interface defined and configured in the Manifest. The tooling of the stack implementation will generate stub and skeleton implementations of these interfaces in the respective programming language.
aapFunctionalCluster	Component	A functional cluster of the AUTOSAR Adaptive Platform.
aapServiceMethod	Operation	A method specified as part of an ara::com service interface.
aapServiceField	Property	A field specified as part of an ara::com service interface.
aapServiceEvent	Property	An event specified as part of an ara::com service interface.
aapCallbackMethod	Operation	A method that acts as a callback.
aapProvidedPort	Usage, Realization	Denotes that the underlying relationship is configured in the Manifest using a provided port (PPortPrototype).
aapRequiredPort	Usage, Realization	Denotes that the underlying relationship is configured in the Manifest using a required port (RPortPrototype).
aapInformative	Message	Denotes that the message only has an informative character. A concrete stack implementation may use a different design (name, type of message, order of messages).

Table 8.1: Overview of Stereotypes

8.4 Technology

8.4.1 Implementation Language

C++ is the programming language of choice for the AUTOSAR Adaptive Platform and Adaptive Applications. C++ was chosen due to its safer programming model (compared to C) and availability of certified compilers that produce highly optimized machine code. Such properties are especially important for safety- and performance-critical, real-time applications (such as typical `Adaptive Application`s) where C++ has become more and more popular in the software industry and academics.

8.4.2 Parallel Processing

Although the design for AUTOSAR Adaptive Platform as a service-oriented architecture inherently leverages parallel processing, the advancement of (heterogeneous) many-core processors offers additional opportunities. The AUTOSAR Adaptive Platform is designed to scale its functionality and performance as (heterogeneous) many-core technologies advance. The design of the AUTOSAR Adaptive Platform includes APIs to leverage heterogeneous high-performance computing based on hardware acceleration. Hardware and platform interface specifications are one part of that equation. However, advancements in operating system and hypervisor technologies as well as development tools (for example automatic parallelization) are also crucial and are to be fulfilled by AUTOSAR Adaptive Platform providers, the software industry, and academics.

8.5 Design Principles

The architecture of the AUTOSAR Adaptive Platform is based on several design principles that are outlined below.

8.5.1 Leveraging existing standards

AUTOSAR Adaptive Platform aims to leverage existing standards and specifications wherever possible. For example, AUTOSAR Adaptive Platform ist built on the existing and open C++ standard (cf. Section 8.4.1) to facilitate a faster development of the AUTOSAR Adaptive Platform itself and benefiting from the eco-systems of such standards. It is, therefore, a critical focus in developing the AUTOSAR Adaptive Platform specification not to casually introduce a new replacement functionality that an existing standard already offers. For instance, no new interfaces are casually introduced just because an existing standard provides the functionality required but the interface is superficially hard to understand.

8.5.2 SOLID principles

The SOLID principles [6] are a central part of the design principles of AUTOSAR. While these five principles are all valid, only the Single-responsibility Principle, the Interface Segregation Principle and the Dependency Inversion Principle are relevant on the abstraction level of this document. Therefore, they are elaborated in the following.

8.5.2.1 Single-responsibility Principle

The single-responsibility principle (SRP, SWEBOK3) [6] states that a component or class should be responsible for a single part of the overall functionality provided by the software. That responsibility should be encapsulated by the component or class. The services provided by the component or class (via its interface(s)) should be closely aligned with its responsibility.

The single-responsibility principle minimizes the reasons (i.e. a change to the single responsibility) that require a change to its interface. Thus, it minimizes impact on clients of such an interface and leads to a more maintainable architecture (or code).

8.5.2.2 Interface Segregation Principle

The interface segregation principle (ISP) [6], [7] states that clients should not be forced to depend on methods that they don't use. As a consequence of the interface segregation principle, interfaces should be split up to reflect different roles of clients.

Similar to the single-responsibility principle, the segregation of interfaces reduce the impact of a change to an interface to the clients and suppliers of an segregated interface.

8.5.2.3 Dependency Inversion Principle

The dependency inversion principle (DIP) [6], [7] states that high-level building blocks should not depend on low-level building blocks. Both should depend on abstractions (e.g. interfaces). Furthermore, the dependency inversion principle states that abstractions (e.g. interfaces) should not depend on details. Details (e.g. a concrete implementation) should depend on abstractions.

The dependency inversion principle results in a decoupling of the implementations of building blocks. This is important to scale implementation efforts (cf. Section 5.2) and to perform proper integration tests.

8.5.3 Acyclic Dependencies Principle

The acyclic dependencies principle (ADP) [6], [7] states that dependencies between building blocks should form a directed acyclic graph.

The acyclic dependencies principle helps to identify participating building blocks and reason about error propagation and freedom from interference. In general, it also reduces the extend of building blocks to consider during activities such as test, build and deployment.

8.5.4 Versioning of the Service Interface API

Communication API of the AUTOSAR Adaptive Platform is split into several abstraction layers, each of which can use its own versioning schema to define compatibility rules on that layer. Versioning of the Service Interface API should not be mixed with versioning of the transport layer mechanism (e.g. SOME/IP), as the transport layer may define its own compatibility rules based on the features used, and its version is separated from that one defined for the Service Interface API, which belongs to the application layer. But when the major version of the Service Interface is increased, the transport layer version needs to be updated as well to indicate the influence of Service Interface level changes.

8.5.5 Platform Extensions

The AUTOSAR Adaptive Platform supports extensions of its behavior by an OEM using standardized extension interfaces, so called `Platform Extension Interface`s. The use of `Platform Extension Interface`s is limited to cases in which it is easier to provide an implementation of a behavior rather than configuring a generic behavior via the `Manifest`. The implementation of a `Platform Extension Interface` is always provided by an OEM. Similar interfaces (e.g., `Callback`s) that extend the behavior of the AUTOSAR Adaptive Platform but are implemented by an Adaptive Application are not considered to be `Platform Extension Interface`s.

The Adaptive Platform differentiates between three types of platform extension interfaces, `Callout`, `Callback`, and `Plugin`. These interface types are described in the subsequent sections.

8.5.5.1 Callout

The `Callout` pattern relies on the extension providing an implementation of one or more functions or classes with a standardized name and signature. No implementation is provided for `Callout`s by the stack vendor but they may be used by the Adaptive Platform. The `Callout` becomes a part of the stack (e.g., an executable of a daemon process). If the functions or classes are not implemented, the application or Adaptive Platform will not compile or fail to start due to missing dependencies.

8.5.5.1.1 Purpose

A `Callout` is intended for adapting a specific mechanism in the Adaptive Platform to OEM needs.

8.5.5.1.2 Applicability

The `Callout` pattern is applicable if exactly one such extension needs to be provided for a given project and no runtime variability is required.

8.5.5.1.3 Class Structure

There is no special structure necessary to define a `Callout`.

8.5.5.1.4 Behavior

There is no special behavior necessary to define a `Callout`. The registration of the `Platform Extension Interface` is performed statically by the linker.

8.5.5.1.5 Consequences

It is mandatory to provide an implementation of a `Callout`. This pattern adds no runtime overhead and no registration at runtime is necessary.

8.5.5.1.6 Examples

- `Freshness Value Manager` in `Communication Management`

8.5.5.2 Callback

The `Callback` pattern relies on a dynamic registration of functionality to be invoked asynchronously by the Adaptive Platform at a later point in time.

8.5.5.2.1 Purpose

The main purpose of `Callback`s is to signal events e.g., asynchronous results. If the `Callback` is registered from higher level functionality (i.e., an Adaptive Application), the `Callback` is however not considered to be a `Platform Extension Interface`.

8.5.5.2.2 Applicability

This pattern is applicable if the behavior extension can be realized in a single function. The pattern is also applicable if the extension should be changed during runtime and/or there should be more than one `Callback` registered. Parameters in a `Callback` may be used to transfer data from the AUTOSAR Adaptive Platform to the `Callback` only. Return values may be used to transfer data from the `Callback` back to the AUTOSAR Adaptive Platform.

8.5.5.2.3 Class Structure

There are two supported patterns to realize callbacks in the Adaptive Platform:

1. By implementing a std::function in the Adaptive Application and registering (and unregistering) it at runtime with the Adaptive Platform. Registration may be optional (e.g., by a setter method) or mandatory (e.g., as parameter in the constructor).
2. By implementing a skeleton and providing an implementation for a pure virtual function. The implementation is then mandatory. Additional functionality can be provided by the base class of the skeleton. In AUTOSAR, the derived class is typically addressed with an InstanceSpecifier (please refer to ara::com Skeleton as an example).

8.5.5.2.4 Behavior

Callbacks need to be registered and unregistered at runtime by the Adaptive Application.

8.5.5.2.5 Consequences

Results or notifications that occurred prior to the registration or after the unregistration will get lost. The `Callback` pattern introduces overhead for registering and unregistering. In case of the skeleton approach, additional runtime overhead may result from using a virtual function.

8.5.5.2.6 Examples

1. There are currently no such `Callback`s in the Adaptive Platform that are `Platform Extension Interface`s.
2. `DownloadService` in `Diagnostic Management`.

8.5.5.3 Plugin

The `Plugin` pattern relies on a dynamic registration of a class together with meta-information for selecting a `Plugin` at runtime.

8.5.5.3.1 Purpose

The `Plugin` pattern is intended to extend or adapt an existing framework with a larger number of functions.

8.5.5.3.2 Applicability

This pattern is applicable if any number of extensions shall provide their behavior to the Adaptive Platform an a concrete behavior needs to be chosen at runtime based on e.g., input data.

8.5.5.3.3 Class Structure

A `Plugin` is an implementation of an abstract base class. It is registered with meta-information (e.g., an identifier obtained by calling `GetID()` in figure 8.2) to be able to choose the right implementation at runtime. Alternatively, a plugin may decide itself if it can handle a certain request (e.g., by calling `Matches()` in figure 8.2).

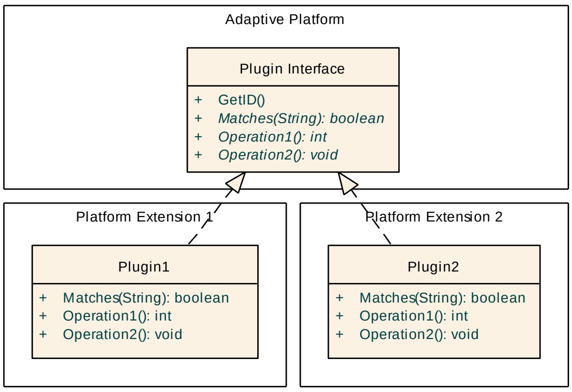

Figure 8.2: Class Structure of a Plugin

8.5.5.3.4 Behavior

The `Plugin` needs to be registered with the Adaptive Platform. The methods of the `Plugin` are then called by the Adaptive Platform.

8.5.5.3.5 Consequences

This pattern requires a registration of the `Plugin`(s) at runtime. It is recommended to register the plugins via the `Manifest`. This way, the AUTOSAR Adaptive Platform can ensure that the extension is initialized/deinitialized correctly. Having a fixed list of plugins defined in the `Manifest` also makes safety considerations easier because all possible implementations and the conditions under which they are used are known.

This pattern adds some runtime overhead to differentiate between the `Plugin`, usually by means of virtual functions. Using this pattern may complicate safety- and security audits since the concrete implementation is chosen at runtime.

8.5.5.3.6 Examples

- `UDS transport protocols` in `Diagnostic Management`
- `CryptoProvider` in `Cryptography`

8.5.5.3.7 Configuration Patterns

There are three supported variants to configure `Plugin`s in the `Manifest`. A variant may be chosen based on the pros and cons listed for each variant.

8.5.5.3.7.1 Reference to Class

In this variant, the meta-class for configuring the `Plugin` type (`XYPluginConfiguration` in figure 8.3) specifies an additional attribute (`implementation` in figure 8.3) that is used to define the full qualified name of the class that implements the plugin.

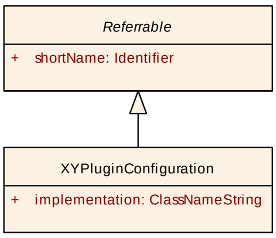

Figure 8.3: Excerpt of Meta-Model

In the `Manifest`, one may then create any number of such `Plugin` configurations and refer to the concrete classes that implement the `Plugin`s.

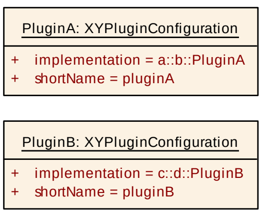

Figure 8.4: Example of Configuration in the Manifest

The tooling needs to generate additional code based on the `Manifest` that registers an instance of the class specified in the `Plugin` configuration with the Adaptive Platform using the identifier (e.g., the shortName) of the `Plugin`.

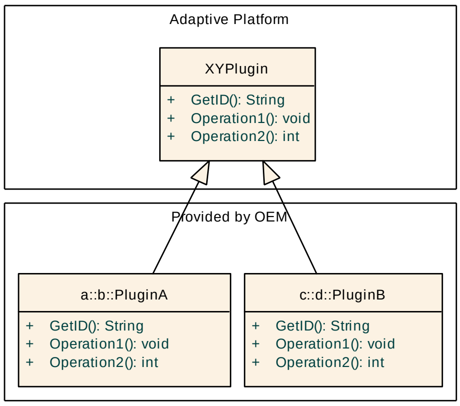

Figure 8.5: Example of Plugin Class Structure

Pros:

- High flexibility in the naming and namespace of the class that implements a `Plugin`.
- Immediate feedback by the compiler/linker if a `Plugin` implementation is missing or malformed.

Cons:

- References to implementation classes in the `Manifest`.

8.5.5.3.7.2 Generated Plugin Definition

This patterns defines `Plugin`s in the `Manifest` without specifying an implementation. A header file needs to be generated that defines a class with a unique name that implements the respective `Plugin` (e.g., the class name and namespace of the generated class is derived from the plugin's short name path in the `Manifest`). An implementation needs to be provided according to the generated header file. Additional code needs to be generated that registers an instance of the generated classes with the stack using the identifier (e.g., the shortName) of the `Plugin`.

In this variant, the meta-class for configuring the `Plugin` type (`XYPluginConfiguration`) does not specify any additional attribute that refers to the implementation of the `Plugin`.

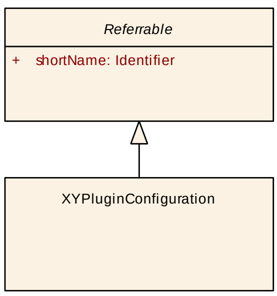

Figure 8.6: Excerpt of Meta-Model

In the Manifest, one may then create any number of such Plugin configurations.

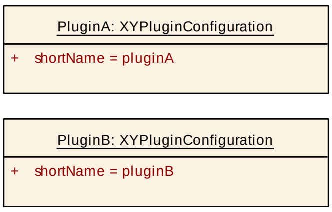

Figure 8.7: Example of Configuration in the Manifest

The tooling needs to generate a class definition (i.e. a header file) for each Plugin configuration. The name of that class is derived from the full qualified name of the Plugin configuration. The tooling also needs to generate additional code based on the Manifest that registers an instance of that generated class with the Adaptive Platform using the identifier (e.g., the shortName) of the Plugin. The OEM then needs to provide an implementation of the generated class definitions similar to the Callout pattern.

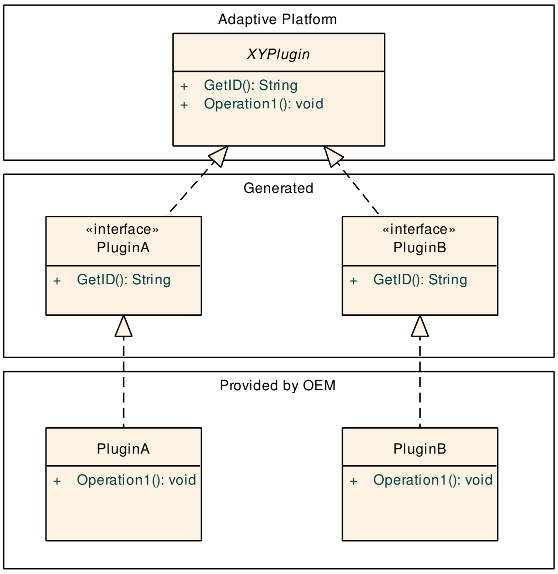

Figure 8.8: Example of Plugin Class Structure

Pros:

- No references to implementation classes in the Manifest.
- Immediate feedback by the compiler/linker if a Plugin implementation is missing or malformed.

Cons:

- No flexibility in the naming and namespace of the class that implements a Plugin. Implementations might not be directly reusable across projects.

8.5.5.3.7.3 Dynamic Library

This patterns defines plugins in the Manifest by means of dynamic libraries that are part of a SoftwareCluster. In this variant, the meta-class for configuring the plugin type (XYPluginConfiguration) refers to exactly one DynamicLibrary. The DynamicLibrary is packaged as a separate file in the resulting Software Package.

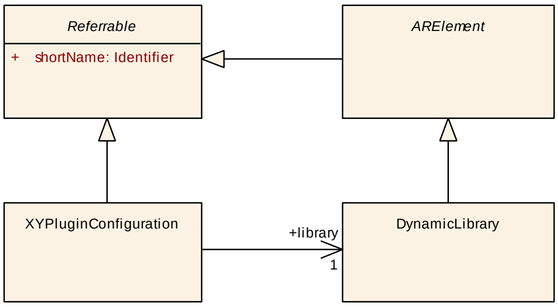

Figure 8.9: Excerpt of Meta-Model

In the Manifest, one may then create any number of such plugin configurations. The DynamicLibrary must contain an implementation of the plugin class which is a concrete class in this variant (i.e., the method names must be known). The registering/unregistering requires no generated code and can be done completely inside the AUTOSAR Adaptive Platform by loading the library dynamically. However, the operating system needs to support dynamic libraries.

Pros:

- No code generation / tooling required.
- High flexibility in the naming and namespace of the class that implements a Plugin.

Cons:

- Missing or malformed Plugin implementation can only be detected at runtime.
- May complicate safety considerations because it is harder to prove which code is actually executed.
- Cannot be used with statically linked binaries.

8.5.6 Offer/StopOffer Pattern

The Offer/StopOffer pattern was originally invented to control the invocations of callbacks for ara::com service interfaces. The pattern is called ’Offer/StopOffer’ due to the names of the two functions that enable and disable invocations. The Offer/StopOffer pattern generalized that approach and is now applied in several different functional clusters.

8.5.6.1 Purpose

The `Offer/StopOffer pattern` controls when asynchronous callbacks may be invoked on an object. It decouples the lifecycle of an object from the ability to handle asynchronous callbacks. The `Offer/StopOffer pattern` guards against race-conditions that could arise if callbacks were already enabled during construction and the callback would be invoked while the object is not yet fully initialized.

8.5.6.2 Applicability

The `Offer/StopOffer pattern` is applicable if the handling of asynchronous callbacks shall be decoupled from the registration of the callback or the lifetime of the object that defines a callback function.

8.5.6.3 Class Structure

The `Offer/StopOffer pattern` has two variants (see Figure 8.10).

- **Variant 1 - Skeleton Approach**: The first variant is called 'skeleton approach' because it was first used in the skeleton class generated for an ara::com service interface. This variant makes use of inheritance. The `Offer` and `StopOffer` functions are defined and implemented in an abstract base class (`ServiceSkeleton`). One or more pure virtual function(s) in the abstract base class define the signature(s) of the callback(s) (`ServiceOperation`). An application needs to provide a concrete class that implements the pure virtual function(s). The functions `Offer` and `StopOffer` should be `final`.

  This variant is susceptible to several race-conditions. The application-provided destructor needs to call `StopOffer` to prevent invocations of the callback on a destructed object. For the same reason, the object must not be moved while it is in the `offered` state. One of the following two solutions should be selected on a case-by-case decision to mitigate the problem:

  – Disable the move constructor and assignment operator.

  – Introduce a Violation in case either the move constructor or assignment operator is called while the the object is in the `offered` state.

- **Variant 2 - std::function**: The second variant does not require inheritance. The `Offer` and `StopOffer` functions are defined in a concrete class (`Service`) together with a function to register a `std::function` (provided by the application) as a callback (`RegisterCallback`). The class `Service` should be `final`.

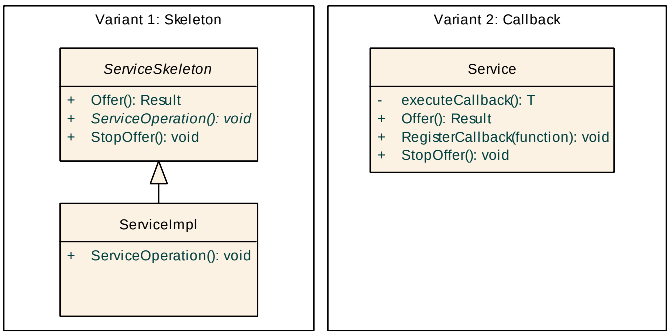

Figure 8.10: Variants of the Offer/StopOffer Pattern

8.5.6.3.1 Offer()

`Offer` should be defined as:

```
ara::core::Result<void> Offer() final noexcept;
```

`Offer` should ignore repeated calls (without calling `StopOffer` in between). Other kinds of errors can be reported with the `Result`, for example if a daemon process is not available. The application is responsible for retrying the operation or e.g., terminating. An application is free to add logical supervisions to detect repeated / unintended calls to `Offer`.

8.5.6.3.2 StopOffer()

`StopOffer` should be defined as:

```
void StopOffer() final noexcept;
```

`StopOffer` should ignore repeated calls (without calling `Offer` in between). `StopOffer` is not expected to fail in a recoverable way (that is why no `Result` is returned). Once the call to `StopOffer` has completed, the callback is not invoked afterwards. If a callback is executed when `StopOffer` is called, `StopOffer` needs to block until the callback has finished. That way, the application can be sure that once `StopOffer` has completed it is safe to free any resources required by the callback. An application is free to add logical supervisions to detect repeated / unintended calls to `StopOffer`.

Please note that the `Offer/StopOffer pattern` can be extended in several ways to fit specific use cases, for example to have control over the order and execution context (thread) of callbacks by means of an `Executor`. Also the names of the `Offer`, `StopOffer`, and all other functions are not fixed and more meaningful names could be chosen.

8.5.6.3.3 Participants

- **Variant 1 - Skeleton Approach**:
  - **ServiceSkeleton**: An abstract class providing the functionality to enable/disable the invocation of the callback(s) (`Offer` and `StopOffer`). It also defines at least one pure virtual function at defines the signature for the callback (`ServiceOperation`). The `ServiceSkeleton` is defined as part of the Adaptive Platform API.
  - **ServiceImpl**: Inherits from the `ServiceSkeleton` and implements the callbacks. The `ServiceImpl` class is part of the application.
- **Variant 2 - std::function**:
  - **Service**: Provides functionality to register one or more callbacks (`RegisterCallback`, but could also be done in a constructor if the callback is mandatory) and functionality to enable/disable the invocation of the callback(s) (`Offer` and `StopOffer`).

8.5.6.4 Behavior

Figure 8.11 shows the registration, the usage, and the de-registration of a callback for **Variant 1 - Skeleton Approach**.

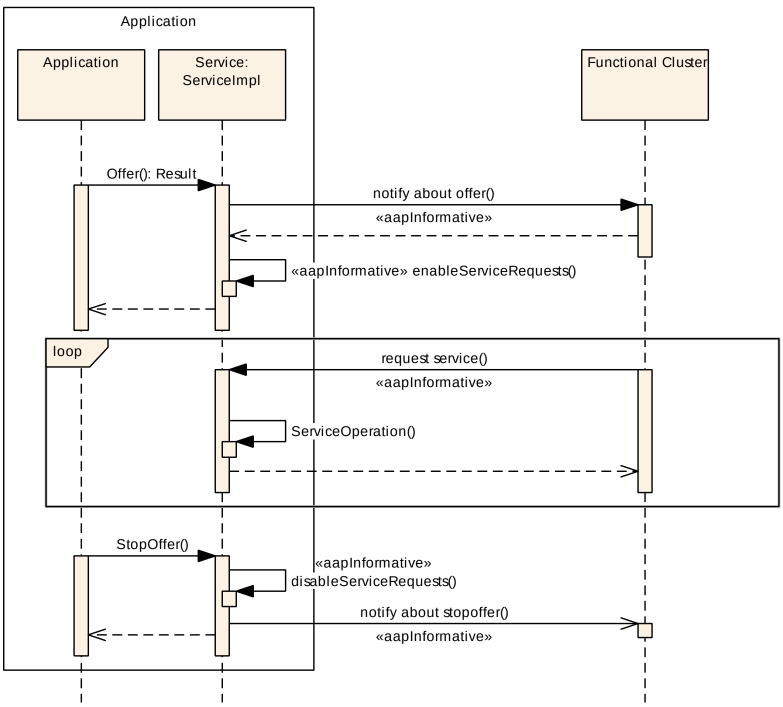

Figure 8.11: Variant 1 - Skeleton Approach

Figure 8.12 shows the registration, the usage, and the de-registration of a callback for **Variant 2 - std::function**.

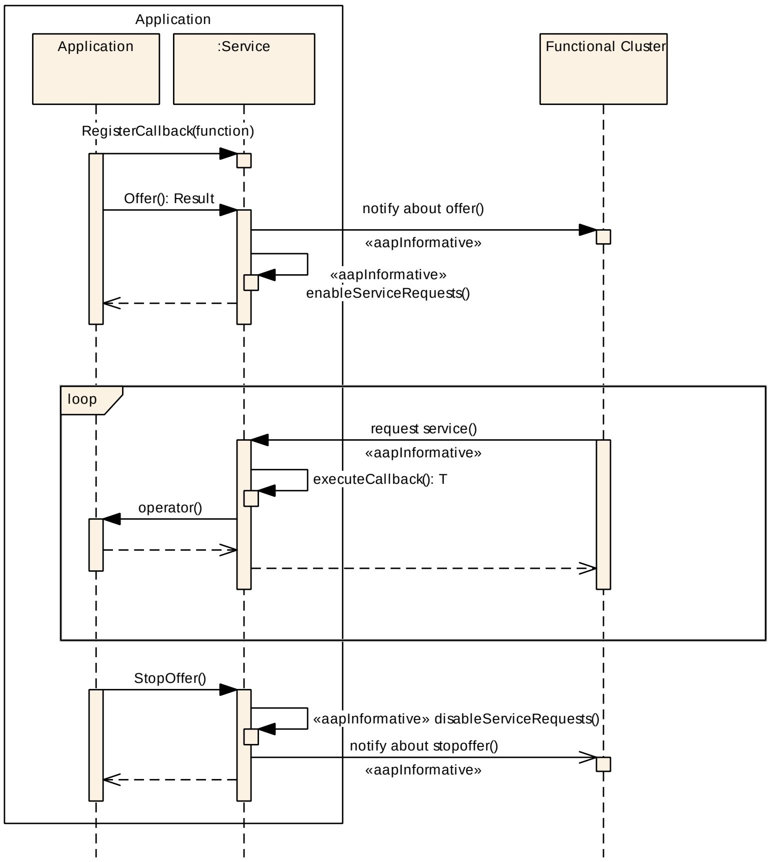

Figure 8.12: Variant 2 - std::function

8.5.6.5 Consequences

The Offer/StopOffer pattern requires more setup due to two separate calls to create/register the callback and to enable invocations. **Variant 1 - Skeleton Approach** may also introduce a small runtime overhead due to the use of virtual functions.

8.5.6.6 Examples

- Generated skeleton for ara::com service interfaces.
- SecurityAccess and many other APIs in Diagnostic Management.
- RecoveryAction in Platform Health Management

8.6 Deployment

The AUTOSAR Adaptive Platform supports the incremental deployment of applications, where resources and communications are managed dynamically to reduce the effort for software development and integration, enabling short iteration cycles. Incremental deployment also supports explorative software development phases. For product delivery, the AUTOSAR Adaptive Platform allows the system integrator to carefully limit dynamic behavior to reduce the risk of unwanted or adverse effects allowing safety qualification. Dynamic behavior of an application will be limited by constraints stated in the `Execution Manifest` (cf. Section 13.8), for example, dynamic allocation of resources and communication paths are only possible in defined ways, within configured ranges. Implementations of an AUTOSAR Adaptive Platform may further remove dynamic capabilities from the software configuration for production use. Examples of reduced behavioral dynamics might be:

- Pre-determination of the service discovery process
- Restriction of dynamic memory allocation to the startup phase only
- Fair scheduling policy in addition to priority-based scheduling
- Fixed allocation of processes to CPU cores
- Access to pre-existing files in the file-system only
- Constraints for AUTOSAR Adaptive Platform API usage by applications
- Execution of authenticated code only

8.7 Verification and Validation

The AUTOSAR Adaptive Platform standard uses a dedicated implementation of the standard (AUTOSAR Adaptive Platform Demonstrator) to validate the requirements and to verify the (still abstract) software design imposed by the individual software specifications.

9 Building Block View

This chapter provides an overview of the static structure of the AUTOSAR Adaptive Platform by describing the high-level building blocks and their inter-dependencies. Please note that the use of interfaces between Functional Clusters in the AUTOSAR Adaptive Platform is currently not standardized. An implementation of the AUTOSAR Adaptive Platform is free to define its own internal interfaces between Functional Clusters or use different standardized interfaces than presented in this document.

The description of all building blocks (FunctionalClusters) in this chapter uses the same pattern. Each FunctionalCluster is described in a separate section of the document that starts with an overview of the FunctionalCluster as outlined in Table 9.1. The overview is followed by a sub-section called "Defined interfaces" that lists all architectural interfaces specified in the namespace(s) of the FunctionalCluster. Each interface is detailed in a separate table. Please be aware that those interfaces do not correspond to classes in the ARA API in all cases, for example, there are additional interfaces defined to group C++ non-member functions because there is no such concept in UML.

Name:	The long name of the functional cluster.
Short Name:	The short name of the functional cluster.
Category:	The category of the functional cluster (see Chapter 9.1 for an overview).
Daemon-based:	Denotes if the FunctionalCluster is intended to be implemented with a separate daemon process or not. An implementation of the AUTOSAR Adaptive Platform is free to deviate from that.
Responsibilities:	A description of the responsibilities of the FunctionalCluster.

Table 9.1: Example of a Functional Cluster overview table

The second to last subsection "Provided interfaces" lists all interfaces provided by the FunctionalCluster to other FunctionalClusters. The last subsection "Required interfaces" lists all interfaces required by the FunctionalCluster from other FunctionalClusters and external components like the operating system. However, an implementation of the AUTOSAR Adaptive Platform is free to define its own internal interfaces between Functional Clusters or use different standardized interfaces than presented in the sections "Provided interfaces" and "Required interfaces".

Please note that the «use» of an interface can also include the user passing callbacks (see Chapter 8.5.5.2) to the interface or implementing abstract functions, even though the provider of the interface will call code that is provided by its user.

9.1 Overview

Figure 9.1 provides an overview of the different categories of building blocks available in the AUTOSAR Adaptive Platform. The categories are explained in more detail in the subsequent sections.

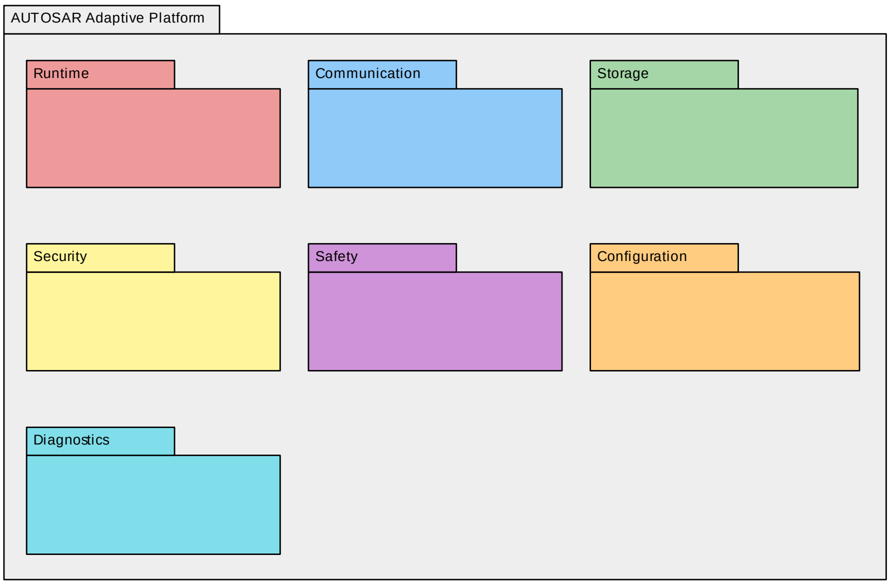

Figure 9.1: Overview of the AUTOSAR Adaptive Platform and its building block categories

9.2 Runtime

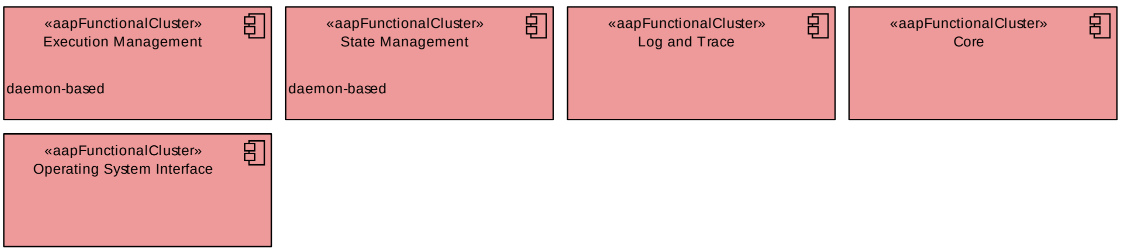

Figure 9.2: Functional Clusters in category Runtime

9.2.1 Execution Management

Name:	Execution Management
Short Name:	exec
Category:	Runtime
Daemon-based:	Yes

▽

△

Responsibilities:	Execution Management is responsible to control Processes of the AUTOSAR Adaptive Platform and Adaptive Applications. That is, it starts, configures, and stops Processes as configured in Function Group States using interfaces of the Operating System. The Operating System is responsible for runtime scheduling of those Processes. The configuration of Processes that Execution Management performs includes limiting their resource consumption (CPU time, memory) using Resource Groups provided by the Operating System. Execution Management is the entry point of AUTOSAR Adaptive Platform and is started by the Operating System during system boot. Execution Management then controls the startup and shutdown of the AUTOSAR Adaptive Platform (see use cases Start Adaptive Platform and Shutdown Adaptive Platform for details). Execution Management optionally supports authenticated startup where it maintains the chain of trust when starting from a Trust Anchor. During authenticated startup Execution Management validates the authenticity and integrity of Processes and shall prevent their execution if violations are detected. Through these mechanisms, a trusted platform can be established.

9.2.1.1 Defined interfaces

The interfaces of Execution Management are categorized into interfaces for state reporting (see Section 9.2.1.1.1) and interfaces for State Management (see Section 9.2.1.1.2).

9.2.1.1.1 Interfaces for state reporting

All processes started by Execution Management (i.e. all processes of the AUTOSAR Adaptive Platform and all processes of Adaptive Applications) shall report their execution state back to Execution Management via the ExecutionClient interface (cf. Figure 9.3).

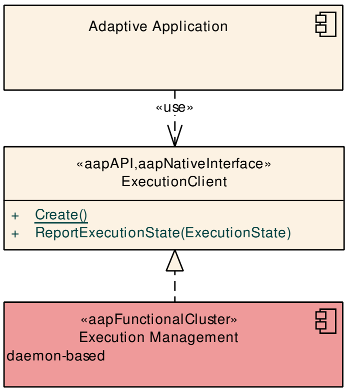

Figure 9.3: Interfaces for state reporting

Name:	ExecutionClient
Technology:	Native interface
Usage:	Public API

▽

△

Description:	This interface provides functionality for a Process to report its execution state to Execution Management.	
Operations:	Create	Named constructor for this interface.
	ReportExecutionState	Report the internal state of a Process to Execution Management.

9.2.1.1.2 Interfaces for State Management

The StateClient API (cf. Figure 9.4) provides operations to control FunctionGroupStates.

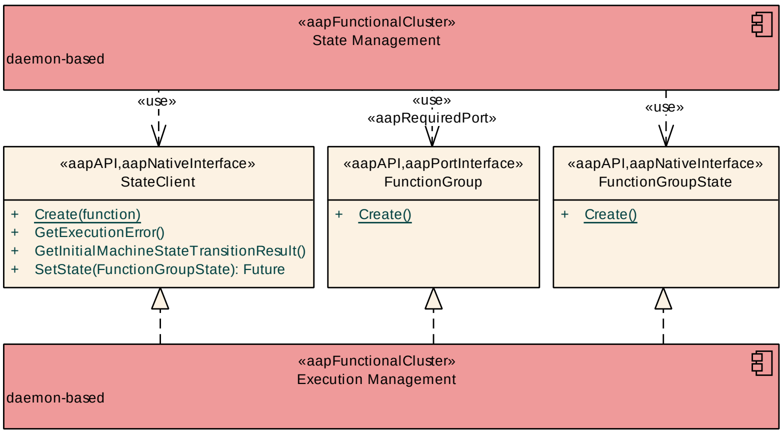

Figure 9.4: Interfaces for State Management

Name:	StateClient	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface provides the functionality to request FunctionGroupState transitions and to perform error detection and error handling.	
Operations:	Create	Named constructor of this interface. It requires a callback to be invoked if a FunctionGroup changes its state unexpectedly to an undefined FunctionGroup State, i.e. without previous request by SetState(). The affected FunctionGroup is provided as an argument to the callback.
	GetExecutionError	Returns the execution error which changed the given FunctionGroup to an undefined FunctionGroup State.
	GetInitialMachineStateTransitionResult	Retrieve the result of Machine State initial transition to Startup state.
	SetState	Request a FunctionGroupState transition for a single FunctionGroup.

Name:	FunctionGroup	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`StateClientInterface`	
Usage:	Public API	
Description:	Represents a `Function Group` defined in the `Manifest`.	
Operations:	Create	Creates a `FunctionGroup`.

Name:	FunctionGroupState	
Technology:	Native interface	
Usage:	Public API	
Description:	Represents a `Function Group State` defined in the `Manifest`.	
Operations:	Create	Creates a `FunctionGroupState`.

9.2.1.2 Provided interfaces

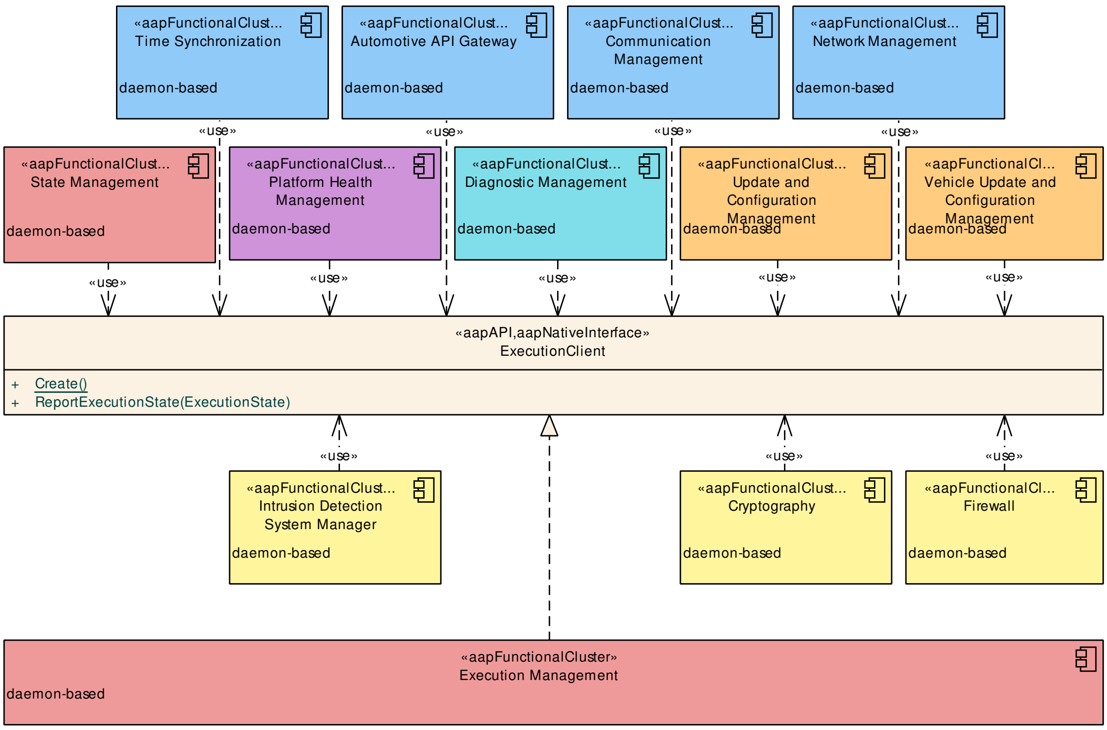

Figure 9.5: Users of the ExecutionClient interface

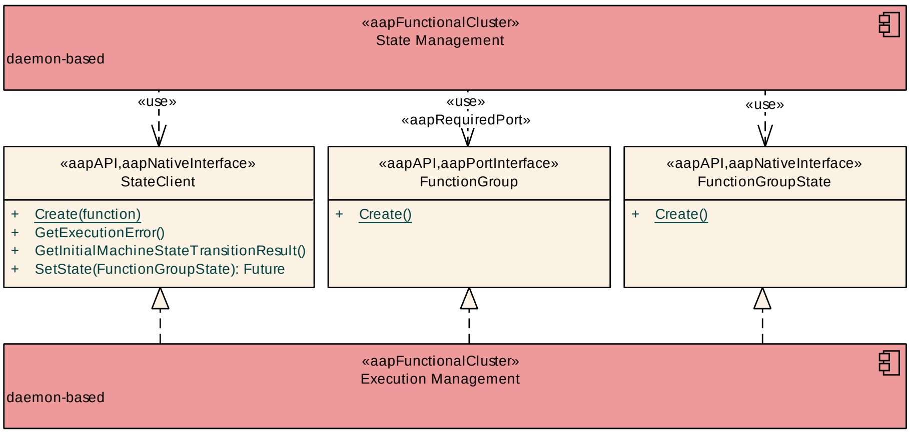

Figure 9.6: Users of the State Management interfaces

Interface	*Requiring functional clusters*
ExecutionClient	Automotive API Gateway
	Communication Management
	Cryptography
	Diagnostic Management
	Firewall
	Intrusion Detection System Manager
	Network Management
	Platform Health Management
	State Management
	Time Synchronization
	Update and Configuration Management
	Vehicle Update and Configuration Management
FunctionGroup	State Management
FunctionGroupState	State Management
StateClient	State Management

Table 9.2: Interfaces provided by Execution Management to other Functional Clusters

9.2.1.3 Required interfaces

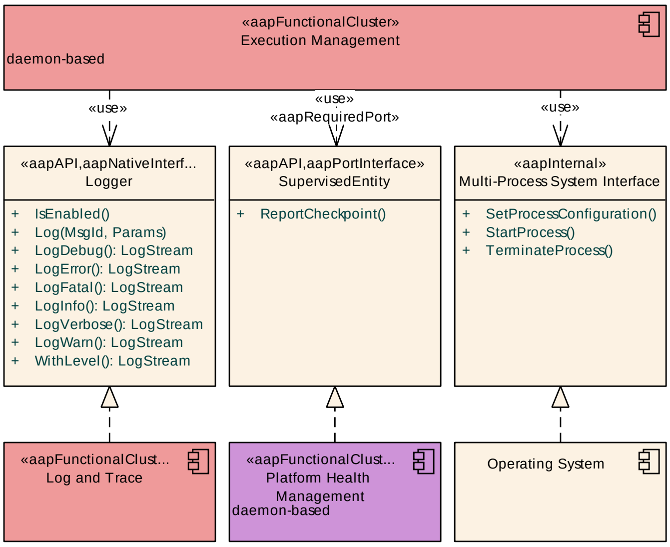

Figure 9.7: Interfaces required by Execution Management

Interface	*Purpose*
ManifestAccessor	Execution Management shall use this interface to read the configuration of Function Groups and Processes from the Manifest.
Multi-Process System Interface	Execution Management shall use this interface to start, configure and control os-level processes.
Log and Trace::Logger	Execution Management shall use this interface to log standardized messages.
Platform Health Management::SupervisedEntity	Execution Management shall use this interface to enable supervision of its process(es) by Platform Health Management.

Table 9.3: Interfaces required by Execution Management

9.2.2 State Management

Name:	State Management
Short Name:	sm
Category:	Runtime
Daemon-based:	Yes
Responsibilities:	State Management determines the desired target state of the AUTOSAR Runtime for Adaptive Applications based on various application-specific inputs. That target state is the set of active Function Group States running on the AUTOSAR Runtime for Adaptive Applications. State Management delegates to Execution Management to switch the individual Function Group States. State Management is a unique component in the AUTOSAR Adaptive Platform because it is not entirely part of an AUTOSAR Adaptive Platform stack. The logic of State Management may be implemented as application-specific code and then configured and integrated with an AUTOSAR Adaptive Platform stack.

9.2.2.1 Defined interfaces

The interfaces of State Management are categorized into interfaces interfaces for a SMControlApplication (see Section 9.2.2.1.1), interfaces for notification of StateMachine transitions to any Adaptive Application (see Section 9.2.2.1.2), interfaces for interaction with Update and Configuration Management (see Section 9.2.2.1.3), and interfaces for Suspend-to-Ram (see Section 9.2.2.1.4).

9.2.2.1.1 Interfaces for a SMControlApplication

The StateMachineService and UpdateAllowedService interfaces provide a means for an SMControlApplication to interact with State Management's StateMachine.

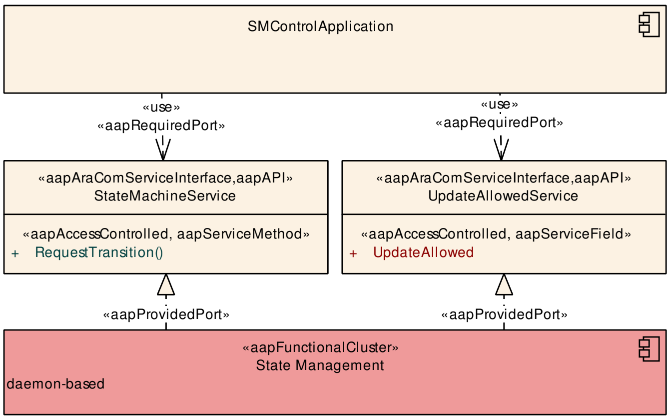

Figure 9.8: Interfaces for a SMControlApplication

Name:	StateMachineService	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface is intended to be used by a `SMControlApplication` to interact with State Management's StateMachine.	
Operations:	RequestTransition	Request a transition in the referenced StateMachine.

Name:	UpdateAllowedService	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface is intended to be used by a `SMControlApplication` to grant an update session or not.	
Fields:	UpdateAllowed	Controls if an update is allowed or not.

9.2.2.1.2 Interfaces for StateMachine Notifications

The `StateMachineNotification` interface provides means for any Adaptive Application to retrieve or get notifications about the current state of a particular StateMachine. If a StateMachine is currently in transition between two different states, then the value is `InTransition`.

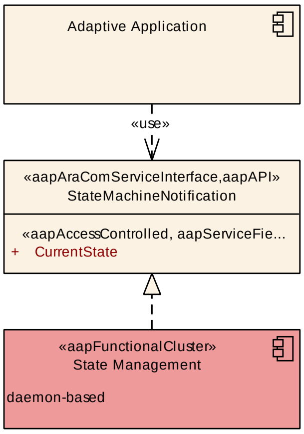

Figure 9.9: Notification interface for an Adaptive Application

Name:	StateMachineNotification	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface provides means to retrieve or get notifications about the current state of a particular StateMachine.	
Fields:	CurrentState	The current state of the StateMachine.

9.2.2.1.3 Interfaces for interaction with Update and Configuration Management

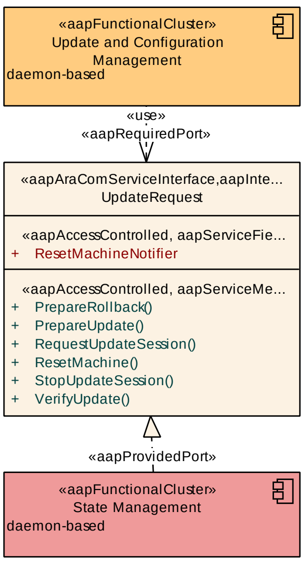

Figure 9.10: Interface for software update handling

Name:	UpdateRequest	
Technology:	ara::com service interface	
Usage:	Internal	
Description:	This interface is intended to be used by Update and Configuration Management to interact with State Management to perform updates (including installation and removal) of Software Clusters.	
Operations:	PrepareRollback	Prepares the affected Function Groups for a rollback.
	PrepareUpdate	Prepares the affected Function Groups for an update.
	RequestUpdateSession	Requests an update session. State Management might decline this request when the Machine is not in a state to be updated.
	ResetMachine	Requests an orderly reset of the Machine. Before the reset is performed all information within the Machine shall be persisted.
	StopUpdateSession	Ends an update session.
	VerifyUpdate	Verifies the affected Function Groups after an update.
Fields:	ResetMachineNotifier	To be set by State Management to inform Update and Configuration Management about changes during and after processing the method ResetMachine.

9.2.2.1.4 Interfaces for Suspend-to-Ram Notifications

The S2RHub provides means for a State Management implementation or an SM-ControlApplication to request entering or leaving a "prepared for Suspend-to-RAM" state from Adaptive Applications implementing S2RSatellite. S2RHub is not intended to be used by regular Adaptive Applications.

S2RSatellite provides means for any Adaptive Application to be notified to enter- or leave the process-local "prepared for Suspend-to-RAM" state. If the state is entered, the Adaptive Application should prepare to be suspend by the operating system at anytime. This could be a storage of persistent data, the stop of offering services, the stop of any safety activities, etc.

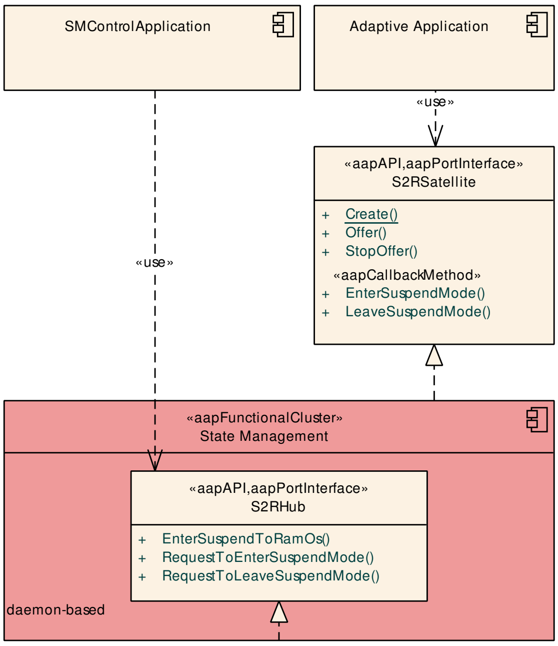

Figure 9.11: Interfaces for Suspend-to-RAM

Name:	S2RHub	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	SuspendToRamHubInterface	
Usage:	Public API	
Description:	This interface provides functionality to request entering or leaving a "prepared for Suspend-to-RAM" state and to actually enter the Suspend-to-RAM state. This interface is intended to be used only within State Management or by an SMControlApplication.	
Operations:	EnterSuspendToRamOs	Function to trigger the operating system to enter suspend-to-RAM state.
	RequestToEnterSuspendMode	Function to request all S2RSatellites to enter suspend state.
	RequestToLeaveSuspendMode	Function to request all registered S2RSatellites to leave suspend state.

Name:	S2RSatellite	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	SuspendToRamSatelliteInterface	
Usage:	Public API	
Description:	Suspend-to-RAM satellite.	
Operations:	Create	
	EnterSuspendMode	Called to enter Suspend Mode.
	LeaveSuspendMode	Called to leave Suspend Mode.
	Offer	Enables potential invocations of handlers.
	StopOffer	Disables invocations of handlers.

9.2.2.2 Provided interfaces

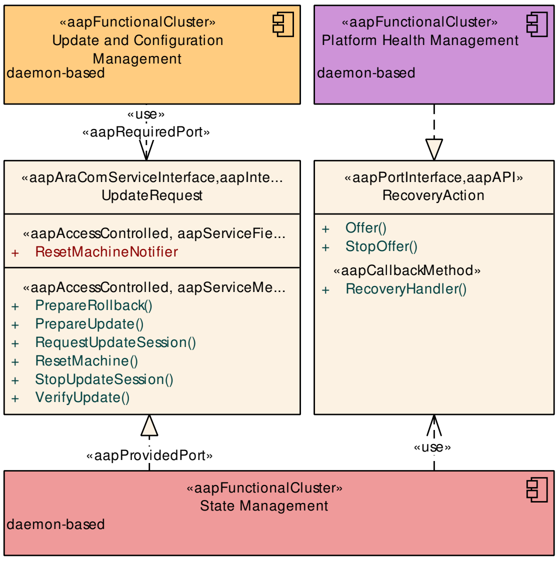

Figure 9.12: Users of the State Management interfaces

Interface	*Requiring functional clusters*
UpdateRequest	Update and Configuration Management

Table 9.4: Interfaces provided by State Management to other Functional Clusters

9.2.2.3 Required interfaces

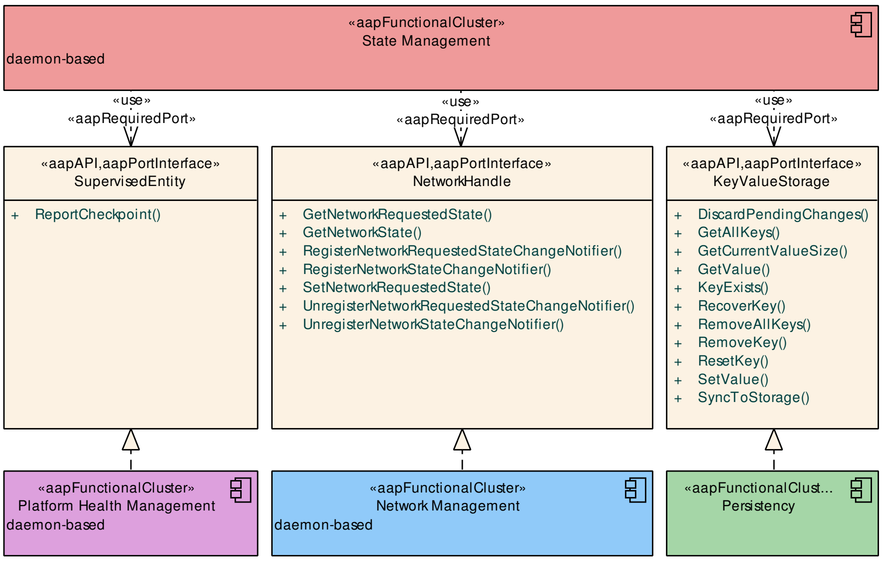

Figure 9.13: Interfaces required by State Management

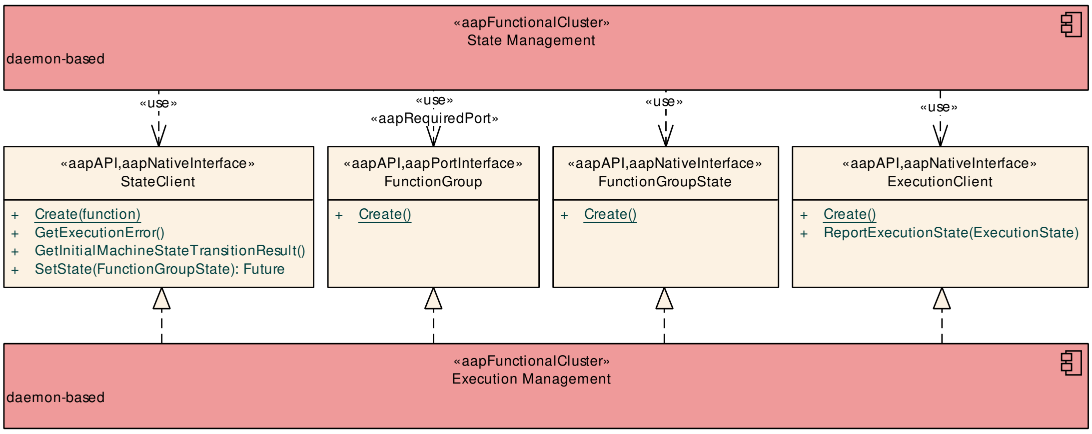

Figure 9.14: Interfaces of Execution Management required by State Management

Interface	***Purpose***
Execution Management::ExecutionClient	This interface shall be used to report the state of the State Management process(es).
Execution Management::FunctionGroup	This interface shall be used to construct FunctionGroupStates.
Execution Management::FunctionGroupState	This interface shall be used to request FunctionGroupState transitions.
Execution Management::StateClient	This interface shall be used to request FunctionGroupState transitions.
Log and Trace::Logger	State Management shall use this interface to log standardized messages.
Network Management::NetworkHandle	This interface shall be used to retrieve information about the network status of a NetworkHandle.
Persistency::KeyValueStorage	Used to store the internal state of State Management.
Platform Health Management::RecoveryAction	Platform Health Management uses this interface to trigger failure recovery.
Platform Health Management::SupervisedEntity	State Management shall use this interface to enable supervision of its process(es) by Platform Health Management.

Table 9.5: Interfaces required by State Management

9.2.3 Log and Trace

Name:	Log and Trace
Short Name:	log
Category:	Runtime
Daemon-based:	No
Responsibilities:	Log and Trace provides functionality to build and log messages of different severity. An Adaptive Application can be configured to forward log messages to various sinks, for example to a network, a serial bus, the console, and to non-volatile storage.

9.2.3.1 Defined interfaces

The entry point to the Log and Trace framework is the CreateLogger() operation that constructs a new Logger context. Afterwards, new messages can be constructed using the LogStream that is returned by the operations in Logger, for example LogInfo().

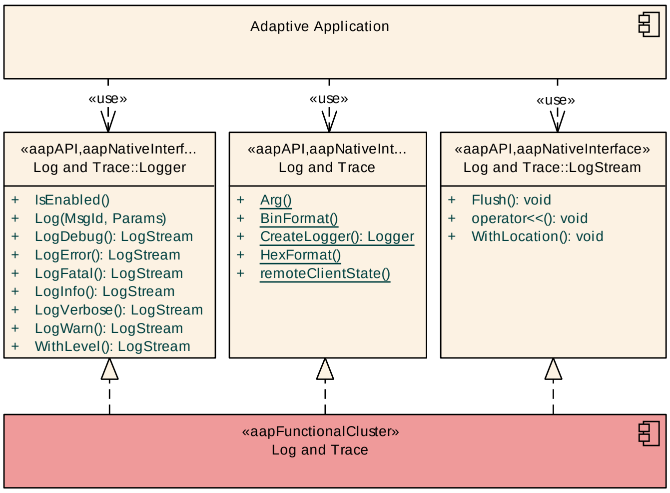

Figure 9.15: Interfaces of Log and Trace

Name:	Logger	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface represents a logger context. The logging framework defines contexts which can be seen as logger instances within one application process or process scope. A context will be automatically registered against the Logging back-end during creation phase, as well as automatically de-registered during process shutdown phase.	
Operations:	IsEnabled	Check if the provided log level is enabled in the current configured log level.
	Log	Logs a message modeled in the Manifest.
	LogDebug	Creates a LogStream object with Debug severity.
	LogError	Creates a LogStream object with Error severity.
	LogFatal	Creates a LogStream object with Fatal severity.
	LogInfo	Creates a LogStream object with Info severity.
	LogVerbose	Creates a LogStream object with Verbose severity.
	LogWarn	Creates a LogStream object with Warn severity.
	WithLevel	Creates a LogStream object with the provided log level.

Name:	LogOperations
Technology:	Native interface
Usage:	Public API
Description:	This interface provides access to the logging framework and utility operations to control the format of value printed to the log output.

▽

△

Operations:	Arg	Create a wrapper object. The wrapper object holds a value and an optional name and unit of the value.
	BinFormat	Conversion of an integer into a binary value. Negatives are represented in 2's complement. The number of represented digits depends on the overloaded parameter type length.
	CreateLogger	Creates a `Logger` object, holding the context which is registered in the logging framework.
	HexFormat	Conversion of an integer into a hexadecimal value. Negatives are represented in 2's complement. The number of represented digits depends on the overloaded parameter type length.
	remoteClientState	Fetches the connection state from the `DLT` back-end of a possibly available remote client.

Name:	LogStream	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface provides functionality to construct a single log message by appending data using stream operators.	
Operations:	Flush	Sends out the current log buffer and initiates a new message stream. Calling this operation is only necessary if the `LogStream` is intended to be reused within the same scope.
	WithLocation	Add source file location into the message.
	operator<<	Writes a value into the log message. Several overloads exist to control the output format.

9.2.3.2 Provided interfaces

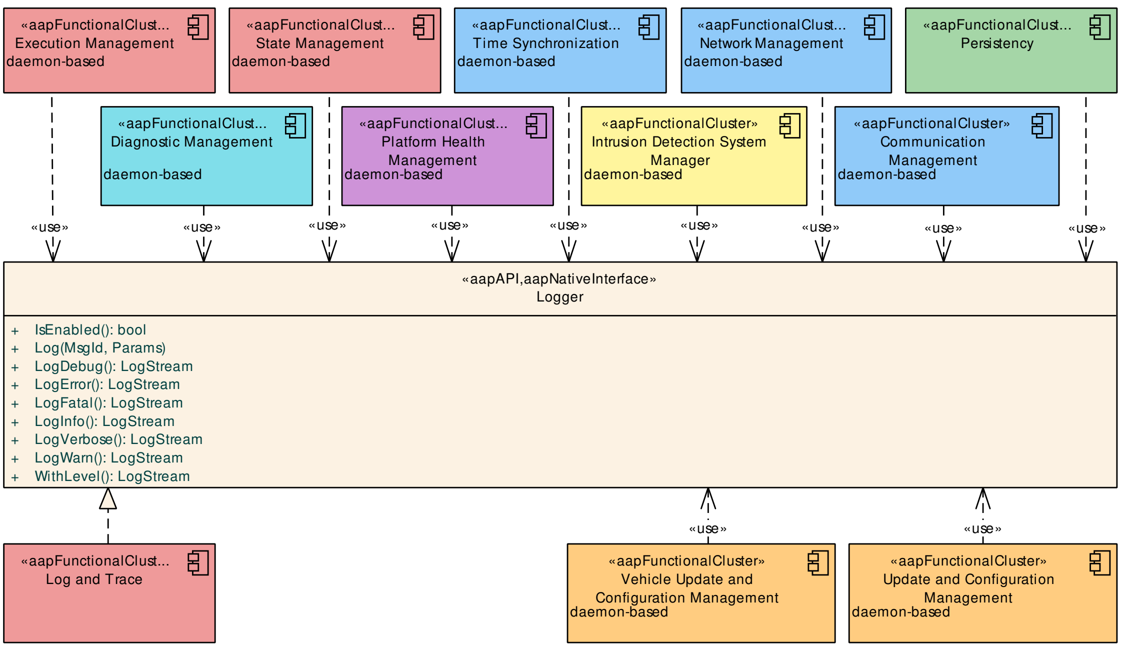

Figure 9.16: Users of the Log and Trace interfaces

Interface	Requiring functional clusters
Logger	Automotive API Gateway
	Communication Management
	Diagnostic Management
	Execution Management
	Intrusion Detection System Manager
	Network Management
	Persistency
	Platform Health Management
	Raw Data Stream
	State Management
	Time Synchronization
	Update and Configuration Management
	Vehicle Update and Configuration Management

Table 9.6: Interfaces provided by Log and Trace to other Functional Clusters

9.2.3.3 Required interfaces

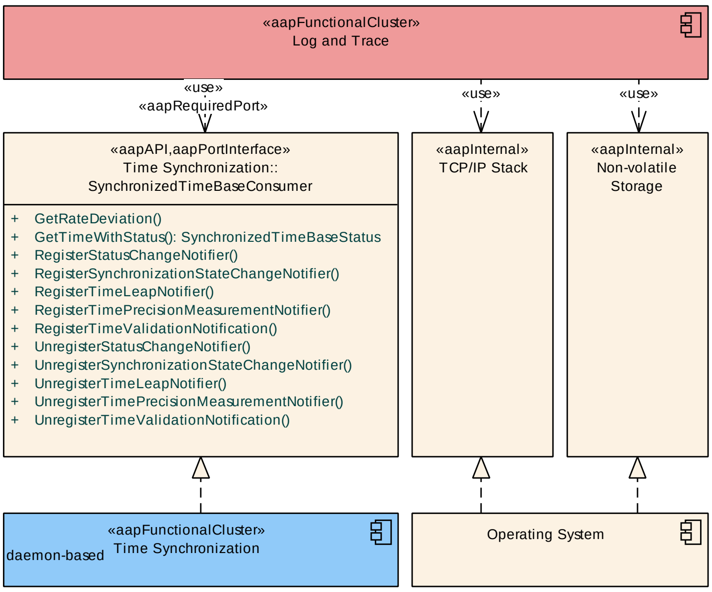

Figure 9.17: Interfaces required by LogAndTrace

Interface	Purpose
ManifestAccessor	`Log and Trace` shall use this interface to read information about modeled messages from the `Manifest`s.
Non-volatile Storage	`Log and Trace` should use this interface to write log messages to a non-volatile storage, e.g., a file in a filesystem.

▽

△

Interface	*Purpose*
TCP/IP Stack	Log and Trace shall use this interface to write log messages to an IP-based network stream.
Time Synchronization::SynchronizedTimeBase Consumer	Log and Trace shall use this interface to determine the timestamps that are associated with log messages.

Table 9.7: Interfaces required by Log and Trace

9.2.4 Core

Name:	Core
Short Name:	core
Category:	Runtime
Daemon-based:	No
Responsibilities:	Core provides functionality for initialization and de-initialization of the AUTOSAR Runtime for Adaptive Applications as well as termination of Processes.

9.2.4.1 Defined interfaces

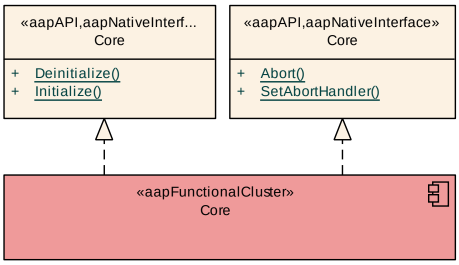

Figure 9.18: Interfaces of Core

9.2.4.1.1 Interfaces for initialization and de-initialization

The AUTOSAR Adaptive Platform for Applications needs to be initialized by an application before it is used (using Initialize()) and de-initialized after it is no longer used (using Deinitialize()).

Name:	InitAndShutdown
Technology:	Native interface
Usage:	Public API
Description:	This interface provides global initialization and shutdown functions that initialize respectively de-initialize data structures and threads of the AUTOSAR Runtime for Adaptive Applications.

▽

△

Operations:	Deinitialize	Destroy all data structures and threads of the AUTOSAR Adaptive Runtime for Applications. After this call, no interaction with the AUTOSAR Adaptive Runtime for Applications is possible.
	Initialize	Initializes data structures and threads of the AUTOSAR Adaptive Runtime for Applications. Prior to this call, no interaction with the AUTOSAR Adaptive Runtime for Applications is possible.

9.2.4.1.2 Interfaces for process termination

The AUTOSAR Adaptive Platform for Applications provides an explicit abnormal termination facility using `Abort()`.

Name:	ProcessTermination	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface provides operation for abnormal termination of processes.	
Operations:	Abort	Abort the current process. This function will never return to its caller.
	SetAbortHandler	Set a custom global abort handler function and return the previously installed one.

9.2.4.1.3 Interfaces for memory resources

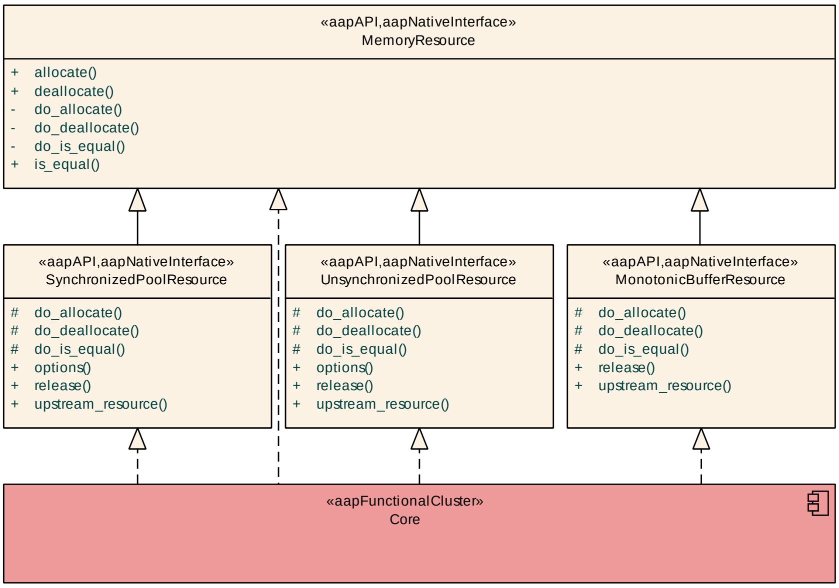

Figure 9.19: Interfaces for memory resources

Name:	MemoryResource	
Technology:	Native interface	
Usage:	Public API	
Description:	Interface for memory resources.	
Operations:	allocate	Allocates the specified amount of memory from this MemoryResource.
	deallocate	Deallocates memory from this MemoryResource.
	do_allocate	Allocates the specified amount of memory from this MemoryResource.
	do_deallocate	Deallocates memory from this MemoryResource.
	do_is_equal	Compare for equality with another MemoryResource.
	is_equal	Compare for equality with another MemoryResource.

Name:	MonotonicBufferResource	
Technology:	Native interface	
Usage:	Public API	
Description:	MonotonicBufferResource is a special-purpose MemoryResource that releases the allocated memory only when the resource is destroyed. It is intended for very fast memory allocations in situations where memory is used to build up a few objects and then is released all at once.	
Operations:	do_allocate	Allocates the specified amount of memory from this MemoryResource.
	do_deallocate	Deallocates memory from this MemoryResource.
	do_is_equal	Compare for equality with another MemoryResource.
	release	Release all allocated memory.
	upstream_resource	Get the upstream MemoryResource.

Name:	SynchronizedPoolResource	
Technology:	Native interface	
Usage:	Public API	
Description:	SynchronizedPoolResource is a special-purpose MemoryResource that may be accessed from multiple threads without external synchronization. Allocation requests are served from a collection of pools that serve different block sizes.	
Operations:	do_allocate	Allocates the specified amount of memory from this MemoryResource.
	do_deallocate	Deallocates memory from this MemoryResource.
	do_is_equal	Compare for equality with another MemoryResource.
	options	Returns the options that control the pooling behavior of this resource.
	release	Release all allocated memory.
	upstream_resource	Get the upstream MemoryResource.

Name:	UnsynchronizedPoolResource
Technology:	Native interface
Usage:	Public API
Description:	UnsynchronizedPoolResource is a special-purpose MemoryResource that is not thread-safe. Allocation requests are served from a collection of pools that serve different block sizes.

▽

Operations:	do_allocate	Allocates the specified amount of memory from this MemoryResource.
	do_deallocate	Deallocates memory from this MemoryResource.
	do_is_equal	Compare for equality with another MemoryResource.
	options	Returns the options that control the pooling behavior of this resource.
	release	Release all allocated memory.
	upstream_resource	Get the upstream MemoryResource.

9.2.4.1.4 Interfaces for memory handling

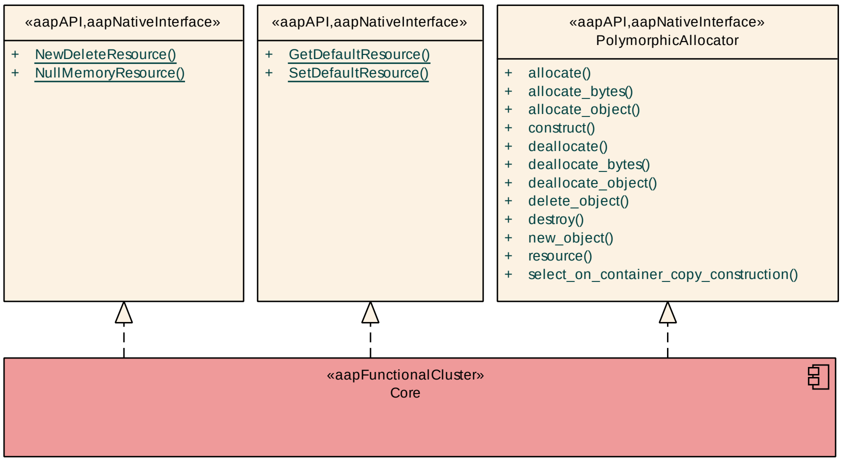

Figure 9.20: Interfaces for memory handling

Name:	StandardMemoryResourceOperations	
Technology:	Native interface	
Usage:	Public API	
Description:	Interface to get standard MemoryResources.	
Operations:	NewDeleteResource	Get a MemoryResource that uses the global operator new and operator delete to allocate memory.
	NullMemoryResource	Get a MemoryResource that doesn't perform any allocation.

Name:	DefaultMemoryResourceOperations	
Technology:	Native interface	
Usage:	Public API	
Description:	Interface to manage the default MemoryResource.	
Operations:	GetDefaultResource	Get the default MemoryResource.
	SetDefaultResource	Set the default MemoryResource to be used.

Name:	PolymorphicAllocator	
Technology:	Native interface	
Usage:	Public API	
Description:	An Allocator which exhibits different allocation behavior depending upon the MemoryResource from which it is constructed.	
Operations:	allocate	Allocate memory.
	allocate_bytes	Allocate raw aligned memory from the underlying resource.
	allocate_object	Allocates raw memory suitable for an object or an array.
	construct	Constructs an object in allocated storage.
	deallocate	Deallocate memory.
	deallocate_bytes	Free raw memory.
	deallocate_object	Frees raw memory
	delete_object	Destroys and deallocates an object.
	destroy	Destroys an object in allocated storage.
	new_object	Allocates and constructs an object.
	resource	The underlying MemoryResource.
	select_on_container_copy_construction	Create a new PolymorphicAllocator for use by a container's copy constructor.

9.2.4.1.5 Datatypes

The AUTOSAR Adaptive Platform for Applications defines several datatypes to be used by applications and the platform APIs (e.g., as parameter or return types). Some of them are derived from corresponding datatypes defined in the C++ standard library.

Name:	Result	
Technology:	Native interface	
Usage:	Public API	
Description:	This datatype wraps either a value or an error. It is used as the standard mechanism to return values from APIs that may fail.	
Operations:	Error	Access the contained error.
	Value	Access the contained value.
	operator*	Access the contained value.
	operator->	Access the contained value.

9.2.4.2 Provided interfaces

Core currently provides no interfaces to other Functional Clusters.

9.2.4.3 Required interfaces

Core currently requires no interfaces.

9.2.5 Operating System Interface

Name:	Operating System Interface
Short Name:	n/a
Category:	Runtime
Daemon-based:	No
Responsibilities:	The `Operating System Interface` provides services for implementing multi-threaded real-time embedded applications (`Adaptive Application`s) which make use of the POSIX PSE51 profile. That profile provides support to create `Thread`s that may be executed in parallel on modern multi-core processors and to control their properties such as stack memory or their scheduling. In addition, primitives for shared resource access are provided such as `Semaphore`s or memory locking. Asynchronous (real-time) signals and message passing enable inter-process communication. High resolution timers and clocks are provided to control real-time behavior precisely. Some input/output functions are provided as well but no file system APIs. POSIX PSE51 and the `Operating System Interface` do not provide any means to execute and control `Process`es. `Process`es (of the AUTOSAR Adaptive Platform) are entirely controlled by `Execution Management` via interfaces that are implementation specific. Note that a typical AUTOSAR Adaptive Platform stack will not provide an actual implementation of the `Operating System Interface` because all functionality is already provided by standard libraries of the programming language (e.g. Standard C++ Library).

9.2.5.1 Defined interfaces

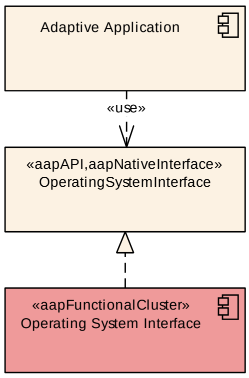

Figure 9.21: Interfaces defined by Operating System Interface

Name:	OperatingSystemInterface
Technology:	Native interface
Usage:	Public API
Description:	This interface represents the POSIX PSE51 profile, which is the API for `Adaptive Application`s. This interface is not detailed in this document.

9.2.5.2 Provided interfaces

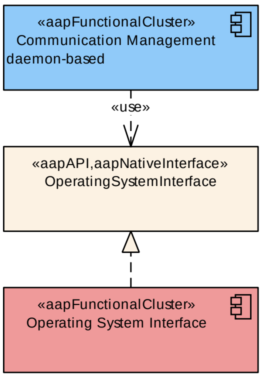

Figure 9.22: Users of the OperatingSystemInterface interfaces

Interface	*Requiring functional clusters*
OperatingSystemInterface	Communication Management

Table 9.8: Interfaces provided by Operating System Interface to other Functional Clusters

9.2.5.3 Required interfaces

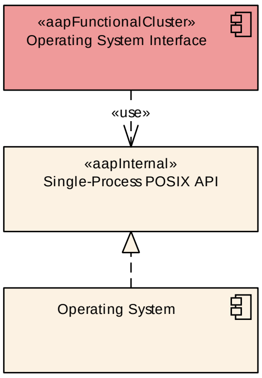

Figure 9.23: Interfaces required by Operating System Interface

Interface	*Purpose*
Single-Process POSIX API	The `Operating System Interface` uses services of the POSIX PSE51 profile, e.g. `Thread`s.

Table 9.9: Interfaces required by Operating System Interface

9.3 Communication

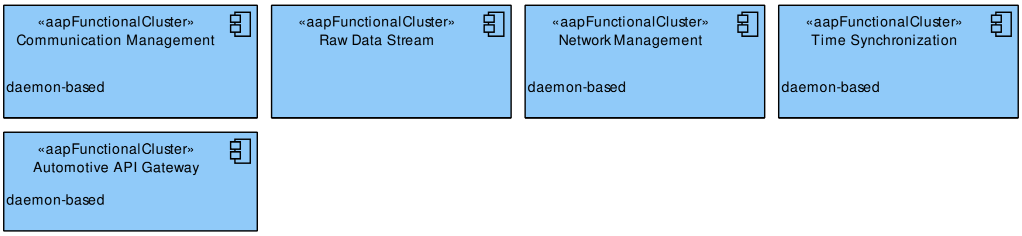

Figure 9.24: Functional Clusters in category Communication

9.3.1 Communication Management

Name:	Communication Management
Short Name:	com
Category:	Communication
Daemon-based:	Yes
Responsibilities:	`Communication Management` is responsible for all levels of service-oriented communication between applications in a distributed real-time embedded environment. That is, intra-process communication, inter-process communication and inter-machine communication. The latter is also possible with AUTOSAR Classic Platforms and third-party platforms. Communication paths can be established at design-, start-up-, and run-time. `Communication Management` consists of a generic part that handles brokering and configuration as well as generated skeletons for service providers and respective proxies for service consumers.

9.3.1.1 Defined interfaces

The interfaces of `Communication Management` are categorized into interfaces for `SecOC` (see Section 9.3.1.1.2), and interfaces freshness value management (see Section 9.3.1.1.3). Please note that a implementation of `Communication Management` will generate additional interfaces for each modeled Service Interface, e.g. a `Proxy` and a `Skeleton` interface. However, these generated interfaces are currently not covered in this document.

9.3.1.1.1 Generated interfaces for Services

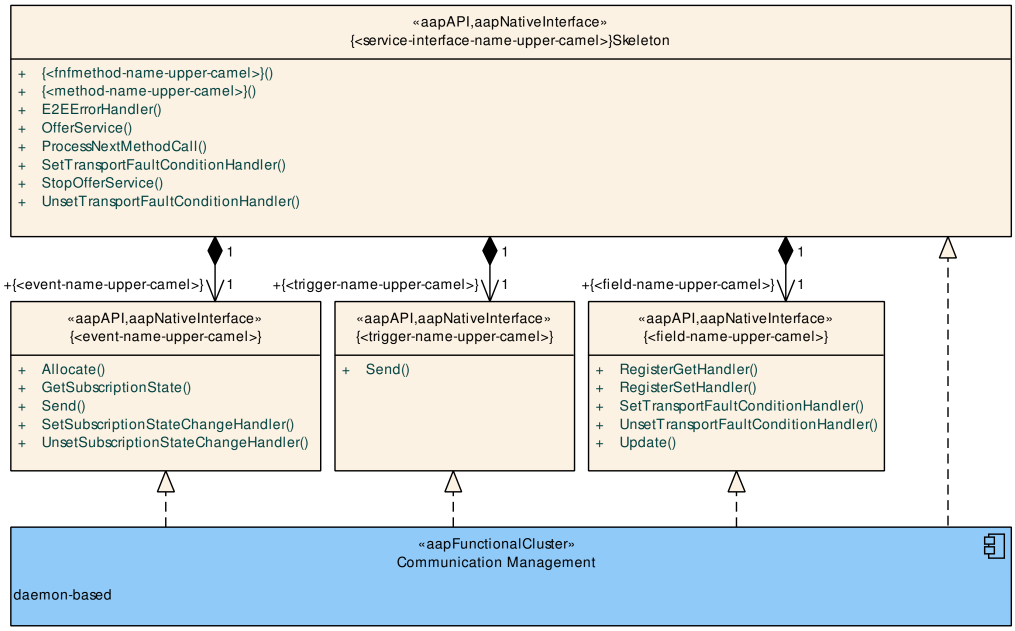

Figure 9.25: Generated Skeleton interfaces

Name:	{<service-interface-name-upper-camel>}Skeleton	
Technology:	Native interface	
Usage:	Public API	
Description:	Generated service skeleton class.	
Operations:	*E2EErrorHandler*	E2E Error Handler function called by ara::com if the E2E check of the request message failed.
	OfferService	Offer a service.
	ProcessNextMethodCall	Allows the service to trigger the execution of the next service consumer method call at a specific point of time.
	SetTransportFaultConditionHandler	Provides possibility to register a `ara::com::e2e::TransportFaultConditionHandler` for all methods in a given Skeleton.
	StopOfferService	Stop offering a service.
	UnsetTransportFaultConditionHandler	Provides the possibility to unregister a previously set `ara::com::e2e::TransportFaultConditionHandler` for all methods in a given Skeleton.
	{<fnfmethod-name-upper-camel>}	Generated fire-and-forget service method.
	{<method-name-upper-camel>}	Generated service method.

Name:	{<event-name-upper-camel>}
Technology:	Native interface
Usage:	Public API
Description:	Generated service Skeleton event class, one for each `VariableDataPrototype` defined in the `ServiceInterface` in the role `event`.

▽

△

Operations:	Allocate	Allocating data for event transfer when Communication Management is responsible for the data.
	GetSubscriptionState	Query `ara::com::SubscriptionState` of an event.
	Send	Send an event.
	SetSubscriptionStateChangeHandler	Set/Register `ara::com::SubscriptionState` change handler.
	UnsetSubscriptionStateChangeHandler	Unset/Deregister `ara::com::SubscriptionState` change handler.

Name:	{<field-name-upper-camel>}	
Technology:	Native interface	
Usage:	Public API	
Description:	Generated service skeleton Field class, one for each `Field` defined in the `ServiceInterface` in the role `field`.	
Operations:	RegisterGetHandler	Register a callback function to be invoked when a Get request is received.
	RegisterSetHandler	Register a callback function to be invoked when a Set request is received.
	SetTransportFaultConditionHandler	Provides possibility to register a `ara::com::e2e::TransportFaultConditionHandler` for a given Skeleton Field Get/Set.
	UnsetTransportFaultConditionHandler	Provides the possibility to unregister a previously set `ara::com::e2e::TransportFaultConditionHandler` for a given Skeleton Field Get/Set.
	Update	Initiate the transmission of updated field data to the subscribers.

Name:	{<trigger-name-upper-camel>}	
Technology:	Native interface	
Usage:	Public API	
Description:	Generated service skeleton Trigger class, one for each `Trigger` defined in the `ServiceInterface` in the role `trigger`.	
Operations:	Send	Send trigger to all subscribing applications.

9.3.1.1.2 Interfaces for SecOC

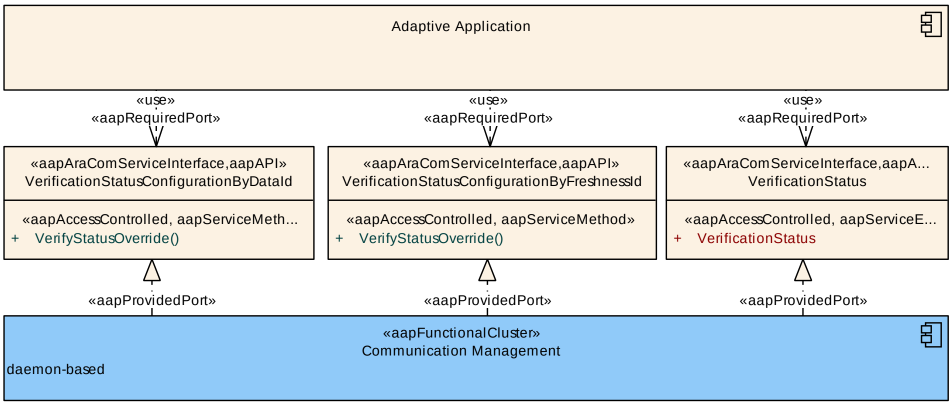

Figure 9.26: Interfaces for SecOC

Name:	VerificationStatus	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface provides an event to informed about the verification status of messages.	
Events:	VerificationStatus	This event is fired for each verification and holds the verification status.

Name:	VerificationStatusConfigurationByDataId	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface provides functionality to control the verification status of messages.	
Operations:	VerifyStatusOverride	This service method provides the ability to force to accept or to drop a message with or without performing the verification of authenticator or independent of the authenticator verification result, and to force a specific result allowing additional fault handling in the application.

Name:	VerificationStatusConfigurationByFreshnessId	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface provides functionality to control the verification status of messages.	
Operations:	VerifyStatusOverride	This service method provides the ability to force to accept or to drop a message with or without performing the verification of authenticator or independent of the authenticator verification result, and to force a specific result allowing additional fault handling in the application.

9.3.1.1.3 Interfaces for freshness value management

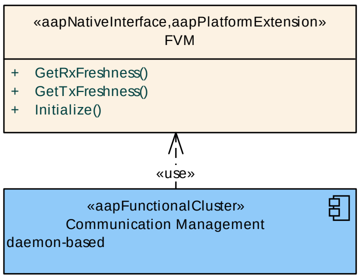

Figure 9.27: Interfaces for freshness value management

Name:	FVM	
Technology:	Native interface	
Usage:	Platform extension	
Description:	This interface provides functionality for freshness value management.	
Operations:	GetRxFreshness	Obtain the current freshness value for received messages.
	GetTxFreshness	Obtain the current freshness value for transmitted messages.
	Initialize	Initializes freshness value manager plugin implementation.

9.3.1.2 Provided interfaces

`Communication Management` currently provides no interfaces to other `Functional Cluster`s.

9.3.1.3 Required interfaces

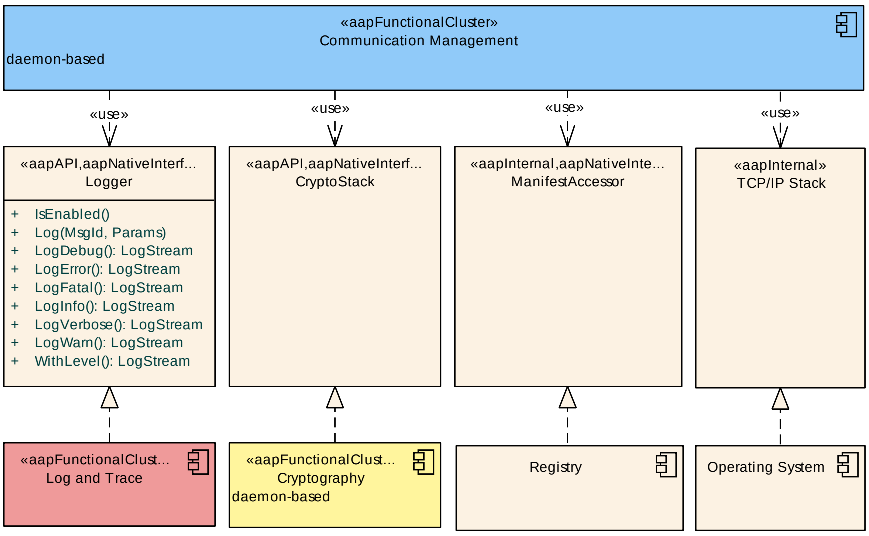

Figure 9.28: Interfaces required by Communication Management

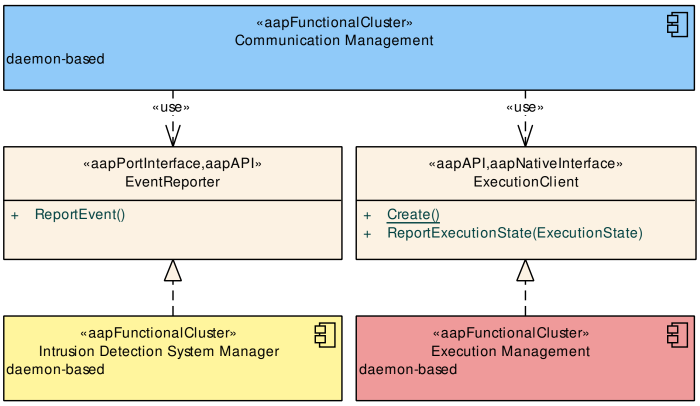

Figure 9.29: Interfaces required by Communication Management

Interface	***Purpose***
ManifestAccessor	Communication Management shall use this interface to read service information from the Manifests.
TCP/IP Stack	Communication Management shall use this interface to establish and control TCP/IP-based network connections.
Communication Management::FVM	Communication Management shall use this interface to get freshness values.
Cryptography::CryptoStack	This interface may be used e.g., to establish encrypted connections and generate / verify checksums (MAC).
Execution Management::ExecutionClient	
Intrusion Detection System Manager::Event Reporter	Communication Management may use this interface to report security events.
Log and Trace::Logger	Communication Management shall use this interface to log standardized messages.
Operating System Interface::OperatingSystem Interface	Communication Management should use this interface to create and control Threads used by the implementation.

Table 9.10: Interfaces required by Communication Management

9.3.2 Raw Data Stream

Name:	Raw Data Stream
Short Name:	rds
Category:	Communication
Daemon-based:	No
Responsibilities:	Raw Data Stream is responsible for raw communication between applications in a distributed real-time embedded environment.

9.3.2.1 Defined interfaces

9.3.2.1.1 Raw Data Streaming using IP based protocols (network layer)

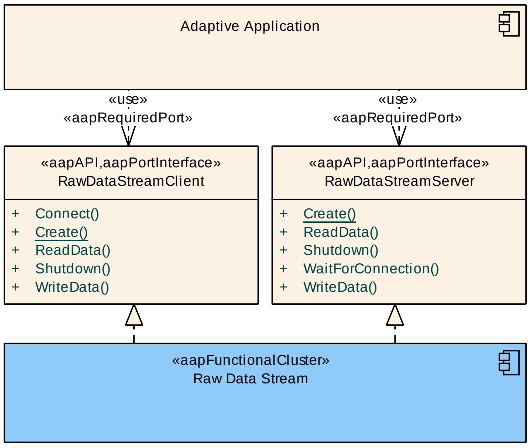

Figure 9.30: Interfaces for raw data streams using IP based protocols

Name:	RawDataStreamClient	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`RawDataStreamClientInterface`	
Usage:	Public API	
Description:	This interface provides functionality for a client reading and writing binary data streams over a IP network connection.	
Operations:	Connect	Sets up a socket connection for the raw data stream defined by the instance, and establish a connection to the TCP server.
	Create	Constructs a new `RawDataStreamClient`.
	ReadData	Requests to read a number of bytes of data from the socket connection for the raw data stream defined by the instance.
	Shutdown	Closes the socket connection for the raw data stream defined by the instance. Both, the receiving and the sending part of the socket connection shall be shut down.
	WriteData	Requests to write of a number of bytes to the socket connection for the raw data stream defined by the instance.

Name:	RawDataStreamServer	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`RawDataStreamServerInterface`	
Usage:	Public API	
Description:	This interface provides functionality for a server reading and writing binary data streams over a IP network connection.	
Operations:	Create	Constructs a new `RawDataStreamServer`.

▽

△

	ReadData	Requests to read a number of bytes of data from the socket connection for the raw data stream defined by the instance.
	Shutdown	Closes the socket connection for the raw data stream defined by the instance. Both the receiving and the sending part of the socket connection shall be shut down.
	WaitForConnection	Initializes the socket and prepare the `RawDataStream Server` instance for incoming connections.
	WriteData	Requests to write of a number of bytes to the socket connection for the raw data stream defined by the instance.

9.3.2.1.2 Raw Data Streaming using IEEE1722 protocol (data link layer)

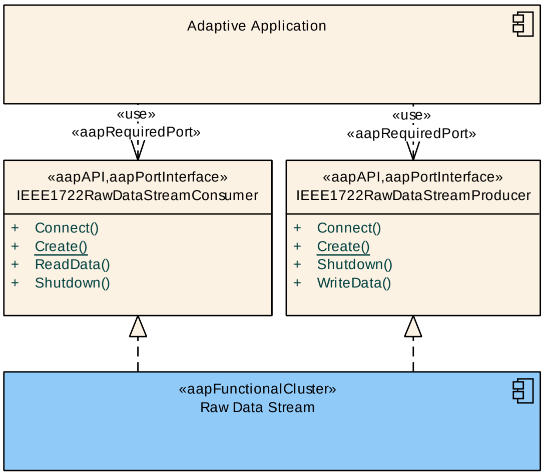

Figure 9.31: Interfaces for raw data streams using the IEEE1722 protocol

Name:	IEEE1722RawDataStreamConsumer	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`IEEE1722RawDataStreamConsumerInterface`	
Usage:	Public API	
Description:	Provides functionality to consume (read) data via an IEEE1722 stream from a network connection.	
Operations:	Connect	Sets up a data link socket connection and establishes internal connection.
	Create	Constructs a new `IEEE1722RawDataStreamConsumer`.
	ReadData	Requests to read IEEE1722 datagrams from the stream.
	Shutdown	Shuts down the data link socket connection.

Name:	IEEE1722RawDataStreamProducer	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`IEEE1722RawDataStreamProducerInterface`	
Usage:	Public API	
Description:	Provides functionality to produce (write) data via an IEEE1722 stream to a network connection.	
Operations:	Connect	Sets up a data link socket connection and establishes internal connection.
	Create	Constructs a new `IEEE1722RawDataStreamProducer`.
	Shutdown	Shuts down the data link socket connection.
	WriteData	Requests to write IEEE1722 datagrams to the stream.

9.3.2.2 Provided interfaces

`Raw Data Stream` currently provides no interfaces to other `Functional Clusters`.

9.3.2.3 Required interfaces

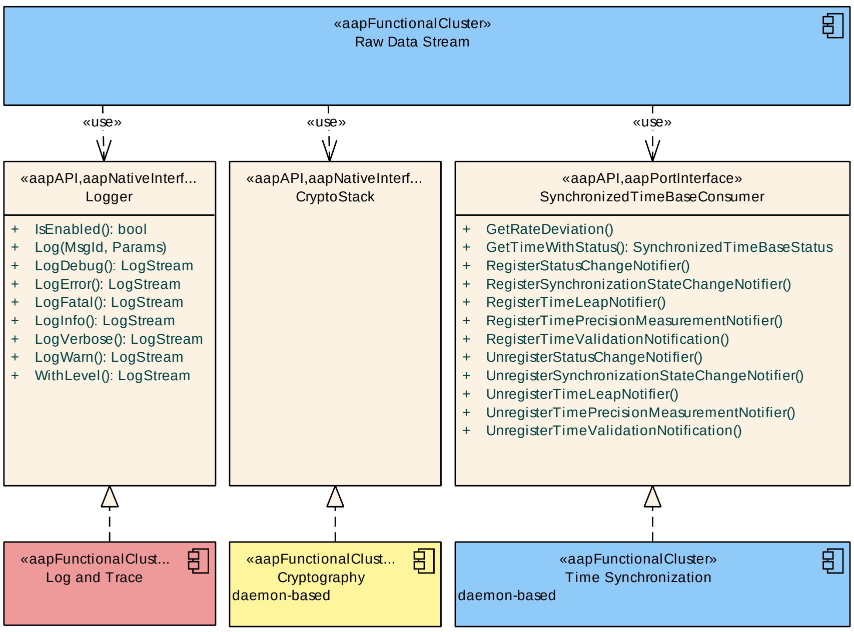

Figure 9.32: Interfaces required by Raw Data Stream

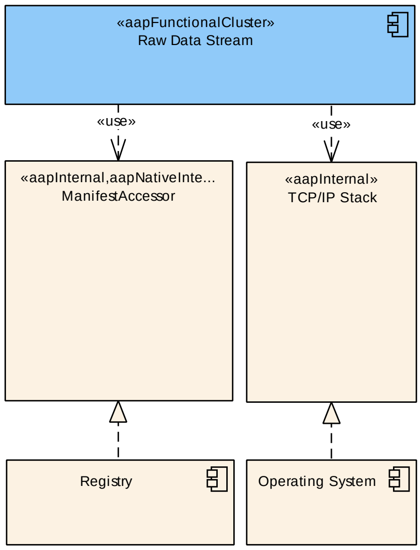

Figure 9.33: Internal interfaces required by Raw Data Stream

Interface	***Purpose***
ManifestAccessor	`Raw Data Stream` uses this interface to read information from the `Manifest`.
TCP/IP Stack	`Raw Data Stream` uses this interface to establish and control TCP/IP-based network connections and to establish data link layer socket connections.
Cryptography::CryptoStack	This interface may be used to establish encrypted connections.
Log and Trace::Logger	`Raw Data Stream` uses this interface to log standardized messages.
Time Synchronization::SynchronizedTimeBase Consumer	This interface is used to determine the current synchronized global time for IEEE1722-based communication.

Table 9.11: Interfaces required by Raw Data Stream

9.3.3 Network Management

Name:	Network Management
Short Name:	nm
Category:	Communication
Daemon-based:	Yes
Responsibilities:	`Network Management` provides functionality to request and query the network states for logical network handles, Such network handles can be mapped to physical or partial networks.

9.3.3.1 Defined interfaces

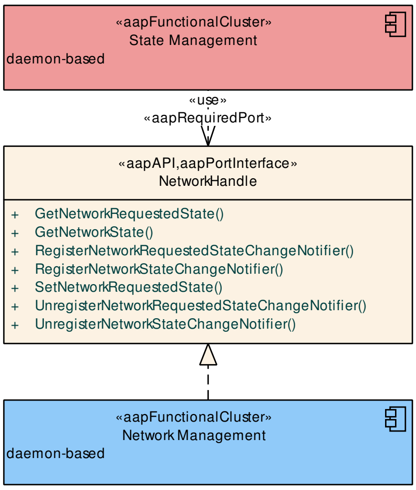

Figure 9.34: Interfaces for time base providers

Name:	NetworkHandle	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	NetworkManagementPortInterface	
Usage:	Public API	
Description:	This interface provides information about network status of a NetworkHandle. This interface is intended to be used by State Management only.	
Operations:	GetNetworkRequestedState	Obtain the current network requested state, i.e. if the PNC / VLAN / Physical Network is currently requested or released.
	GetNetworkState	Obtain the current network state, i.e. if the PNC / VLAN / Physical Network is currently active or not.
	RegisterNetworkRequestedStateChangeNotifier	Register a notifier function which is called if the current network requested state is changed (i.e. changed into FullCom or into NoCom). A maximum of one notifier can be registered. Every further registration overwrites the current registration.
	RegisterNetworkStateChangeNotifier	Register a notifier function which is called if the current network state is changed (i.e. changed into FullCom or into NoCom).
	SetNetworkRequestedState	Set a new network requested state. Setting a new network requested state will request or release the PNC / VLAN / Physical Network.
	UnregisterNetworkRequestedStateChangeNotifier	Unregister a notifier function which is called if a current network requested state is changed.
	UnregisterNetworkStateChangeNotifier	Unregister a notifier function which is called if a current network state is changed.

9.3.3.2 Provided interfaces

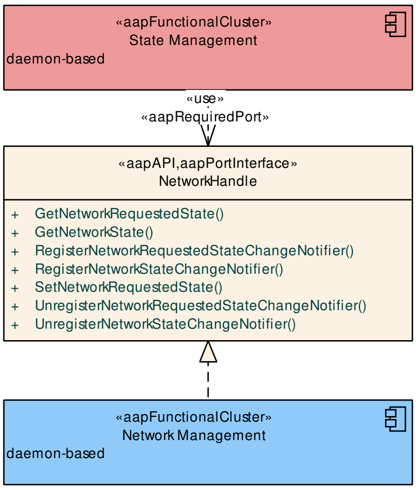

Figure 9.35: Users of Network Management interfaces

Interface	*Requiring functional clusters*
NetworkHandle	State Management

Table 9.12: Interfaces provided by Network Management to other Functional Clusters

9.3.3.3 Required interfaces

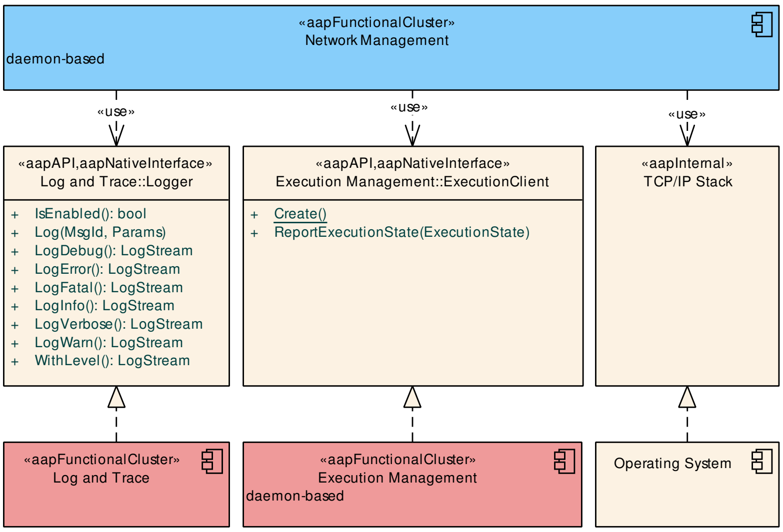

Figure 9.36: Interfaces required by Network Management

Interface	Purpose
ManifestAccessor	Network Management shall use this interface to read information about NmNetworkHandles from the Manifests.
TCP/IP Stack	Network Management should use this interface to set and to determine the status of IP-based networks.
Execution Management::ExecutionClient	
Log and Trace::Logger	Network Management shall use this interface to log standardized messages.

Table 9.13: Interfaces required by Network Management

9.3.4 Time Synchronization

Name:	Time Synchronization
Short Name:	tsync
Category:	Communication
Daemon-based:	Yes
Responsibilities:	Time Synchronization provides synchronized time information in distributed applications. Synchronized time information between different applications and/or Machines is of paramount importance when the correlation of different events across a distributed system is needed, either to be able to track such events in time or to trigger them at an accurate point in time.

9.3.4.1 Defined interfaces

The interfaces of Time Synchronization are categorized into interfaces for providing time information (see Section 9.3.4.1.1), interfaces for consuming time information (see Section 9.3.4.1.2), and interfaces for freshness value management (see Section 9.3.4.1.3).

9.3.4.1.1 Interfaces for time base providers

Time Synchronization defines the SynchronizedTimeBaseProvider interface to provide time information for a synchronized time base.

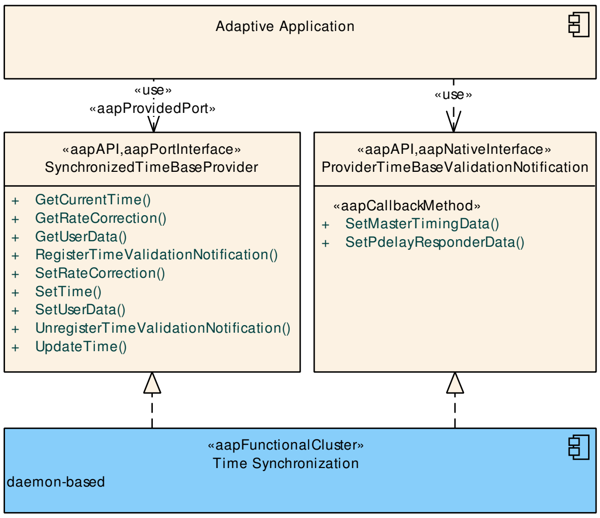

Figure 9.37: Interfaces for time base providers

Name:	SynchronizedTimeBaseProvider	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`SynchronizedTimeBaseProviderInterface`	
Usage:	Public API	
Description:	Provides access to a synchronized time base. It allows to get the current time point, the rate deviation, the current status and the received user data.	
Operations:	GetCurrentTime	Obtain the current time (regardless of the current sync status).
	GetRateCorrection	Obtain the current rate deviation of the clock.
	GetUserData	Get the user defined data of the time base.
	RegisterTimeValidationNotification	Used by time provider applications to receive time sync parameters. A maximum of one notifier can be registered. Every further registration overwrites the current registration.
	SetRateCorrection	Set the rate correction that will be applied to time values.
	SetTime	Set a new time value for the clock. Setting a new time also triggers transmission on the bus.
	SetUserData	Set the user data of the time base.
	UnregisterTimeValidationNotification	Used by time provider applications to receive time sync parameters.
	UpdateTime	Set a new time value for the clock. The clock value is only updated locally, transmission on the bus will happen in the next cycle.

Name:	ProviderTimeBaseValidationNotification	
Technology:	Native interface	
Usage:	Public API	
Description:	Callback interface to notify (Validator) Application about the availability of a new data block recorded for the Time Base.	
Operations:	*SetMasterTimingData*	Provide the recorded data block for the Time Master of the Time Base.

▽

△

	SetPdelayResponderData	Provide the recorded data block for the pPelay Responder of the Time Base.

9.3.4.1.2 Interfaces for time base consumers

Time Synchronization defines the SynchronizedTimeBaseConsumer interface to retrieve time information for a synchronized time base. SynchronizedTimeBaseStatus is used to determine the status of a synchronized time base.

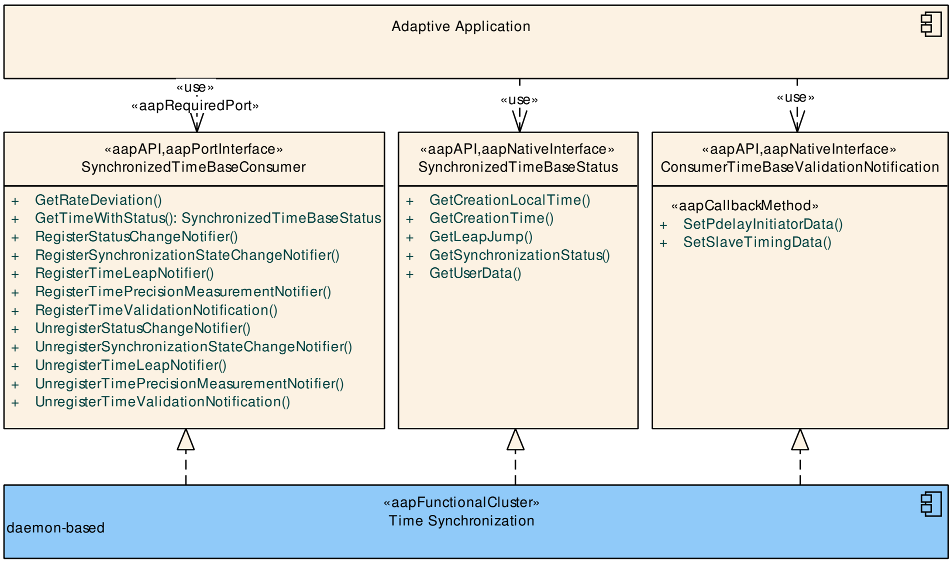

Figure 9.38: Interfaces for time base consumers

Name:	SynchronizedTimeBaseConsumer	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	SynchronizedTimeBaseConsumerInterface	
Usage:	Public API	
Description:	Provides access to the synchronized time base. It allows to get the current time point, the rate deviation, the current status and the received user data.	
Operations:	GetRateDeviation	Obtain the current rate deviation of the clock.
	GetTimeWithStatus	Obtain a snapshot of the current state of the clock. This includes status flags, clock configuration and the actual time value (local and synchronized) of the created status object.
	RegisterStatusChangeNotifier	Register a notifier function which is called if a status flag is changed (i.e. synchronization state, time leap or userdata). A maximum of one notifier can be registered. Every further registration overwrites the current registration.

▽

△

	RegisterSynchronizationStateChange Notifier	Register a notifier function which is called if a synchronization state is changed. A maximum of one notifier can be registered. Every further registration overwrites the current registration.
	RegisterTimeLeapNotifier	Register a notifier function which is called if a time leap happened. A maximum of one notifier can be registered. Every further registration overwrites the current registration.
	RegisterTimePrecisionMeasurement Notifier	Register a notifier function which is called if a new time precision snapshot is available. A maximum of one notifier can be registered. Every further registration overwrites the current registration. Time Synchronization will not do any queuing. If needed it has to be done within the notifier.
	RegisterTimeValidationNotification	Used by time consumer applications to receive time sync parameters. A maximum of one notifier can be registered. Every further registration overwrites the current registration.
	UnregisterStatusChangeNotifier	Un-register a notifier function which is called if a status flag is changed (i.e. synchronization state, time leap or userdata).
	UnregisterSynchronizationStateChange Notifier	Un-register a notifier function which is called if a synchronization state is changed.
	UnregisterTimeLeapNotifier	Un-register a notifier function which is called if a time leap happened.
	UnregisterTimePrecisionMeasurement Notifier	Un-register a notifier function which is called if a new time precision snapshot is available.
	UnregisterTimeValidationNotification	Un-register a notifier function for receiving time sync parameters.

Name:	SynchronizedTimeBaseStatus	
Technology:	Native interface	
Usage:	Public API	
Description:	Represents a snapshot of the current state of the clock (status flags, clock configuration, and the actual time value (local and synchronized)).	
Operations:	GetCreationLocalTime	Get the local time when this object was created.
	GetCreationTime	Get the creation time of this object.
	GetLeapJump	Determine the direction of a leap jump. Only the jump until the previous object creation is included.
	GetSynchronizationStatus	Returns the synchronization state when the object was created.
	GetUserData	Returns the user defined data of the time base.

Name:	ConsumerTimeBaseValidationNotification	
Technology:	Native interface	
Usage:	Public API	
Description:	Callback interface to notify Consumer Application about the availability of a new data block recorded for the Time Base.	
Operations:	*SetPdelayInitiatorData*	Provide the recorded data block for the pPelay Initiator of the Time Base.
	SetSlaveTimingData	Provide the recorded data block for the Time Slave of the Time Base.

9.3.4.1.3 Interfaces for freshness value management

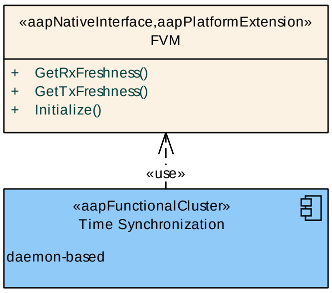

Figure 9.39: Interfaces for freshness value management

Name:	FVM	
Technology:	Native interface	
Usage:	Platform extension	
Description:	This interface provides functionality for freshness value management.	
Operations:	GetRxFreshness	Obtain the current freshness value for received messages.
	GetTxFreshness	Obtain the current freshness value for transmitted messages.
	Initialize	Initializes the freshness value plugin manager.

9.3.4.2 Provided interfaces

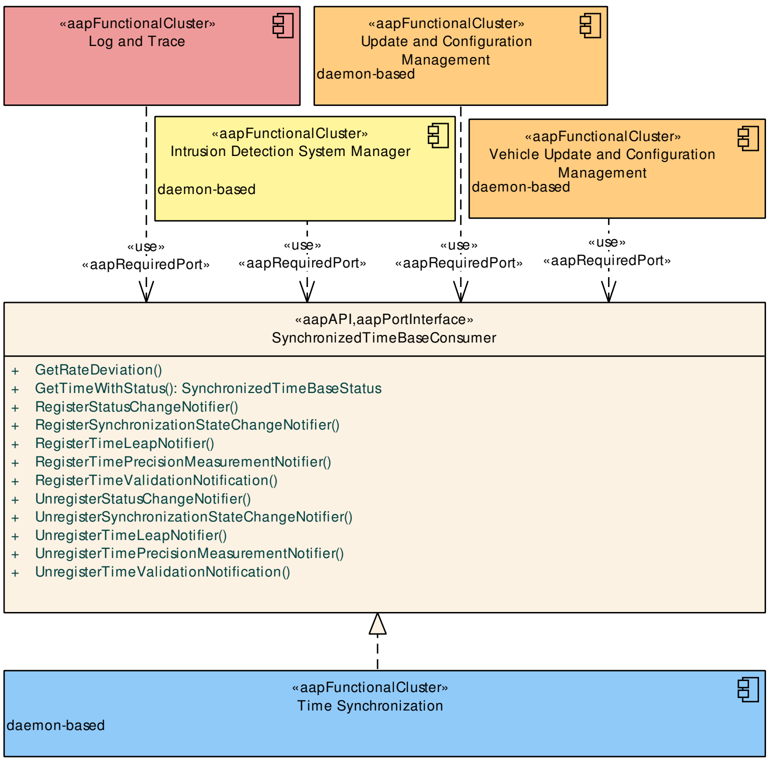

Figure 9.40: Users of Time Synchronization interfaces

Interface	*Requiring functional clusters*
SynchronizedTimeBaseConsumer	Intrusion Detection System Manager
	Log and Trace
	Raw Data Stream
	Update and Configuration Management
	Vehicle Update and Configuration Management

Table 9.14: Interfaces provided by Time Synchronization to other Functional Clusters

9.3.4.3 Required interfaces

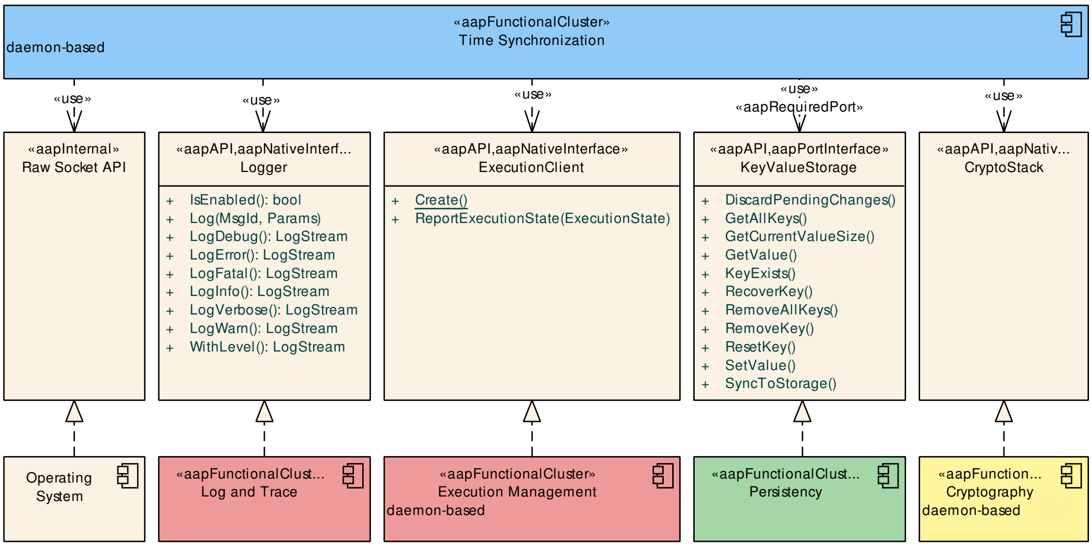

Figure 9.41: Interfaces required by Time Synchronization

Interface	*Purpose*
ManifestAccessor	`Time Synchronization` shall use this interface to read information about `TimeBaseResource`s as well as their providers and consumers from the `Manifest`s.
Raw Socket API	`Time Synchronization` should use this interface to send and receive raw ethernet packets as required by the time synchronization protocol.
Cryptography::CryptoStack	`Time Synchronization` shall use this interface to generate / verify checksums (MAC).
Execution Management::ExecutionClient	`Time Synchronization` shall use this interface to report the state of its daemon process.
Log and Trace::Logger	`Time Synchronization` shall use this interface to log standardized messages.
Persistency::KeyValueStorage	Used to store the last received timestamp to enable a faster startup.
Platform Health Management::SupervisedEntity	`Time Synchronization` should use this interface to enable supervision of its daemon process by `Platform Health Management`
Time Synchronization::FVM	`Time Synchronization` shall use this interface for freshness value management.

Table 9.15: Interfaces required by Time Synchronization

9.3.5 Automotive API Gateway

Name:	Automotive API Gateway
Short Name:	aag
Category:	Communication
Daemon-based:	Yes
Responsibilities:	The Automotive API Gateway is responsible for communication with the external world providing capabilities to get/set/subscribe to in-vehicle data using Vehicle Signal Specification and VISS as a communication protocol.

9.3.5.1 Defined interfaces

The Automotive API Gateway currently defines no interfaces.

9.3.5.2 Provided interfaces

Automotive API Gateway currently provides no interfaces to other Functional Clusters.

9.3.5.3 Required interfaces

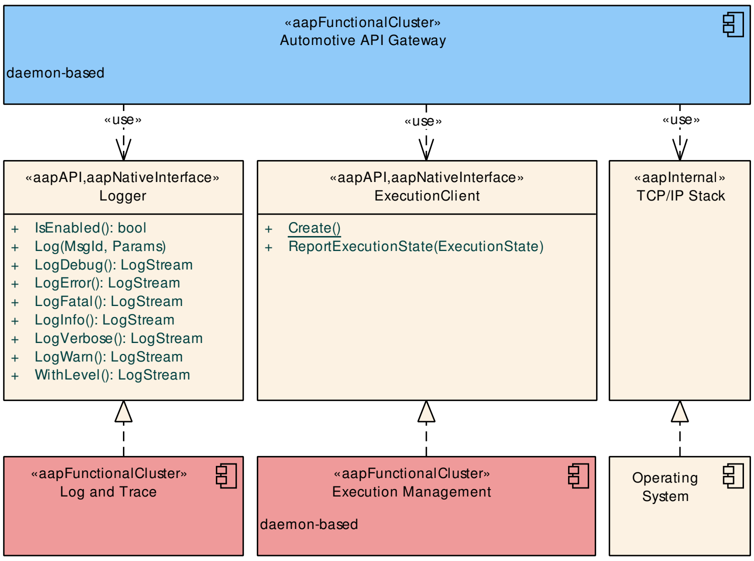

Figure 9.42: Interfaces required by the Automotive API Gateway

Interface	**Purpose**
TCP/IP Stack	The Automotive API Gateway shall use this interface to establish and control TCP/IP-based network connections.
Execution Management::ExecutionClient	
Log and Trace::Logger	The Automotive API Gateway shall use this interface to log messages.

Table 9.16: Interfaces required by Automotive API Gateway

9.4 Storage

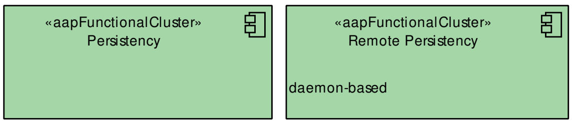

Figure 9.43: Functional Clusters in category Storage

9.4.1 Persistency

Name:	Persistency
Short Name:	per
Category:	Storage
Daemon-based:	No
Responsibilities:	Persistency provides functionality to store and retrieve information to/from non-volatile storage of a Machine. Persistent data is always private to one Process and is persisted across boot and ignition cycles. There is no mechanism available to share data between different Processes using Persistency to prevent a second path of data exchange besides Communication Management. However, Persistency supports concurrent access from multiple threads of the same application running in the context of the same Process. Persistency offers integrity of the stored data and provides error detection and correction schemes. Persistency also offers confidentiality of the stored data using encryption. Persistency supports updating of its storages using pre-defined values, or deleting files or keys during an update. Persistency provides statistics, for example, the used storage space.

9.4.1.1 Defined interfaces

The interfaces of Persistency are categorized into interfaces for file access (see Section 9.4.1.1.1), interfaces for a key-value-based data access (see Section 9.4.1.1.2) and interfaces for general management of persistent data (see Section 9.4.1.1.3).

9.4.1.1.1 Interfaces for file storage

Persistency provides read and write access to plain files by means of a FileStorage (cf. Figure 9.44). A FileStorage has to be opened using OpenFileStorage(). A FileStorage then provides access to several files using their name.

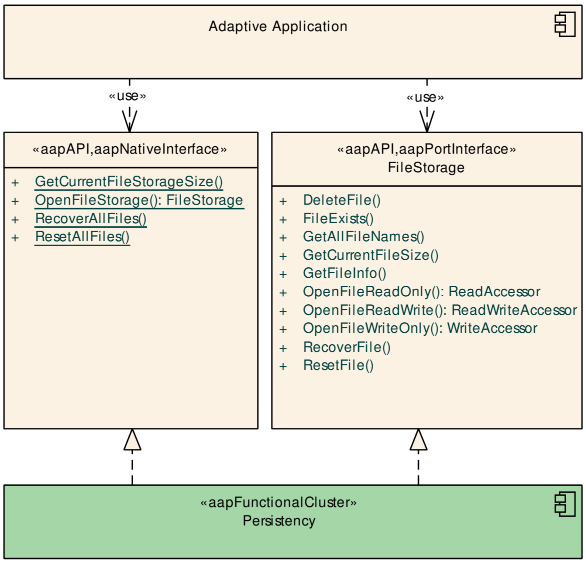

Figure 9.44: Interfaces for file storage

Name:	FileStorageOperations	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface provides functions to open and manage FileStorages.	
Operations:	GetCurrentFileStorageSize	Returns the space in bytes currently occupied by a FileStorage.
	OpenFileStorage	Opens a FileStorage.
	RecoverAllFiles	Recovers a FileStorage including all files.
	ResetAllFiles	Resets a FileStorage including all files.

Name:	FileStorage	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	PersistencyFileStorageInterface	
Usage:	Public API	
Description:	This interface provides functions to open and manage files.	
Operations:	DeleteFile	Deletes a file from this FileStorage.
	FileExists	Checks if a file exists in this FileStorage.
	GetAllFileNames	Returns a list of all currently available files of this FileStorage.
	GetCurrentFileSize	Returns the space in bytes currently occupied by the content of a file of this FileStorage.

▽

△

	GetFileInfo	Returns additional information on a file of this `FileStorage`.
	OpenFileReadOnly	Opens a file of this `FileStorage` for reading.
	OpenFileReadWrite	Opens a file of this `FileStorage` for reading and writing.
	OpenFileWriteOnly	Opens a file of this `FileStorage` for writing.
	RecoverFile	Recovers a file of this `FileStorage`.
	ResetFile	Resets a file of this `FileStorage` to its initial content.

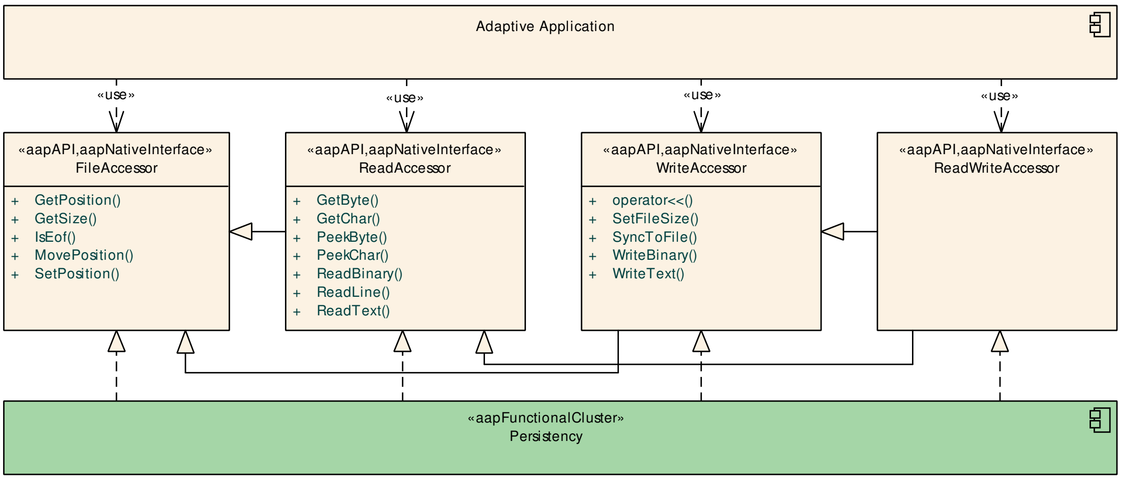

Figure 9.45: Interfaces for file access

Name:	FileAccessor	
Technology:	Native interface	
Usage:	Public API	
Description:	FileAccessor contains a common basis for ReadAccessor and WriteAccessor.	
Operations:	GetPosition	Returns the current position relative to the beginning of the file.
	GetSize	Returns the current size of a file in bytes.
	IsEof	Checks if the current position is at end of file.
	MovePosition	Moves the current position in the file relative to the origin.
	SetPosition	Sets the current position relative to the beginning of the file.

Name:	ReadAccessor	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface provides functions to read text and binary data from a file.	
Operations:	GetByte	Returns the byte at the current position of the file, advancing the current position.
	GetChar	Returns the character at the current position of the file, advancing the current position.

▽

△

	PeekByte	Returns the byte at the current position of the file.
	PeekChar	Returns the character at the current position of the file.
	ReadBinary	Reads all remaining bytes into a Vector of Byte, starting from the current position.
	ReadLine	Reads a complete line of characters into a String, advancing the current position accordingly.
	ReadText	Reads all remaining characters into a String, starting from the current position.

Name:	WriteAccessor	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface provides functions to write text and binary data to a file.	
Operations:	SetFileSize	Reduces the size of the file to 'size', effectively removing the current content of the file beyond this size.
	SyncToFile	Triggers flushing of the current file content to the physical storage.
	WriteBinary	Writes binary data to the file.
	WriteText	Writes a string to the file.
	operator<<	Writes a String to the file.

Name:	ReadWriteAccessor
Technology:	Native interface
Usage:	Public API
Description:	This interface provides functions to read and write text and binary data from / to a file.

9.4.1.1.2 Interfaces for key-value storage

`Persistency` provides read and write access to data structured as key-value pairs by means of the `KeyValueStorage` API (cf. Figure 9.46). A `KeyValueStorage` has to be created by calling `OpenKeyValueStorage()`. A `KeyValueStorage` then provides access to data stored for individual keys using the `GetValue()` and `SetValue()` operations.

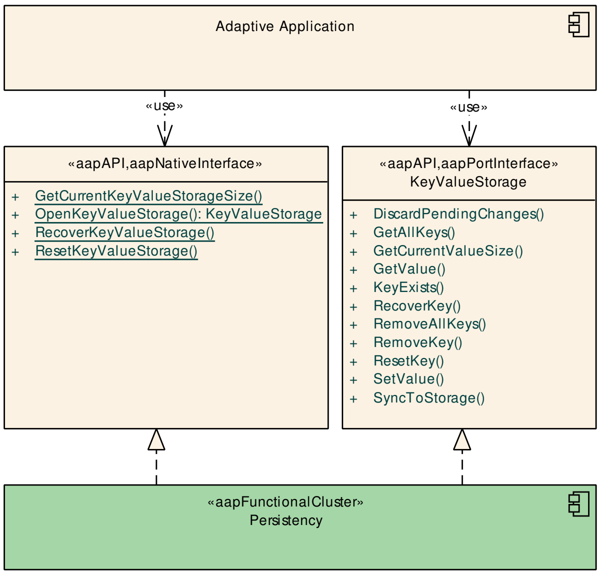

Figure 9.46: Interfaces for key-value-based data storage

Name:	KeyValueStorageOperations	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface provides functions to open and manage `KeyValueStorage`s.	
Operations:	GetCurrentKeyValueStorageSize	Returns the space in bytes currently occupied by a `KeyValueStorage`.
	OpenKeyValueStorage	Opens a `KeyValueStorage`.
	RecoverKeyValueStorage	Recovers a `KeyValueStorage`.
	ResetKeyValueStorage	Resets a `KeyValueStorage` to the initial state.

Name:	KeyValueStorage	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`PersistencyKeyValueStorageInterface`	
Usage:	Public API	
Description:	This interface provides functions to access values associated with keys.	
Operations:	DiscardPendingChanges	Discards changed key-value pairs of the `KeyValueStorage` an re-reads them from the physical storage.
	GetAllKeys	Returns a list of all currently available keys of this `KeyValueStorage`.
	GetCurrentValueSize	Returns the size (in bytes) of the value assigned to a key of this `KeyValueStorage`.
	GetValue	Returns the value assigned to a key of this `KeyValueStorage`.
	KeyExists	Checks if a key exists in this `KeyValueStorage`.
	RecoverKey	Recovers a single key of this `KeyValueStorage`.
	RemoveAllKeys	Removes all keys and associated values from this `KeyValueStorage`.

▽

	RemoveKey	Removes a key and the associated value from this KeyValueStorage.
	ResetKey	Resets a key of this KeyValueStorage to its initial value.
	SetValue	Stores a key in this KeyValueStorage.
	SyncToStorage	Triggers flushing of changed key-value pairs of the KeyValueStorage to the physical storage.

9.4.1.1.3 Interfaces for general persistency handling

Persistency provides operations for handling and recovery of persistent data of a Process (cf. Figure 9.47).

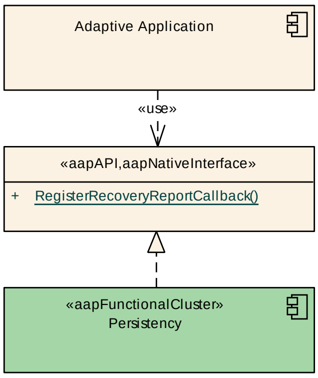

Figure 9.47: Interfaces for general persistency handling

Name:	PersistencyHandlingOperations	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface provides operations to manage persistent data.	
Operations:	RegisterRecoveryReportCallback	Register a recovery reporting callback with Persistency.

9.4.1.1.4 Installation and Update of Persistent Data

Persistency provides functionality to manage (install/update or reset to the initial state) KeyValueStorages and FileStorages during the installation/update process of applications and to cleanup any unused backup data afterwards (cf. Figure 9.48).

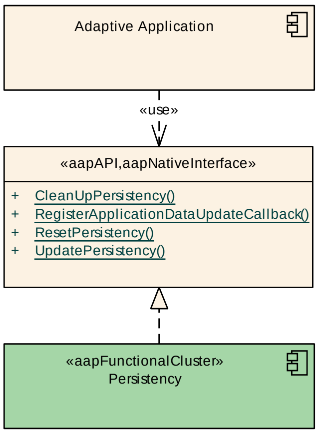

Figure 9.48: Interfaces for installation and update of persistent data

Name:	InstallationAndUpdateOperations	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface provides functions to manage KeyValueStorages and FileStorages during the installation/update process of applications.	
Operations:	CleanUpPersistency	Removes backup data and other unused data of all KeyValueStorages and FileStorages.
	RegisterApplicationDataUpdateCallback	Registers an application data update callback with Persistency.
	ResetPersistency	Resets all FileStorages and KeyValueStorages by entirely removing their content.
	UpdatePersistency	Updates all FileStorages and KeyValueStorages after a new manifest was installed.

9.4.1.2 Provided interfaces

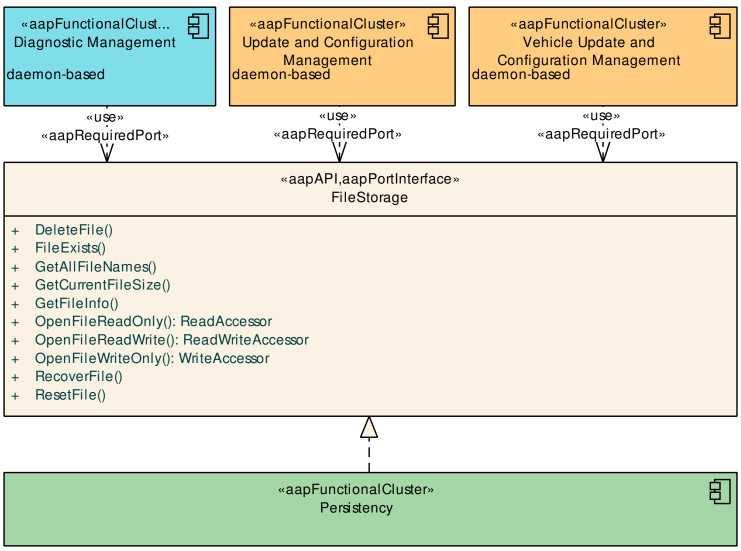

Figure 9.49: Users of the FileStorage interface

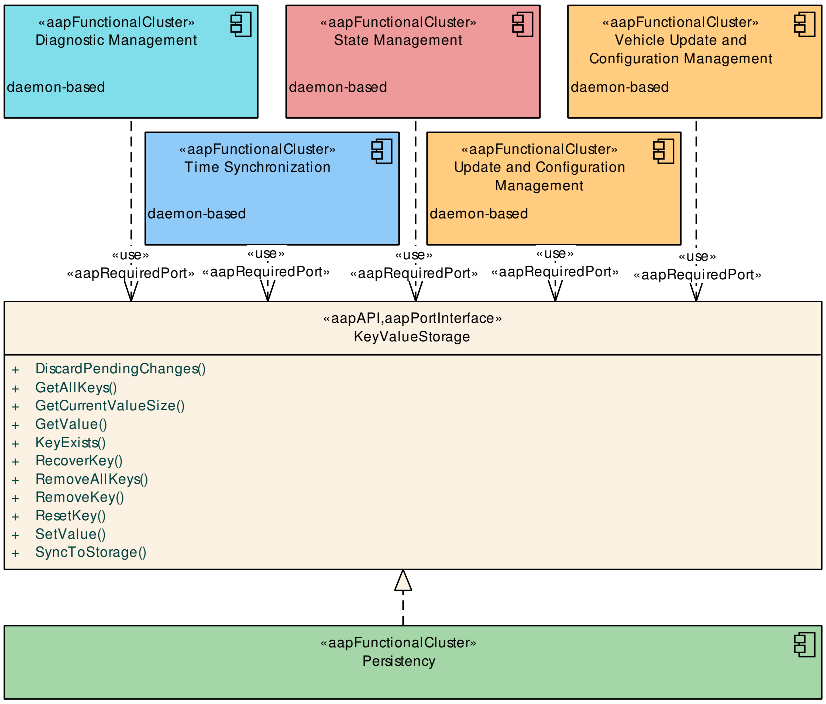

Figure 9.50: Users of the KeyValueStorage interface

Interface	***Requiring functional clusters***
KeyValueStorageOperations	Remote Persistency
FileStorage	Diagnostic Management
	Update and Configuration Management
	Vehicle Update and Configuration Management
KeyValueStorage	Diagnostic Management
	Remote Persistency
	State Management
	Time Synchronization
	Update and Configuration Management
	Vehicle Update and Configuration Management

Table 9.17: Interfaces provided by Persistency to other Functional Clusters

9.4.1.3 Required interfaces

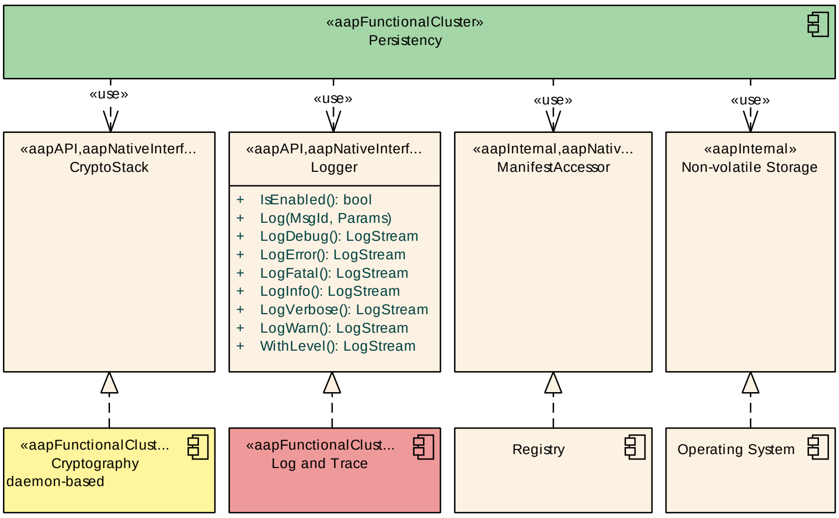

Figure 9.51: Interfaces required by Persistency

Interface	***Purpose***
ManifestAccessor	Used to read configuration information from the `Manifest`s.
Non-volatile Storage	Used to access non-volatile storage provided by the underlying operating system, for example, a file system.
Cryptography::CryptoStack	Used to ensure confidentiality and integrity of the persisted data.
Log and Trace::Logger	Used to provide information about errors and internal states and actions of `Persistency`.

Table 9.18: Interfaces required by Persistency

9.4.2 Remote Persistency

Name:	Remote Persistency
Short Name:	rper
Category:	Storage
Daemon-based:	Yes
Responsibilities:	`Remote Persistency` integrates `KeyValueStorage`s and service interfaces to provide persistent storage of data items.

9.4.2.1 Defined interfaces

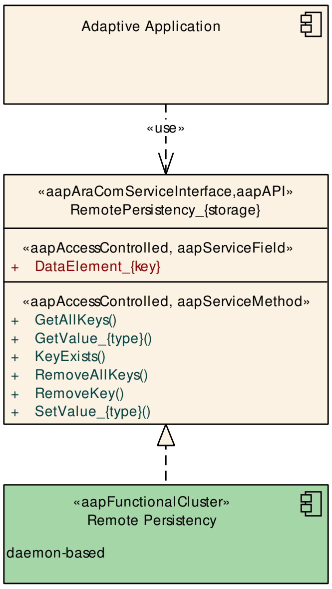

Figure 9.52: Interface for accessing KeyValueStorage

Name:	RemotePersistency_{storage}	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	Interface for accessing the KeyValueStorage {storage}.	
Operations:	GetAllKeys	Get the names of all data items of this storage which have been configured in the Manifest or were introduced at runtime.
	GetValue_{type}	Read the value of a data item of type {type}.
	KeyExists	Check if a data item has been configured in the Manifest or was introduced at runtime.
	RemoveAllKeys	Remove all data items from the storage that were created at run-time or by deployment. Please note: Data items that are available as a field will not be removed.
	RemoveKey	Remove a data item from the storage that was created at run-time or by deployment. Please note: Data items that are available as a field cannot be removed.
	SetValue_{type}	Update the value of a data item of type {type}, or create it if it does not exist.
Fields:	DataElement_{key}	Field for accessing the configured Key-Value Pair {key}.

9.4.2.2 Provided interfaces

Remote Persistency currently provides no interfaces to other Functional Clusters.

9.4.2.3 Required interfaces

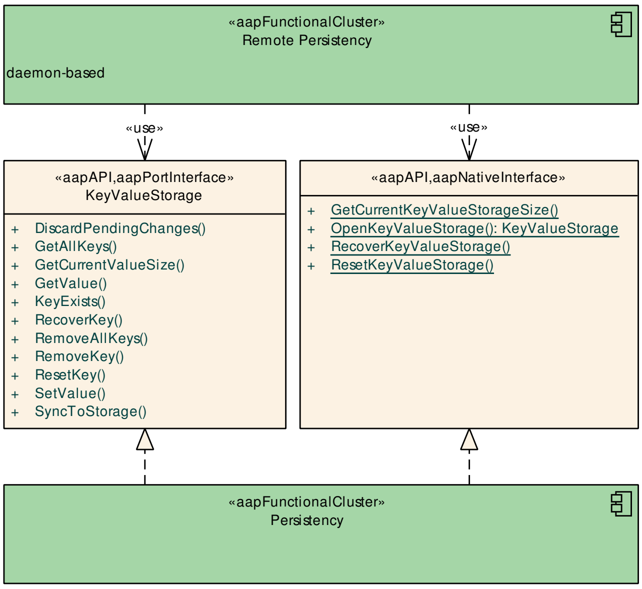

Figure 9.53: Interfaces required by Remote Persistency

Interface	*Purpose*
Persistency::KeyValueStorageOperations	Used to access an underlying `KeyValueStorage`.
Persistency::KeyValueStorage	Used to access an underlying `KeyValueStorage`.

Table 9.19: Interfaces required by Remote Persistency

9.5 Security

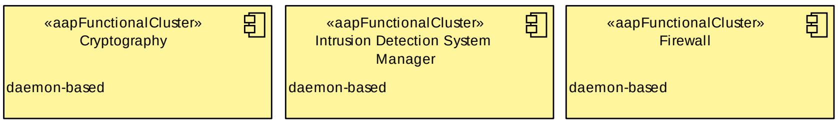

Figure 9.54: Functional Clusters in category Security

9.5.1 Cryptography

Name:	Cryptography
Short Name:	crypto
Category:	Security
Daemon-based:	Yes

▽

△

Responsibilities:	Cryptography provides various cryptographic routines to ensure confidentiality of data, to ensure integrity of data (e.g., using hashes), and auxiliary functions for example key management and random number generation. Cryptography is designed to support encapsulation of security-sensitive operations and decisions in a separate component, such as a Hardware Security Module (HSM). Additional protection of keys and key usage can be provided by constraining keys to particular usages (e.g., decrypt-only), or limiting the availability of keys to individual applications. Depending on application support, Cryptography can also be used to protect session keys and intermediate secrets when processing cryptographic protocols such as TLS and SecOC.

9.5.1.1 Defined interfaces

9.5.1.1.1 Common interfaces

The main entry point for using the Cryptography API are the factory functions LoadCryptoProvider() for using cryptographic routines, LoadKeyStorageProvider() for access to the key store, and LoadX509Provider() for X.509 certificate handling.

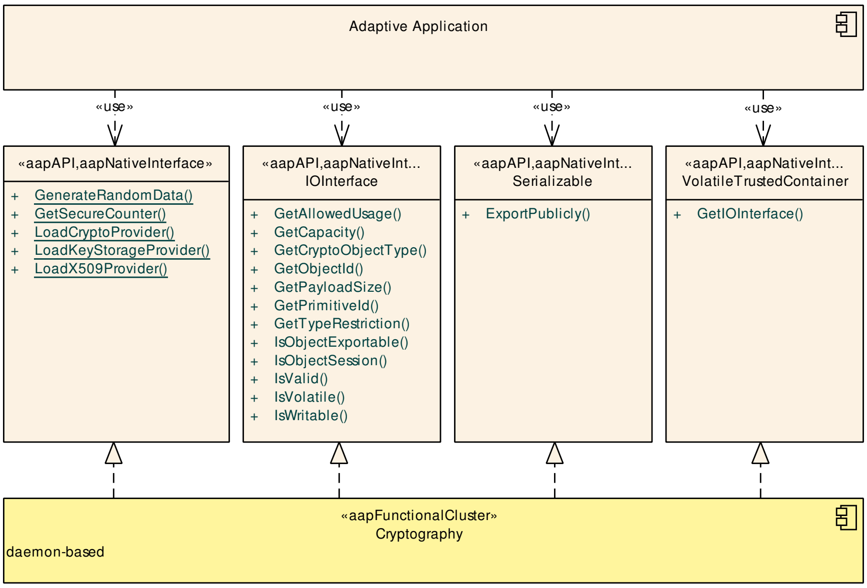

Figure 9.55: Common Interfaces of Cryptography

Name:	EntryPoint	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface provides the main entry points for using the Cryptography API.	
Operations:	GenerateRandomData	Generate random data of the requested size.

▽

△

	GetSecureCounter	Get current value of 128 bit SecureCounter supported by the Crypto Stack.
	LoadCryptoProvider	Factory that creates or returns existing single instance of a specific CryptoProvider.
	LoadKeyStorageProvider	Factory that creates or returns an existing single instance of the KeyStorageProvider.
	LoadX509Provider	Factory that creates or returns an existing single instance of the X509Provider.

Name:	IOInterface	
Technology:	Native interface	
Usage:	Public API	
Description:	Interface for saving and loading of security objects.	
Operations:	GetAllowedUsage	Return actual allowed key/seed usage flags defined by the key slot prototype for this "Actor" and current content of the container.
	GetCapacity	Return capacity of the underlying resource.
	GetCryptoObjectType	Return the type of the object referenced by this IOInterface.
	GetObjectId	Return an ID of an object stored to this IOInterface.
	GetPayloadSize	Return size of an object payload stored in the underlying buffer.
	GetPrimitiveId	Get vendor specific ID of the primitive.
	GetTypeRestriction	Return content type restriction.
	IsObjectExportable	Return the exportable attribute of an object stored to the container.
	IsObjectSession	Return the session (or temporary) attribute of an object as set.
	IsValid	Get whether the underlying KeySlot is valid.
	IsVolatile	Return volatility of the underlying buffer.
	IsWritable	Get whether the underlying KeySlot is writable.

Name:	Serializable	
Technology:	Native interface	
Usage:	Public API	
Description:	Serializable object interface.	
Operations:	ExportPublicly	Serialize itself publicly.

Name:	VolatileTrustedContainer	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface is used for buffering Cryptography API objects in RAM.	
Operations:	GetIOInterface	Retrieve the IOInterface used for importing/exporting objects into this container.

9.5.1.1.2 General cryptography interfaces

The CryptoProvider interface provides access to various cryptographic routines. Each of those routines is managed by specializations of the CryptoContext interface.

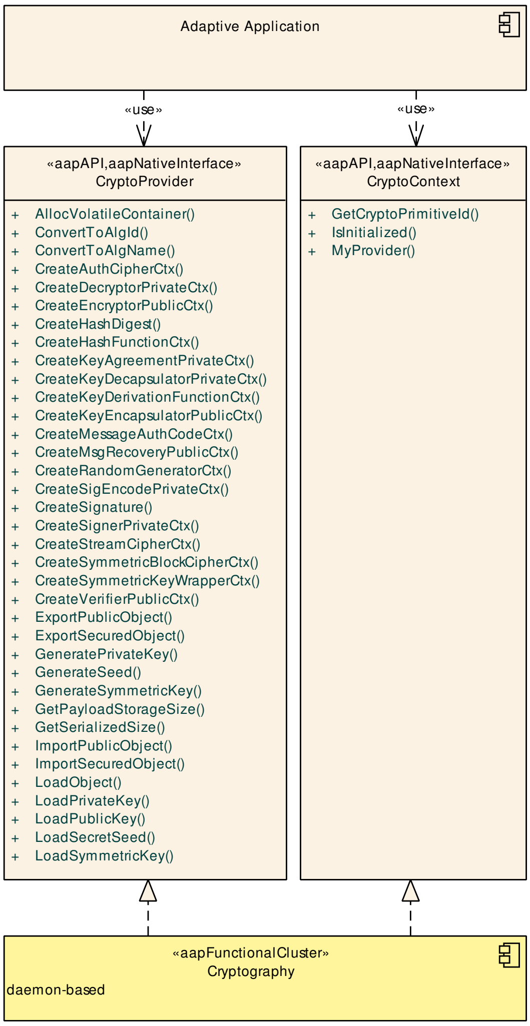

Figure 9.56: General Cryptography Interfaces

Name:	CryptoProvider	
Technology:	Native interface	
Usage:	Public API	
Description:	This is a "factory" interface of all supported crypto primitives and a "trusted environment" for internal communications between them.	
Operations:	AllocVolatileContainer	Allocate a VolatileTrustedContainer according to directly specified capacity.
	ConvertToAlgId	Convert a common name of crypto algorithm to a correspondent vendor specific binary algorithm ID.
	ConvertToAlgName	Convert a vendor specific binary algorithm ID to a correspondent common name of the crypto algorithm.

▽

△

	CreateAuthCipherCtx	Create a symmetric authenticated cipher context.
	CreateDecryptorPrivateCtx	Create a decryption private key context.
	CreateEncryptorPublicCtx	Create an encryption public key context.
	CreateHashDigest	Construct signature object from directly provided components of a hash digest.
	CreateHashFunctionCtx	Create a hash function context.
	CreateKeyAgreementPrivateCtx	Create a key-agreement private key context.
	CreateKeyDecapsulatorPrivateCtx	Create a key-decapsulator private key context of a key encapsulation mechanism.
	CreateKeyDerivationFunctionCtx	Create a key derivation function context.
	CreateKeyEncapsulatorPublicCtx	Create a key-encapsulator public key context of a key encapsulation mechanism.
	CreateMessageAuthCodeCtx	Create a symmetric message authentication code context.
	CreateMsgRecoveryPublicCtx	Create a message recovery public key context.
	CreateRandomGeneratorCtx	Create a random number generator context.
	CreateSigEncodePrivateCtx	Create a signature encoding private key context.
	CreateSignature	Construct a signature object from directly provided components of a digital signature/MAC or authenticated encryption (AE/AEAD).
	CreateSignerPrivateCtx	Create a signature private key context.
	CreateStreamCipherCtx	Create a symmetric stream cipher context.
	CreateSymmetricBlockCipherCtx	Create a symmetric block cipher context.
	CreateSymmetricKeyWrapperCtx	Create a symmetric key-wrap algorithm context.
	CreateVerifierPublicCtx	Create a signature verification public key context.
	ExportPublicObject	Export publicly an object.
	ExportSecuredObject	Export a crypto object in a secure manner.
	GeneratePrivateKey	Allocate a new private key context of correspondent type and generates the key value randomly.
	GenerateSeed	Generate a random Secret Seed object of requested algorithm.
	GenerateSymmetricKey	Allocate a new symmetric key object and fill it by a new randomly generated value.
	GetPayloadStorageSize	Return minimally required capacity of a key slot for saving of the object's payload.
	GetSerializedSize	Return required buffer size for serialization of an object in specific format.
	ImportPublicObject	Import publicly serialized object to a storage location.
	ImportSecuredObject	Import securely serialized object to the persistent or volatile storage.
	LoadObject	Load any crypto object from the `IOInterface` provided.
	LoadPrivateKey	Load a `PrivateKey` from the `IOInterface` provided.
	LoadPublicKey	Load a `PublicKey` from the `IOInterface` provided.
	LoadSecretSeed	Load a `SecretSeed` from the `IOInterface` provided.
	LoadSymmetricKey	Load a `SymmetricKey` from the `IOInterface` provided.

Name:	CryptoContext	
Technology:	Native interface	
Usage:	Public API	
Description:	A common interface of a mutable cryptographic context, i.e. that is not bound to a single crypto object.	
Operations:	GetCryptoPrimitiveId	Returns a `CryptoPrimitiveId` instance containing instance identification.
	IsInitialized	Check if the crypto context is already initialized and ready to use.
	MyProvider	Get a reference to the `CryptoProvider` of this context.

9.5.1.1.3 Cryptography context interfaces

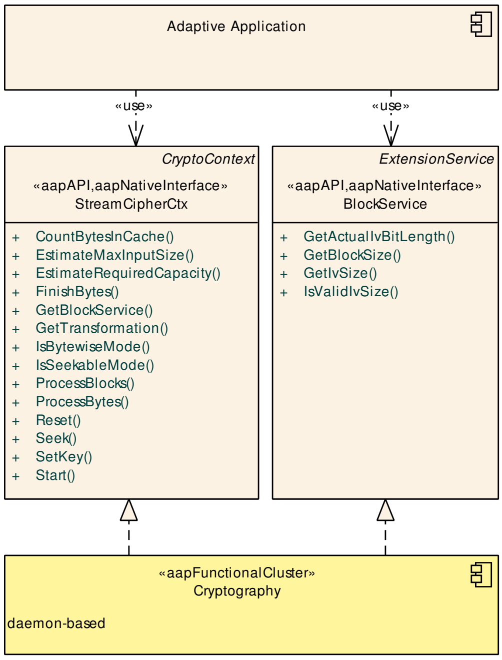

Figure 9.57: BlockService and CryptoContext Interfaces

Name:	BlockService	
Technology:	Native interface	
Usage:	Public API	
Description:	Extension meta-information service for block cipher contexts.	
Operations:	GetActualIvBitLength	Get the actual bit-length of an initialization vector loaded to the context.
	GetBlockSize	Get the block (or internal buffer) size of the base algorithm.
	GetIvSize	Get default expected size of the initialization vector or nonce.
	IsValidIvSize	Verify validity of specific initialization vector length.

Name:	StreamCipherCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	Generalized stream cipher context interface covering all modes of operation.	
Operations:	CountBytesInCache	Count number of bytes now kept in the context cache.
	EstimateMaxInputSize	Estimate maximal number of input bytes that may be processed for filling of an output buffer without overflow.
	EstimateRequiredCapacity	Estimate minimal required capacity of the output buffer, which is enough for saving a result of input data processing.
	FinishBytes	Process the final part of message (that may be not aligned to the block-size boundary).
	GetBlockService	Get the `BlockService` instance.
	GetTransformation	Get the kind of transformation configured for this context: Encrypt or Decrypt.
	IsBytewiseMode	Check the operation mode for the byte-wise property.
	IsSeekableMode	Check if the seek operation is supported in the current mode.
	ProcessBlocks	Process initial parts of message aligned to the block-size boundary.
	ProcessBytes	Process a non-final part of message (that is not aligned to the block-size boundary).
	Reset	Clear the crypto context.
	Seek	Set the position of the next byte within the stream of the encryption/decryption gamma.
	SetKey	Set (deploy) a key to the stream cihper algorithm context.
	Start	Initialize the context for a new data stream processing or generation (depending from the primitive).

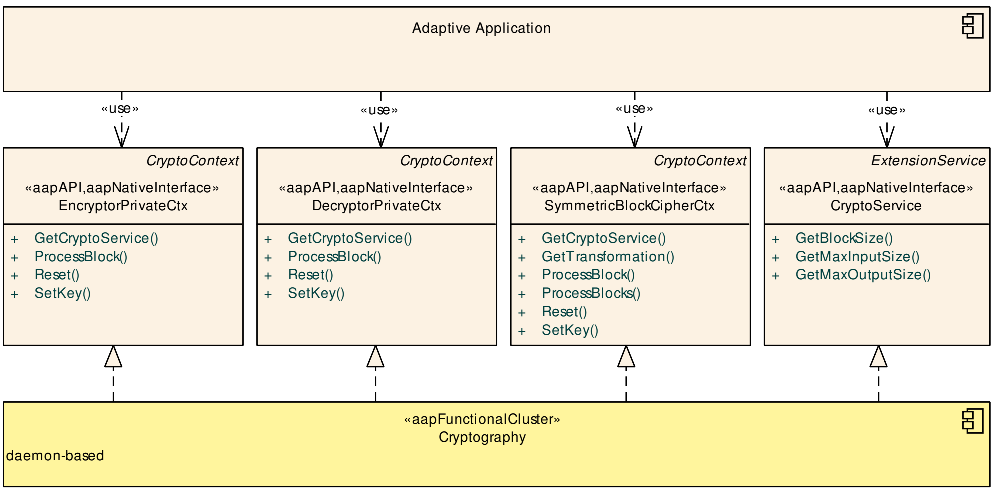

Figure 9.58: CryptoService and CryptoContext Interfaces

Name:	CryptoService	
Technology:	Native interface	
Usage:	Public API	
Description:	Extension meta-information service for cryptographic contexts.	
Operations:	GetBlockSize	Get block (or internal buffer) size of the base algorithm.
	GetMaxInputSize	Get maximum expected size of the input data block.
	GetMaxOutputSize	Get maximum possible size of the output data block.

Name:	EncryptorPrivateCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	Asymmetric decryption private key context interface.	
Operations:	GetCryptoService	Get the `CryptoService` instance.
	ProcessBlock	Encrypt an input block according to the encryptor configuration.
	Reset	Clear the crypto context.
	SetKey	Set (deploy) a key to the decryptor private algorithm context.

Name:	DecryptorPrivateCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	Asymmetric decryption private key context interface.	
Operations:	GetCryptoService	Get the `CryptoService` instance.
	ProcessBlock	Decrypt an input block according to the decryptor configuration.
	Reset	Clear the crypto context.
	SetKey	Set (deploy) a key to the decryptor private algorithm context.

Name:	SymmetricBlockCipherCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	Interface of a symmetric block cipher context with padding.	
Operations:	GetCryptoService	Get the `CryptoService` instance.
	GetTransformation	Get the kind of transformation configured for this context: Encrypt or Decrypt.
	ProcessBlock	Process (encrypt / decrypt) an input block according to the configuration.
	ProcessBlocks	Process (encrypt / decrypt) input blocks according to the configuration.
	Reset	Clear the crypto context.
	SetKey	Set (deploy) a key to the symmetric algorithm context.

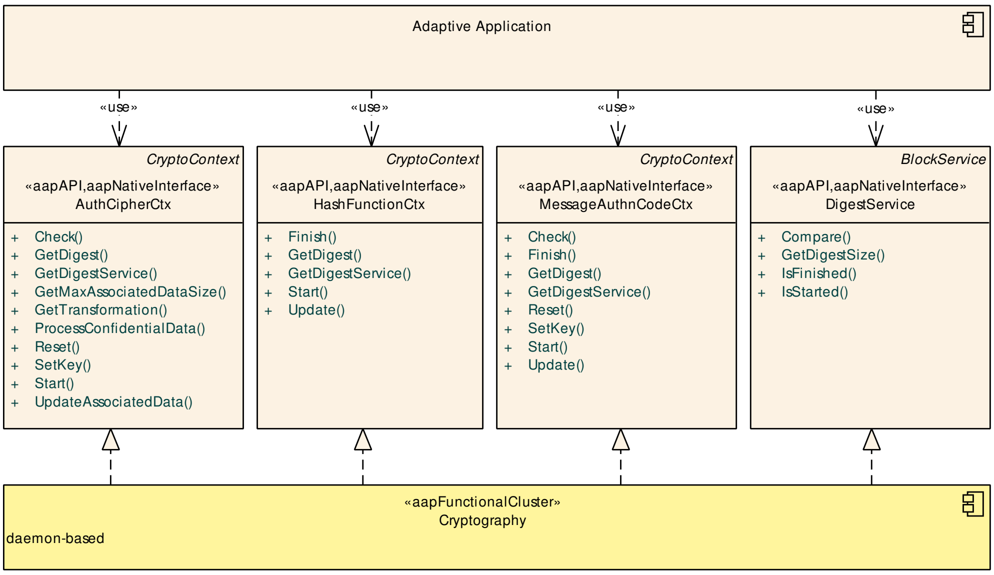

Figure 9.59: DigestService and CryptoContext Interfaces

Name:	DigestService	
Technology:	Native interface	
Usage:	Public API	
Description:	Extension meta-information service for digest producing contexts.	
Operations:	Compare	Compare the calculated digest against an expected value.
	GetDigestSize	Get the output digest size.
	IsFinished	Check current status of the stream processing: finished or not.
	IsStarted	Check current status of the stream processing: started or not.

Name:	AuthCipherCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	Generalized authenticated cipher context interface.	
Operations:	Check	Check the calculated digest against an expected signature object.
	GetDigest	Retrieve the calculated digest.
	GetDigestService	Get the `DigestService` instance.
	GetMaxAssociatedDataSize	Get maximal supported size of associated public data.
	GetTransformation	Get the kind of transformation configured for this context: Encrypt or Decrypt.
	ProcessConfidentialData	Process confidential data and return the result.
	Reset	Clear the crypto context.
	SetKey	Set (deploy) a key to the authenticated cipher symmetric algorithm context.
	Start	Initialize the context for a new data processing or generation (depending from the primitive).

▽

△

	UpdateAssociatedData	Update the digest calculation by the specified data.

Name:	HashFunctionCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	Hash function interface.	
Operations:	Finish	Finish the digest calculation and optionally produce the "signature" object.
	GetDigest	Get requested part of calculated digest.
	GetDigestService	Get the `DigestService` instance.
	Start	Initialize the context for a new data stream processing or generation (depending on the primitive).
	Update	Update the digest calculation context by a new part of the message.

Name:	MessageAuthnCodeCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	Keyed message authentication code context interface definition (MAC/HMAC).	
Operations:	Check	Check the calculated digest against an expected "signature" object.
	Finish	Finish the digest calculation and optionally produce the "signature" object.
	GetDigest	Get requested part of calculated digest to existing memory buffer.
	GetDigestService	Get the `DigestService` instance.
	Reset	Clear the crypto context.
	SetKey	Set (deploy) a key to the message authn code algorithm context.
	Start	Initialize the context for a new data stream processing or generation (depending from the primitive).
	Update	Update the digest calculation context by a new part of the message.

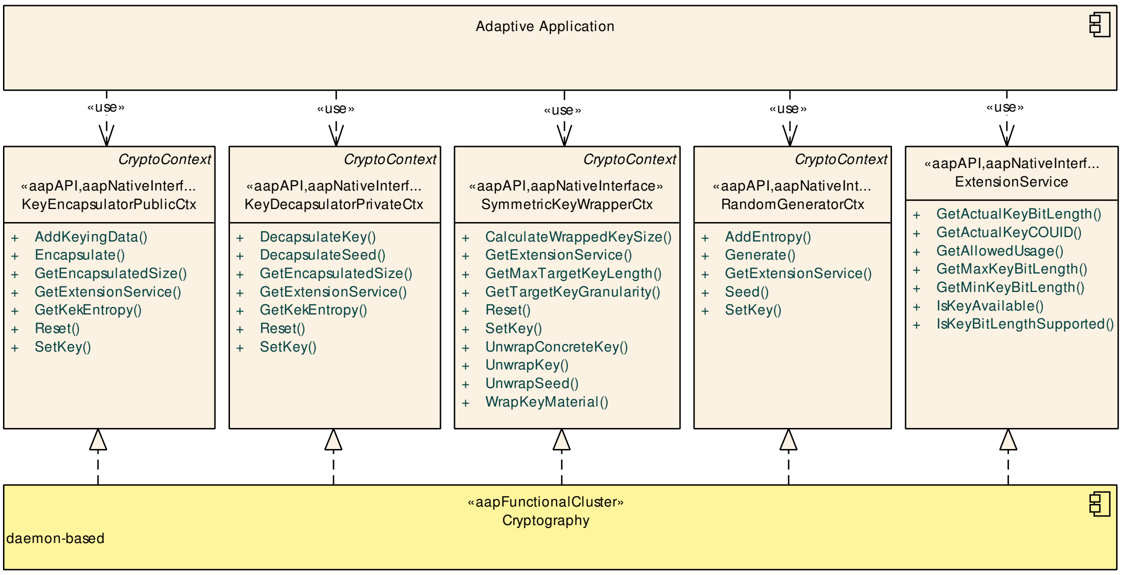

Figure 9.60: ExtensionService and CryptoContext Interfaces (1 of 2)

Name:	ExtensionService	
Technology:	Native interface	
Usage:	Public API	
Description:	Basic meta-information service for all contexts.	
Operations:	GetActualKeyBitLength	Get actual bit-length of a key loaded to the context.
	GetActualKeyCOUID	Get the `CryptoObjectUid` of the key deployed to the context this extension service is attached to.
	GetAllowedUsage	Get allowed usages of this context (according to the key object attributes loaded to this context).
	GetMaxKeyBitLength	Get maximum supported key length in bits.
	GetMinKeyBitLength	Get minimal supported key length in bits.
	IsKeyAvailable	Check if a key has been set to this context.
	IsKeyBitLengthSupported	Verify if the specific key length is supported by the context.

Name:	KeyEncapsulatorPublicCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	Asymmetric key encapsulation mechanism public key context interface.	
Operations:	AddKeyingData	Add the content to be encapsulated (payload) according to RFC 5990 ("keying data").
	Encapsulate	Encapsulate the last set keying-data.
	GetEncapsulatedSize	Get fixed size of the encapsulated data block.
	GetExtensionService	Get the `ExtensionService` instance.
	GetKekEntropy	Get entropy (bit-length) of the key encryption key (KEK) material.
	Reset	Clear the crypto context.
	SetKey	Set (deploy) a key to the key encapsulator public algorithm context.

Name:	KeyDecapsulatorPrivateCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	Asymmetric key encapsulation mechanism private key context interface.	
Operations:	DecapsulateKey	Decapsulate the keying data to be used for subsequent processing (e.g. secure communication).
	DecapsulateSeed	Decapsulate key material.
	GetEncapsulatedSize	Get fixed size of the encapsulated data block.
	GetExtensionService	Get the `ExtensionService` instance.
	GetKekEntropy	Get entropy (bit-length) of the key encryption key (KEK) material.
	Reset	Clear the crypto context.
	SetKey	Set (deploy) a key to the key decapsulator private algorithm context.

Name:	SymmetricKeyWrapperCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	Context of a symmetric key wrap algorithm (for AES it should be compatible with RFC3394 or RFC5649).	
Operations:	CalculateWrappedKeySize	Calculate size of the wrapped key in bytes from original key length in bits.
	GetExtensionService	Get the `ExtensionService` instance.
	GetMaxTargetKeyLength	Get maximum length of the target key supported by the implementation.
	GetTargetKeyGranularity	Get expected granularity of the target key (block size).
	Reset	Clear the crypto context.
	SetKey	Set (deploy) a key to the symmetric key wrapper algorithm context.
	UnwrapConcreteKey	Execute the "key unwrap" operation for the provided BLOB and produce a key object of the expected type.
	UnwrapKey	Execute the "key unwrap" operation for the provided BLOB and produce a key object.
	UnwrapSeed	Execute the "key unwrap" operation for the provided BLOB and produce a `SecretSeed` object.
	WrapKeyMaterial	Execute the "key wrap" operation for the provided key material.

Name:	RandomGeneratorCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	Interface of a random number generator context.	
Operations:	AddEntropy	Update the internal state of the RNG by mixing it with the provided additional entropy.
	Generate	Return an allocated buffer with a generated random sequence of the requested size.
	GetExtensionService	Get the `ExtensionService` instance.

▽

△

	Seed	Set the internal state of the RNG using the provided seed.
	SetKey	Set the internal state of the RNG using the provided seed.

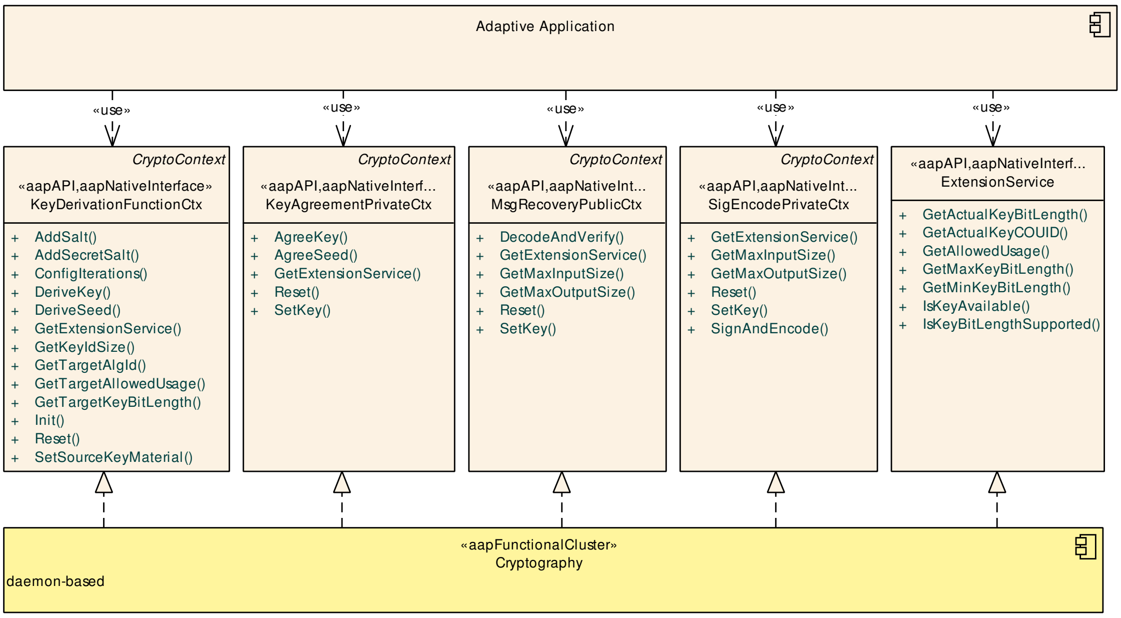

Figure 9.61: ExtensionService and CryptoContext Interfaces (2 of 2)

Name:	KeyDerivationFunctionCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	Key derivation function interface.	
Operations:	AddSalt	Add a salt value stored in a non-secret memory region.
	AddSecretSalt	Add a secret salt value stored in a `SecretSeed` object.
	ConfigIterations	Configure the number of iterations that will be applied by default.
	DeriveKey	Derive a symmetric key from the provided key material and provided context configuration.
	DeriveSeed	Derive key material (secret seed) from the provided "master" key material and the provided context configuration.
	GetExtensionService	Get the `ExtensionService` instance.
	GetKeyIdSize	Get the fixed size of the target key ID required by diversification algorithm.
	GetTargetAlgId	Get the symmetric algorithm ID of target key.
	GetTargetAllowedUsage	Get allowed key usage of target key.
	GetTargetKeyBitLength	Get the bit-length of target (diversified) keys.
	Init	Initialize this context by setting at least the target key ID.
	Reset	Clear the crypto context.

▽

△

	SetSourceKeyMaterial	Set (deploy) key-material to the key derivation algorithm context.

Name:	KeyAgreementPrivateCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	Key agreement private key context interface (Diffie Hellman or conceptually similar).	
Operations:	AgreeKey	Produce a common symmetric key via execution of the key-agreement algorithm between this private key.
	AgreeSeed	Produce a common secret seed via execution of the key-agreement algorithm between this private key.
	GetExtensionService	Get the `ExtensionService` instance.
	Reset	Clear the crypto context.
	SetKey	Set (deploy) a key to the key agreement private algorithm context.

Name:	MsgRecoveryPublicCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	A public key context for asymmetric recovery of a short message and its signature verification (RSA-like).	
Operations:	DecodeAndVerify	Process (encrypt / decrypt) an input block according to the cryptor configuration.
	GetExtensionService	Get the `ExtensionService` instance.
	GetMaxInputSize	Get maximum expected size of the input data block.
	GetMaxOutputSize	Get maximum possible size of the output data block.
	Reset	Clear the crypto context.
	SetKey	Set (deploy) a key to the msg recovery public algorithm context.

Name:	SigEncodePrivateCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	A private key context for asymmetric signature calculation and short message encoding (RSA-like).	
Operations:	GetExtensionService	Get the `ExtensionService` instance.
	GetMaxInputSize	Get maximum expected size of the input data block.
	GetMaxOutputSize	Get maximum possible size of the output data block.
	Reset	Clear the crypto context.
	SetKey	Set (deploy) a key to the sig encode private algorithm context.
	SignAndEncode	Process (encrypt / decrypt) an input block according to the cryptor configuration.

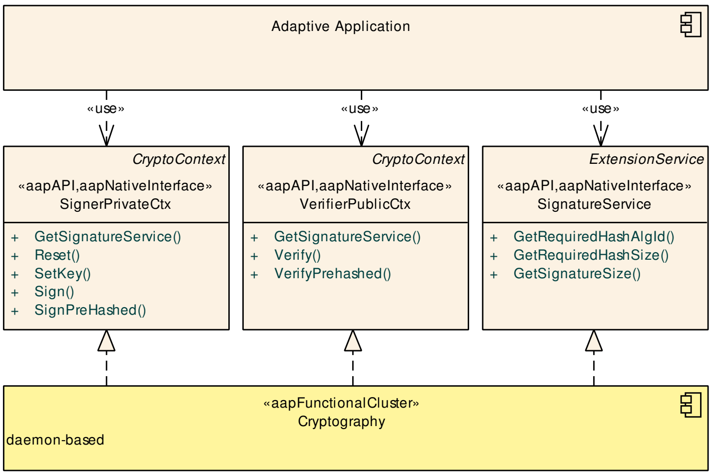

Figure 9.62: SignatureService and CryptoContext Interfaces

Name:	SignatureService	
Technology:	Native interface	
Usage:	Public API	
Description:	Extension meta-information service for signature contexts.	
Operations:	GetRequiredHashAlgId	Get an ID of hash algorithm required by current signature algorithm.
	GetRequiredHashSize	Get the hash size required by current signature algorithm.
	GetSignatureSize	Get size of the signature value produced and required by the current algorithm.

Name:	SignerPrivateCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	Signature private key context interface.	
Operations:	GetSignatureService	Get the `SignatureService` instance.
	Reset	Clear the crypto context.
	SetKey	Set (deploy) a key to the signer private algorithm context.
	Sign	Sign a directly provided hash or message value.
	SignPreHashed	Sign a provided digest value stored in the hash-function context.

Name:	VerifierPublicCtx	
Technology:	Native interface	
Usage:	Public API	
Description:	Signature verification public key context interface.	
Operations:	GetSignatureService	Get the `SignatureService` instance.
	Verify	Verify signature BLOB by a directly provided hash or message value.

▽

	VerifyPrehashed	Verify a signature by a digest value stored in the hash-function context.

9.5.1.1.4 Cryptographic object interfaces

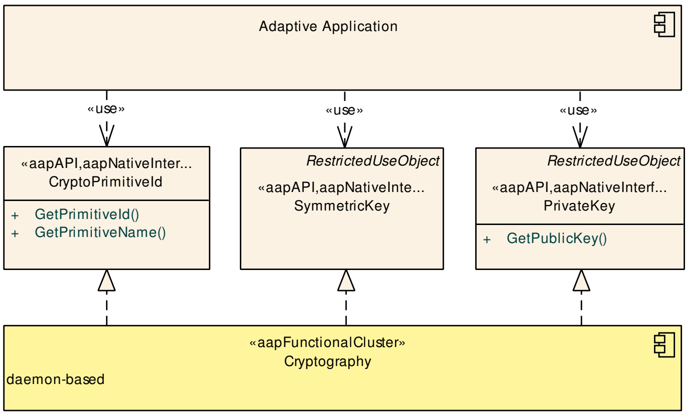

Figure 9.63: Cryptographic Object Interfaces

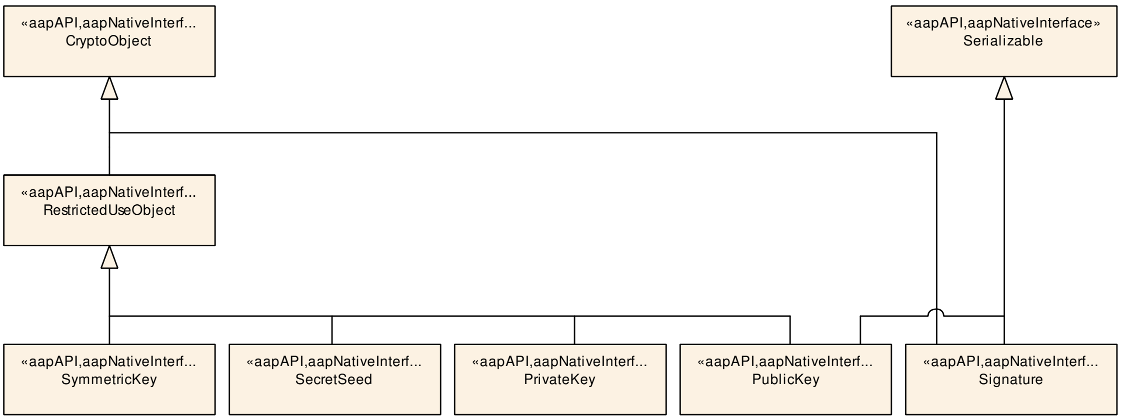

Figure 9.64: Taxonomy of Cryptographic Object Interfaces

Name:	CryptoObject	
Technology:	Native interface	
Usage:	Public API	
Description:	A common interface for all cryptographic objects recognizable by the `CryptoProvider`.	
Operations:	GetCryptoPrimitiveId	Return the `CryptoPrimitiveId` of this object.
	GetObjectId	Return the object's `COIdentifier`, which includes the object's type and UID.
	GetPayloadSize	Return actual size of the object's payload.
	HasDependence	Return the COIdentifier of the object that this object depends on.
	IsExportable	Get the `exportability` attribute of the crypto object.

△

	IsSession	Return the session (or temporary) attribute of the object.
	Save	Save itself to provided IOInterface

Name:	RestrictedUseObject	
Technology:	Native interface	
Usage:	Public API	
Description:	A common interface for all objects supporting the usage restriction.	
Operations:	GetAllowedUsage	Get allowed usages of this object.

Name:	CryptoPrimitiveId	
Technology:	Native interface	
Usage:	Public API	
Description:	Common interface for identification of all CryptoPrimitives and their keys and parameters.	
Operations:	GetPrimitiveId	Get vendor specific ID of the primitive.
	GetPrimitiveName	Get a unified name of the primitive.

Name:	SecretSeed	
Technology:	Native interface	
Usage:	Public API	
Description:	Secret seed object contains a raw bit sequence of specific length (without any filtering of allowed/ disallowed values).	
Operations:	Clone	Clone this object to new session object.
	Jump	Set value of this seed object as a "jump" from it's current state to specified number of steps, according to "counting" expression defined by a cryptographic algorithm associated with this object.
	JumpFrom	Set value of this seed object as a "jump" from an initial state to specified number of steps, according to "counting" expression defined by a cryptographic algorithm associated with this object.
	Next	Set next value of the secret seed according to "counting" expression defined by a cryptographic algorithm associated with this object.
	operator ˆ=	XOR value of this seed object with another one and save result to this object. If seed sizes in this object and in the source argument are different then only correspondent number of leading bytes in this seed object shall be updated.

Name:	SymmetricKey
Technology:	Native interface
Usage:	Public API
Description:	Symmetric Key interface.

Name:	PublicKey	
Technology:	Native interface	
Usage:	Public API	
Description:	General asymmetric public key interface.	
Operations:	CheckKey	Check the key for its correctness.
	HashPublicKey	Calculate hash of the public key value.

Name:	PrivateKey	
Technology:	Native interface	
Usage:	Public API	
Description:	Generalized asymmetric private key interface.	
Operations:	GetPublicKey	Get the public key correspondent to this private key.

Name:	Signature	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface is applicable for keeping the Digital Signature, Hash Digest, (Hash-based) Message Authentication Code (MAC/HMAC).	
Operations:	GetHashAlgId	Get an ID of hash algorithm used for this signature object production.
	GetRequiredHashSize	Get the hash size required by current signature algorithm.

9.5.1.1.5 Cryptographic key handling interfaces

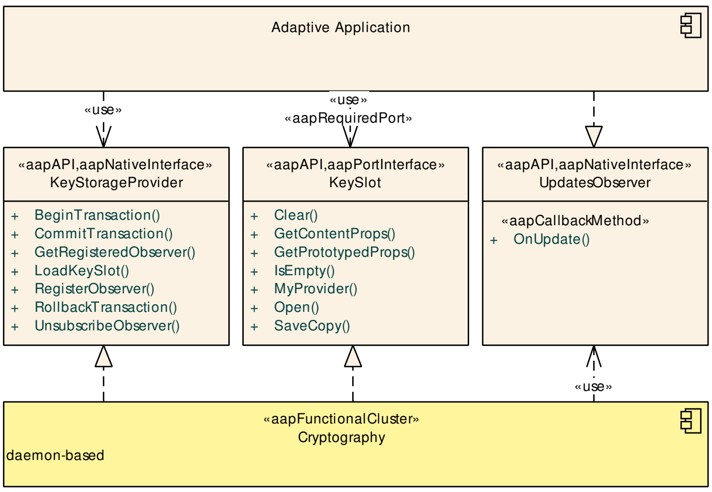

Figure 9.65: Cryptographic Key Handling Interfaces

Name:	KeyStorageProvider	
Technology:	Native interface	
Usage:	Public API	
Description:	Key Storage Provider interface.	
Operations:	BeginTransaction	Begin new transaction for key slots update.
	CommitTransaction	Commit changes of the transaction to storage.
	GetRegisteredObserver	Get the currently registered `UpdatesObserver` for key slots.
	LoadKeySlot	Load a key slot.
	RegisterObserver	Register an `UpdatesObserver` for key slots.
	RollbackTransaction	Rollback all changes executed during the transaction in storage.
	UnsubscribeObserver	Unregister an `UpdatesObserver` from key slots.

Name:	KeySlot	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`CryptoKeySlot`	
Usage:	Public API	
Description:	Key slot interface enables access to a physical key-slot.	
Operations:	Clear	Clear the content of this key slot.
	GetContentProps	Get an actual properties of a content in the key slot.
	GetPrototypedProps	Get the prototyped properties of the key slot.
	IsEmpty	Check the slot for emptiness.
	MyProvider	Retrieve the instance of the `CryptoProvider` that owns this `KeySlot`.
	Open	Open this key slot and return an `IOInterface` to its content.
	SaveCopy	Save the content of a provided source `IOInterface` to this key slot.

Name:	UpdatesObserver	
Technology:	Native interface	
Usage:	Public API	
Description:	Interface for observing updates on key slots.	
Operations:	OnUpdate	This method is called if the content of the specified slots was changed.

9.5.1.1.6 X.509 certificate handling interfaces

Figure 9.66: X.509 Certificate Handling Interfaces

Name:	X509Provider	
Technology:	Native interface	
Usage:	Public API	
Description:	X.509 Provider interface supporting two internal storage types: volatile (or session) and persistent.	
Operations:	BuildDn	Create completed X.500 Distinguished Name structure from the provided string representation.
	CheckCertStatus	Check certificate status by directly provided OCSP response.
	CheckCertStatusOnline	Check certificate status via On-line Certificate Status Protocol (OCSP).
	CleanupVolatileStorage	Cleanup the volatile certificates storage.
	CountCertsInChain	Count number of certificates in a serialized certificate chain represented by a single BLOB.
	CreateCertSignRequest	Create certification request for a private key loaded to the context.
	CreateEmptyDn	Create an empty X.500 Distinguished Name (DN) structure.
	CreateEmptyExtensions	Create an empty X.509 Extensions structure.
	CreateOcspRequest	Create OCSP request for specified certificate(s).

△

	DecodeDn	Decode X.500 Distinguished Name structure from the provided serialized format.
	FindCertByDn	Find a certificate by the subject and issuer Distinguished Names (DN).
	FindCertByKeyIds	Find a certificate by its SKID & AKID.
	FindCertBySn	Find a certificate by its serial number and issue DN.
	Import	Import the certificate to volatile or persistent storage.
	ImportCrl	Import Certificate Revocation List (CRL) or Delta CRL from a memory BLOB.
	LoadCertificate	Load a certificate from the persistent certificate storage.
	ParseCert	Parse a serialized representation of the certificate and create its instance.
	ParseCertChain	Parse a serialized representation of the certificate chain and create their instances.
	ParseCertSignRequest	Parse a certificate signing request (CSR) provided by the user.
	ParseCustomCertExtensions	Parse the custom X.509 extensions.
	ParseOcspResponse	Parse serialized OCSP response and create correspondent interface instance.
	Remove	Remove specified certificate from the storage (volatile or persistent) and destroy it.
	SendRequest	Send prepared certificate request to CA and save it to volatile or persistent storage.
	SetAsRootOfTrust	Set specified CA certificate as a "root of trust".
	SetPendingStatus	Set the "pending" status associated to the CSR that means that the CSR already sent to CA.
	UpdateCrlOnline	Get Certificate Revocation List (CRL) or Delta CRL via on-line connection.
	VerifyCert	Verify status of the provided certificate by locally stored CA certificates and CRLs only.
	VerifyCertChain	Verify status of the provided certification chain by locally stored CA certificates and CRLs only.

Name:	X509CustomExtensionsParser	
Technology:	Native interface	
Usage:	Public API	
Description:	X.509 custom extensions parser. This callback interface is to be implemented by an application.	
Operations:	OnBitString	Called when a bit string is encountered.
	OnBool	Called when a boolean is encountered.
	OnGeneralizedTime	Called when a generalized time is encountered.
	OnIa5String	Called when an IA5 string is encountered.
	OnInteger	Called when an integer is encountered.
	OnNull	Called when a NULL is encountered.
	OnOctetString	Called when an octet string is encountered.
	OnOid	Called when an oid is encountered.
	OnParsingEnd	Called when the parsing is completed.
	OnPrintableString	Called when a printable string is encountered.
	OnSequenceEnd	Called when a sequence ends.

▽

△

	OnSequenceStart	Called when a sequence starts.
	OnSetEnd	Called when a set ends.
	OnSetStart	Called when a set starts.
	OnUtcTime	Called when a UTC time is encountered.
	OnUtf8String	Called when an UTF8 string is encountered.

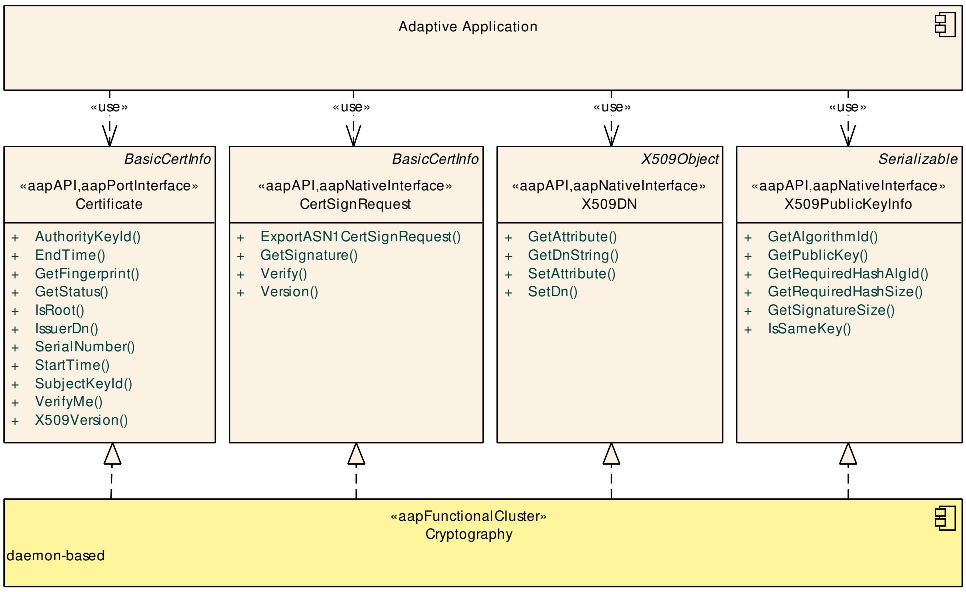

Figure 9.67: X.509 Certificate Object Interfaces

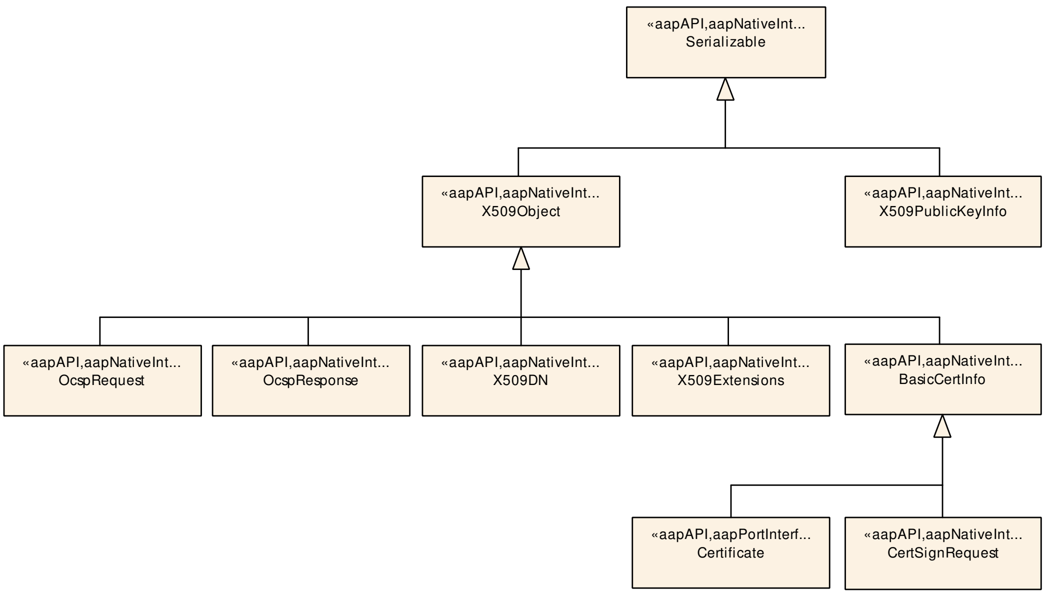

Figure 9.68: Taxonomy of X.509 Certificate Object Interfaces

Name:	OcspRequest	
Technology:	Native interface	
Usage:	Public API	
Description:	On-line Certificate Status Protocol Request.	
Operations:	Version	Get version of the OCSP request format.

Name:	OcspResponse	
Technology:	Native interface	
Usage:	Public API	
Description:	On-line Certificate Status Protocol Response.	
Operations:	Version	Get version of the OCSP response format.

Name:	Certificate	
Technology:	Native interface	
Usage:	Public API	
Description:	X.509 Certificate interface.	
Operations:	AuthorityKeyId	Get the DER encoded `AuthorityKeyIdentifier` of this certificate.
	EndTime	Get the `NotAfter` of the certificate.
	GetFingerprint	Calculate a fingerprint from the whole certificate.
	GetStatus	Return last verification status of the certificate.
	IsRoot	Check whether this certificate belongs to a root CA.
	IssuerDn	Get the issuer certificate DN.
	SerialNumber	Get the serial number of this certificate.
	StartTime	Get the `NotBefore` of the certificate.
	SubjectKeyId	Get the DER encoded `SubjectKeyIdentifier` of this certificate.
	VerifyMe	Verify signature of the certificate.
	X509Version	Get the X.509 version of this certificate object.

Name:	CertSignRequest	
Technology:	Native interface	
Usage:	Public API	
Description:	Certificate Signing Request (CSR) object interface.	
Operations:	ExportASN1CertSignRequest	Export this certificate signing request in DER encoded ASN1 format.
	GetSignature	Return signature object of the request.
	Verify	Verifies self-signed signature of the certificate request.
	Version	Return format version of the certificate request.

Name:	X509Extensions	
Technology:	Native interface	
Usage:	Public API	
Description:	Interface of X.509 Extensions.	
Operations:	Count	Count number of elements in the sequence.

Name:	X509DN	
Technology:	Native interface	
Usage:	Public API	
Description:	Interface of X.509 Distinguished Name (DN).	
Operations:	GetAttribute	Get a DN attribute.
	GetDnString	Get the whole Distinguished Name (DN) as a single string.
	SetAttribute	Set a DN attribute.
	SetDn	Set whole Distinguished Name (DN) from a single string.

Name:	X509PublicKeyInfo	
Technology:	Native interface	
Usage:	Public API	
Description:	X.509 Public Key Information interface.	
Operations:	GetAlgorithmId	Get the CryptoPrimitiveId instance of this class.
	GetPublicKey	Get public key object of the subject.
	GetRequiredHashAlgId	Get an ID of hash algorithm required by current signature algorithm.
	GetRequiredHashSize	Get the hash size required by current signature algorithm.
	GetSignatureSize	Get size of the signature value produced and required by the current algorithm.
	IsSameKey	Verify the sameness of the provided and kept public keys.

9.5.1.2 Provided interfaces

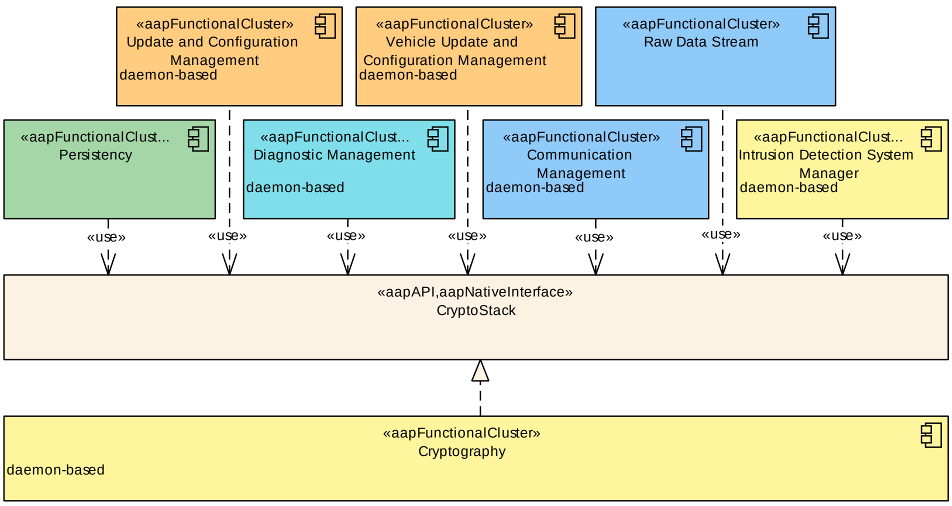

Figure 9.69: Users of the Cryptography interfaces

Interface	*Requiring functional clusters*
CryptoStack	Communication Management
	Diagnostic Management
	Intrusion Detection System Manager
	Persistency
	Raw Data Stream
	Time Synchronization
	Update and Configuration Management
	Vehicle Update and Configuration Management

Table 9.20: Interfaces provided by Cryptography to other Functional Clusters

9.5.1.3 Required interfaces

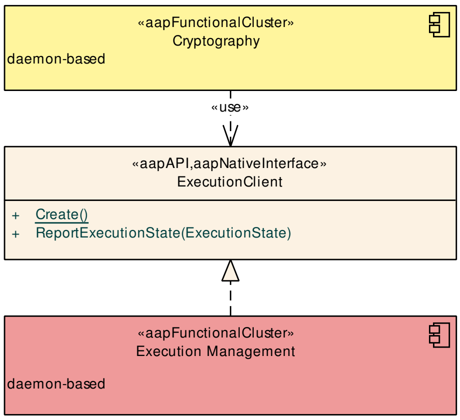

Figure 9.70: Interfaces required by Cryptography

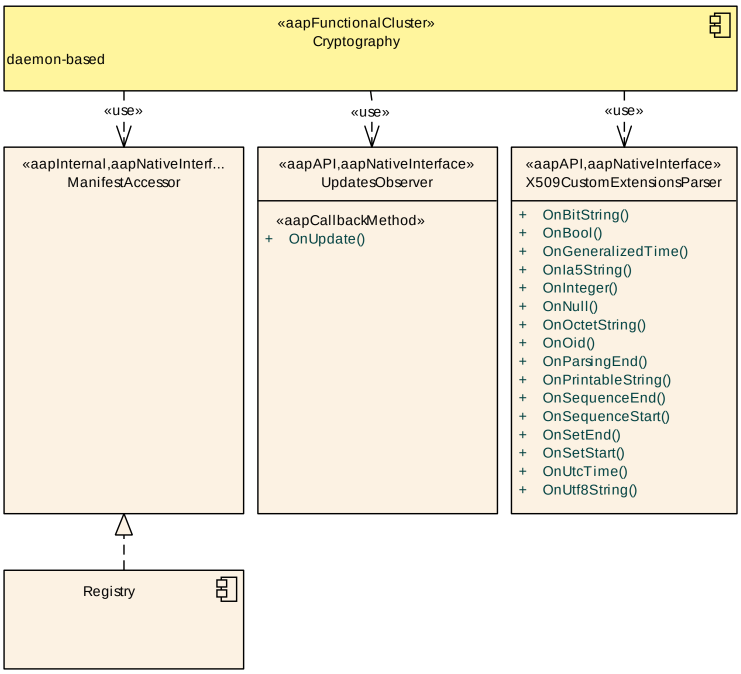

Figure 9.71: Interfaces required by Cryptography

Interface	*Purpose*
ManifestAccessor	Cryptography shall use this interface to read its configuration information from the Manifests.
Cryptography::UpdatesObserver	Cryptography uses this interface to notify when a key has been updated.
Cryptography::X509CustomExtensionsParser	Cryptography uses this interface for propagating parser events.
Execution Management::ExecutionClient	

Table 9.21: Interfaces required by Cryptography

9.5.2 Intrusion Detection System Manager

Name:	Intrusion Detection System Manager
Short Name:	idsm
Category:	Security
Daemon-based:	Yes
Responsibilities:	Intrusion Detection System Manager provides functionality to report security events.

9.5.2.1 Defined interfaces

The interfaces of Intrusion Detection System Manager are categorized into interfaces for reporting security events (see Section 9.5.2.1.1), interfaces for providing additional data for security events (see Section 9.5.2.1.2), and and interface for receiving notifications about qualified security events (see Section 9.5.2.1.3).

9.5.2.1.1 Interfaces for Reporting Security Events

This section lists interfaces for reporting security events.

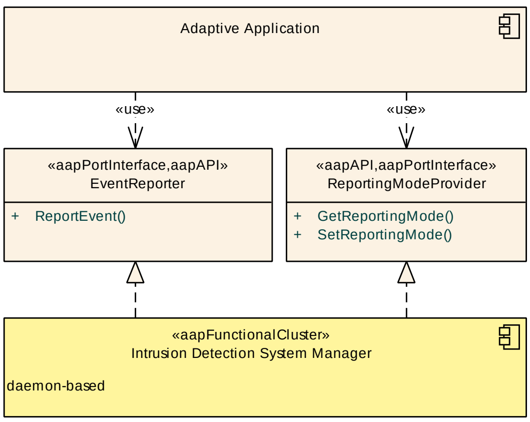

Figure 9.72: Interfaces for reporting security events

Name:	EventReporter	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`SecurityEventDefinition`	
Usage:	Public API	
Description:	This interface is used to report security events to the `Intrusion Detection System Manager`.	
Operations:	ReportEvent	Create a new security event at the `Adaptive Intrusion Detection System Manager`.

Name:	ReportingModeProvider	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`IdsmReportingModeProviderInterface`	
Usage:	Public API	
Description:	Interface for providing reporting modes to the `Intrusion Detection System Manager`.	
Operations:	GetReportingMode	Get the ReportingMode for an event type.
	SetReportingMode	Set the ReportingMode for an event type.

9.5.2.1.2 Interfaces for Providing Data for Security Events

This section lists interfaces for providing additional data for security events, for example, timestamps or context data.

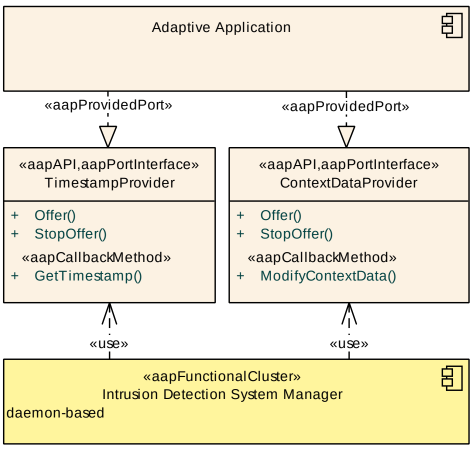

Figure 9.73: Interfaces for providing additional data for security events

Name:	ContextDataProvider
Technology:	Port interface
Generated:	No
Meta-model interface type:	`IdsmContextProviderInterface`

△

Usage:	Public API	
Description:	Interface for an Adaptive Application to modify the original context data.	
Operations:	*ModifyContextData*	Callback to modify the original context data.
	Offer	Enables possible invocations of `ModifyContextData()` by `Intrusion Detection System Manager`.
	StopOffer	Disables invocations of `ModifyContextData()`.

Name:	TimestampProvider	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`IdsmTimestampProviderInterface`	
Usage:	Public API	
Description:	Interface for an Adaptive Application to provide a timestamp.	
Operations:	*GetTimestamp*	Callback function to retrieve the timestamp.
	Offer	Enables possible invocations of `GetTimestamp()` by `Intrusion Detection System Manager`.
	StopOffer	Disables invocations of `GetTimestamp()`.

9.5.2.1.3 Interface for Receiving Notifications about Qualified Security Events

This section lists an interface for receiving notifications about qualified security events.

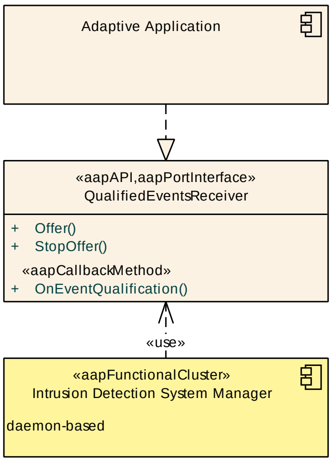

Figure 9.74: Interface for receiving notifications about qualified security events

Name:	QualifiedEventsReceiver
Technology:	Port interface
Generated:	No
Meta-model interface type:	`IdsmQualifiedEventsReceiverInterface`
Usage:	Public API

▽

△

Description:	Interface for receiving qualified events from the Intrusion Detection System Manager.	
Operations:	Offer	Enables invocations of OnEventQualification.
	OnEventQualification	This function is implemented by the application and invoked by Intrusion Detection System Manager on qualification of a security event with versioned context data.
	StopOffer	Disables invocations of OnEventQualification.

9.5.2.2 Provided interfaces

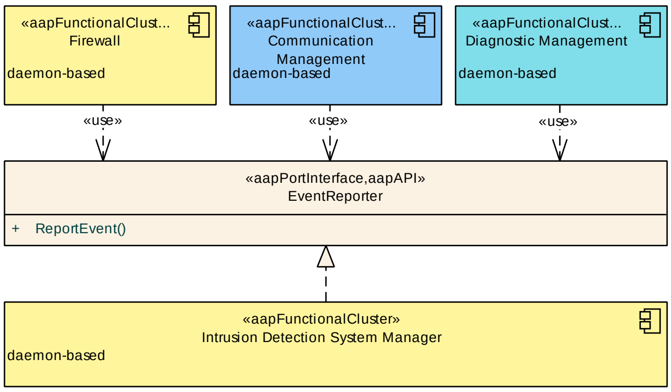

Figure 9.75: Users of the Intrusion Detection System Manager interfaces

Interface	***Requiring functional clusters***
EventReporter	Communication Management
	Diagnostic Management
	Firewall

Table 9.22: Interfaces provided by Intrusion Detection System Manager to other Functional Clusters

9.5.2.3 Required interfaces

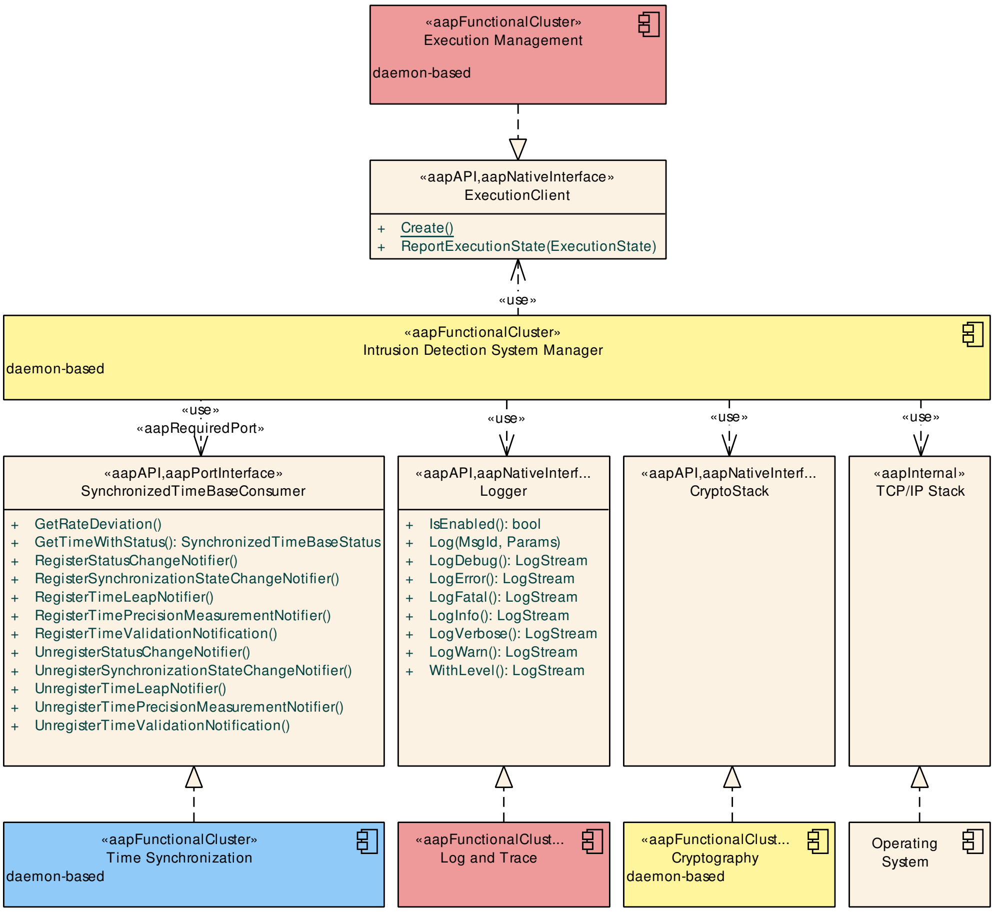

Figure 9.76: Interfaces required by Intrusion Detection System Manager

Interface	*Purpose*
TCP/IP Stack	`Adaptive Intrusion Detection System Manager` shall use this interface to propagate qualified security events via the IDS protocol.
Cryptography::CryptoStack	`Adaptive Intrusion Detection System Manager` uses this interface to sign security events.
Execution Management::ExecutionClient	
Intrusion Detection System Manager::Context DataProvider	This interface is used to let an application modify the context data for security events.
Intrusion Detection System Manager::Qualified EventsReceiver	Used to notify about qualified events.
Intrusion Detection System Manager:: TimestampProvider	This interface is used to let an application provide a timestamp for security events.
Log and Trace::Logger	`Adaptive Intrusion Detection System Manager` shall use this interface to log standardized messages.
Time Synchronization::SynchronizedTimeBase Consumer	`Adaptive Intrusion Detection System Manager` shall use this interface to determine timestamps of security events.

Table 9.23: Interfaces required by Intrusion Detection System Manager

9.5.3 Firewall

Name:	Firewall
Short Name:	fw
Category:	Security
Daemon-based:	Yes
Responsibilities:	The `Firewall` is responsible for filtering network traffic based on firewall rules to protect the system from malicious messages. To this end, the `Firewall` parses the firewall rules from the `Manifest` and configures the underlying firewall engine accordingly. The firewall engine can be realized in different ways (e.g. on the level of the TCP/IP stack or even closer to the hardware), which is considered to be an implementation detail. Additionally, the `Firewall` supports handling of different modes (e.g. driving, parking, diagnostic session) by enabling/disabling firewall rules based on the active mode. For blocked messages security events to the `Adaptive Intrusion Detection System Manager` will be reported to support the AUTOSAR intrusion detection system.

9.5.3.1 Defined interfaces

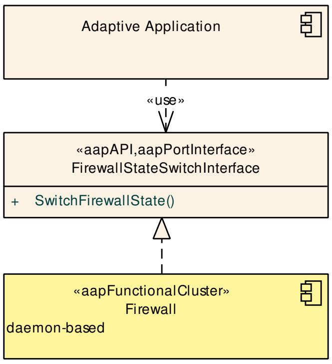

Figure 9.77: Interfaces of Firewall

Name:	FirewallStateSwitchInterface	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`FirewallStateSwitchInterface`	
Usage:	Public API	
Description:	Provides functionality to switch the firewall state.	
Operations:	SwitchFirewallState	This method triggers a switch of the firewall state.

9.5.3.2 Provided interfaces

The `Firewall` does not provide interfaces to other `Functional Cluster`s.

9.5.3.3 Required interfaces

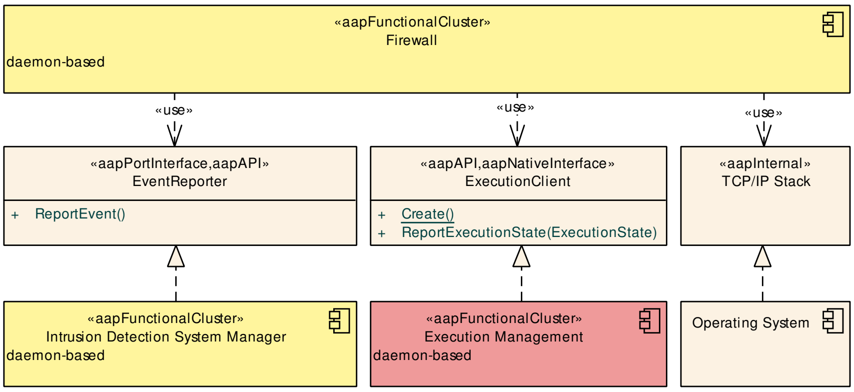

Figure 9.78: Interfaces required by Firewall

Interface	***Purpose***
TCP/IP Stack	The Firewall uses this interface to enable/disable firewall rules.
Execution Management::ExecutionClient	
Intrusion Detection System Manager::Event Reporter	The Firewall uses this interface to report standardized security events.

Table 9.24: Interfaces required by Firewall

9.6 Safety

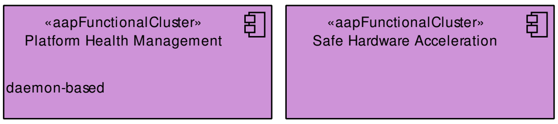

Figure 9.79: Functional Clusters in category Safety

9.6.1 Platform Health Management

Name:	Platform Health Management
Short Name:	phm
Category:	Safety
Daemon-based:	Yes

▽

△

Responsibilities:	Platform Health Management performs (aliveness, logical, and deadline) supervision of Processes in safety-critical setups and reports failures to State Management. Platform Health Management also controls the Watchdog that in turn supervises the Platform Health Management. An Alive Supervision checks that a supervised entity is not running too frequently and not too rarely. A Deadline Supervision checks that steps in a supervised entity are executed within the configured minimum and maximum time. A Logical Supervision checks that the control flow during execution matches the designed control flow. All types of supervision can be used independently and are performed based on reporting of Checkpoints by the supervised entity. State Management and Execution Management are the fundamental Functional Clusters of the AUTOSAR Adaptive Platform and need to run and work properly in any case. Therefore, Platform Health Management shall always supervise the corresponding Processes for State Management and Execution Management. Supervision failures in these Processes shall be recovered by a reset of the Machine because the normal way of error recovery (via State Management and Execution Management) is no longer reliable.

9.6.1.1 Defined interfaces

The interfaces of Platform Health Management are categorized into interfaces for supervision (see Section 9.6.1.1.1), interfaces for performing recovery actions (see Section 9.6.1.1.2), and interfaces for hardware watchdog handling (see Section 9.6.1.1.3).

9.6.1.1.1 Interfaces for supervision

Processes that are supervised by Platform Health Management shall report via the SupervisedEntity interface when they have reached a certain checkpoint in their control flow (see Figure 9.80). Platform Health Management independently monitors that all checkpoints configured in the Manifest have been reached in time and in the expected order (depending on the type of supervision).

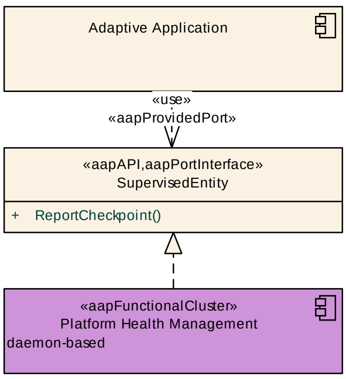

Figure 9.80: Interfaces for supervision

Name:	SupervisedEntity	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	PhmSupervisedEntityInterface	
Usage:	Public API	
Description:	This interface provides functions to report checkpoints to Platform Health Management.	
Operations:	ReportCheckpoint	Reports an occurrence of a checkpoint.

9.6.1.1.2 Interfaces for recovery

Platform Health Management defines the RecoveryAction API to trigger a recovery action in case a supervision failed (see Figure 9.81).

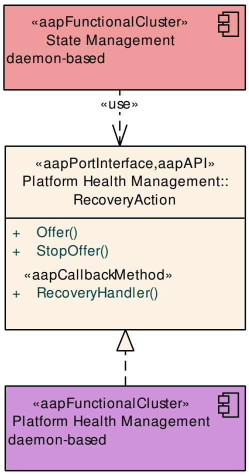

Figure 9.81: Interfaces for recovery

Name:	RecoveryAction	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	PhmRecoveryActionInterface	
Usage:	Public API	
Description:	This interface provides functions to control triggering of recovery actions, to determine the status of the supervision and a callback to perform recovery.	
Operations:	Offer	Enables potential invocations of the callback RecoveryHandler().
	RecoveryHandler	Callback to be invoked by Platform Health Management upon a supervision failure. The handler invocation needs to be enabled before using Offer().
	StopOffer	Disables potential invocations of the callback RecoveryHandler().

9.6.1.1.3 Interfaces for watchdog handling

Platform Health Management defines the WatchdogInterface extension API to interact with the hardware watchdog (see Figure 9.82).

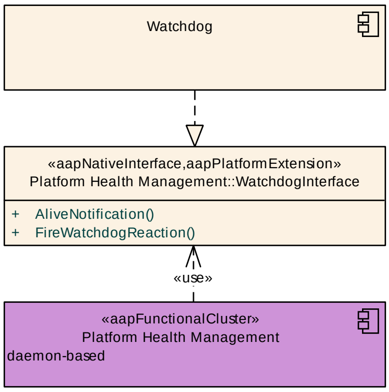

Figure 9.82: Interfaces for watchdog handling

Name:	WatchdogInterface	
Technology:	Native interface	
Usage:	Platform extension	
Description:	This interface provides functions to control the hardware watchdog.	
Operations:	AliveNotification	Called cyclically by Platform Health Management in configurable cycle time. Note: This time might differ from the cycle time of triggering the "real" hardware watchdog. If Platform Health Management does not report aliveness in configured time, Watchdog Interface shall initiate watchdog reaction.
	FireWatchdogReaction	Initiates an error reaction of the hardware watchdog.

9.6.1.2 Provided interfaces

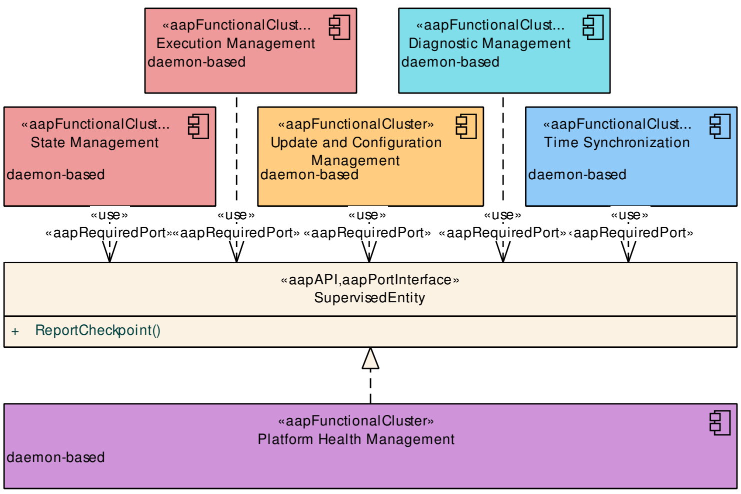

Figure 9.83: Users of the SupervisedEntity interface

Interface	Requiring functional clusters
RecoveryAction	State Management
SupervisedEntity	Diagnostic Management
	Execution Management
	State Management
	Time Synchronization
	Update and Configuration Management

Table 9.25: Interfaces provided by Platform Health Management to other Functional Clusters

9.6.1.3 Required interfaces

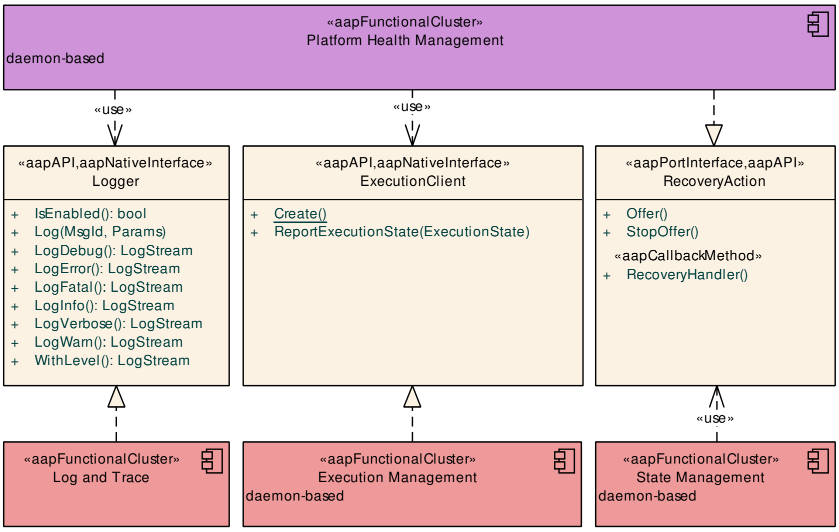

Figure 9.84: Interfaces required by Platform Health Management

Interface	Purpose
ManifestAccessor	Platform Health Management shall use this interface to read information about SupervisedEntities from the Manifests.
Execution Management::ExecutionClient	Platform Health Management uses this interface to report the state of its daemon process to Execution Management.
Log and Trace::Logger	Platform Health Management shall use this interface to log standardized messages.
Platform Health Management::Watchdog Interface	Platform Health Management uses this interface to control the hardware watchdog.

Table 9.26: Interfaces required by Platform Health Management

9.6.2 Safe Hardware Acceleration

Name:	Safe Hardware Acceleration
Short Name:	shwa
Category:	Safety
Daemon-based:	No
Responsibilities:	Safe Hardware Acceleration offers a mechanism to utilize hardware acceleration for different tasks which can be executed in parallel.

9.6.2.1 Defined interfaces

This section lists the interfaces of Safe Hardware Acceleration. Figure 9.85 shows the interfaces for hardware accelerated execution. Figure 9.86 shows the device and related interfaces of Safe Hardware Acceleration. Figure 9.87 and 9.88 show the common interfaces of Safe Hardware Acceleration.

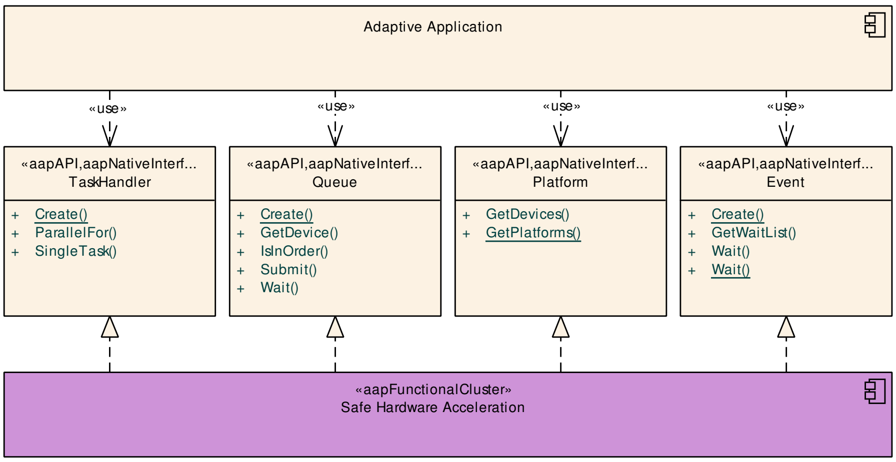

Figure 9.85: Interfaces for hardware accelerated execution

Name:	TaskHandler	
Technology:	Native interface	
Usage:	Public API	
Description:	Handles tasks execution on the selected Device.	
Operations:	Create	
	ParallelFor	Executes in parallel with one work-item per array index on the selected Device. Can be executed only within submitted task to Queue.
	SingleTask	Executes once on the selected Device.

Name:	Queue
Technology:	Native interface
Usage:	Public API
Description:	Provides functionality for parallel processing on the selected Device.

▽

△

Operations:	Create	Creates a new Queue.
	GetDevice	Get the device bound to the Queue.
	IsInOrder	Get if the Queue tasks execution is ordered.
	Submit	Submits a task to the Device for execution.
	Wait	Performs a blocking wait for the completion of all enqueued tasks in the Queue.

Name:	Platform	
Technology:	Native interface	
Usage:	Public API	
Description:	Interface for querying the available Devices.	
Operations:	GetDevices	Get the available Devices.
	GetPlatforms	Get the available Platforms.

Name:	Event	
Technology:	Native interface	
Usage:	Public API	
Description:	Manages task execution.	
Operations:	Create	Creates a new Event.
	GetWaitList	Get the events that the Event waits for in the dependence graph.
	Wait	Blocks until all commands associated with this Event and any dependent events have completed.
	Wait	Behaves as if calling Wait method on each event in the event list.

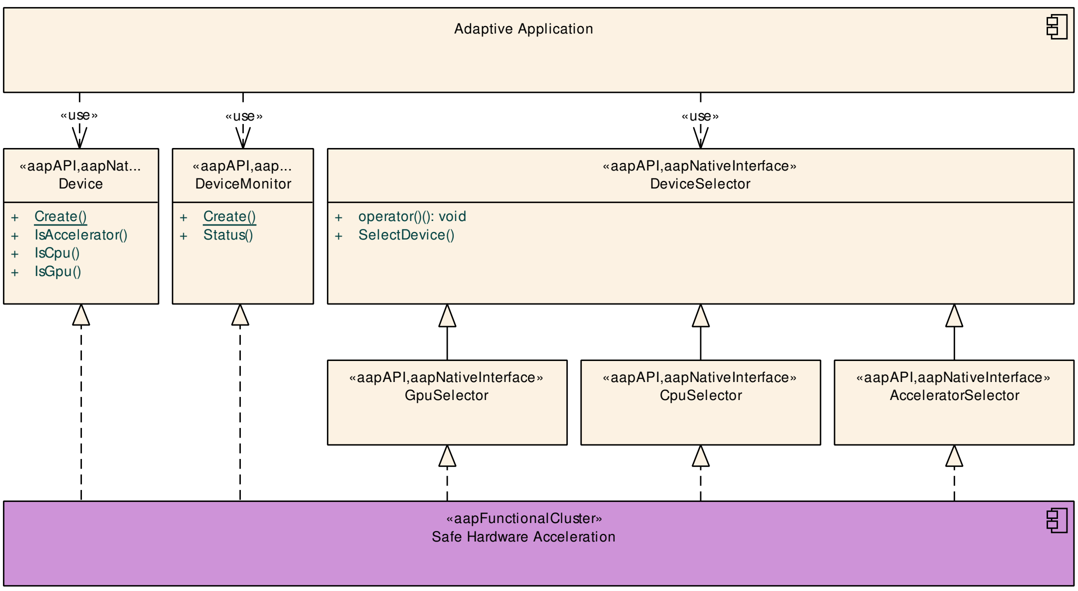

Figure 9.86: Device and related interfaces of Safe Hardware Acceleration

Name:	Device	
Technology:	Native interface	
Usage:	Public API	
Description:	Interface for devices.	
Operations:	Create	Creates a new Device.
	IsAccelerator	Checks if the device type is Accelerator.
	IsCpu	Checks if the device type is CPU.
	IsGpu	Checks if the device type is GPU.

Name:	DeviceMonitor	
Technology:	Native interface	
Usage:	Public API	
Description:	Interface for monitoring devices.	
Operations:	Create	Creates a DeviceMonitor.
	Status	Get the device status.

Name:	DeviceSelector	
Technology:	Native interface	
Usage:	Public API	
Description:	Allows to choose the most appropriate device among the available devices.	
Operations:	SelectDevice	Get the device created as an instance of accelerator device.
	operator()	Get the highest device rank based on the specified device.

Name:	GpuSelector
Technology:	Native interface
Usage:	Public API
Description:	Manages GPU selection for hardware accelerated operations if available.

Name:	CpuSelector
Technology:	Native interface
Usage:	Public API
Description:	Manages CPU selection for hardware accelerated operations if available.

Name:	AcceleratorSelector
Technology:	Native interface
Usage:	Public API
Description:	Manages accelerator selection for hardware accelerated operations if available.

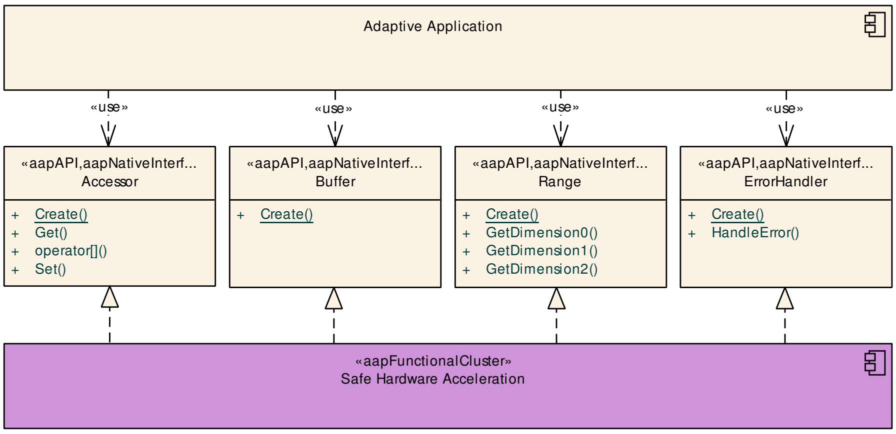

Figure 9.87: Common interfaces of Safe Hardware Acceleration (1)

Name:	Accessor	
Technology:	Native interface	
Usage:	Public API	
Description:	Manages read and write access to allocated memory by a Buffer.	
Operations:	Create	Creates an Accessor.
	Get	Get the element at the specified index.
	Set	Set the element at the specified index.
	operator[]	Get the element at the specified index.

Name:	Buffer	
Technology:	Native interface	
Usage:	Public API	
Description:	Manages memory allocation for operations on the host or/and hardware accelerator.	
Operations:	Create	Creates a Buffer.

Name:	Range	
Technology:	Native interface	
Usage:	Public API	
Description:	Defines size of Buffer for one, two or three dimensions.	
Operations:	Create	Creates a Range for one, two or three dimensions.
	GetDimension0	Get the value of the first dimension.
	GetDimension1	Get the value of the second dimension.
	GetDimension2	Get the value of the third dimension.

Name:	ErrorHandler	
Technology:	Native interface	
Usage:	Public API	
Description:	Provides asynchronous error handling for hardware accelerated operations.	
Operations:	Create	Creates a new ErrorHandler.
	HandleError	Handle the provided error code.

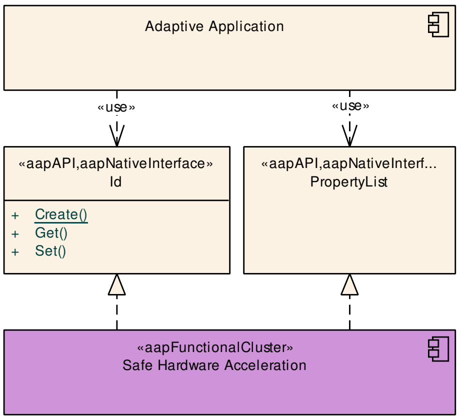

Figure 9.88: Common interfaces of Safe Hardware Acceleration (2)

Name:	Id	
Technology:	Native interface	
Usage:	Public API	
Description:	An identifier.	
Operations:	Create	Creates an Id.
	Get	Get the current id dimension.
	Set	Set the current id dimension.

Name:	PropertyList
Technology:	Native interface
Usage:	Public API
Description:	A list of properties.

9.6.2.2 Provided interfaces

Interface	***Requiring functional clusters***
No interfaces provided.	

Table 9.27: Interfaces provided by Safe Hardware Acceleration to other Functional Clusters

9.6.2.3 Required interfaces

Interface	***Purpose***
No interfaces required.	

Table 9.28: Interfaces required by Safe Hardware Acceleration

9.7 Configuration

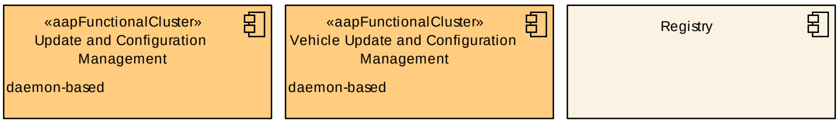

Figure 9.89: Functional Clusters in category Configuration

9.7.1 Update and Configuration Management

Name:	Update and Configuration Management
Short Name:	ucm
Category:	Configuration
Daemon-based:	Yes
Responsibilities:	`Update and Configuration Management` is responsible for updating, installing, removing and keeping a record of the software on an AUTOSAR Adaptive Platform in a safe and secure way.

9.7.1.1 Defined interfaces

The interfaces of `Update and Configuration Management` are categorized into general interfaces (see Section 9.7.1.1.1) and interfaces for the D-PDU API (see Section 9.7.1.1.2).

9.7.1.1.1 General

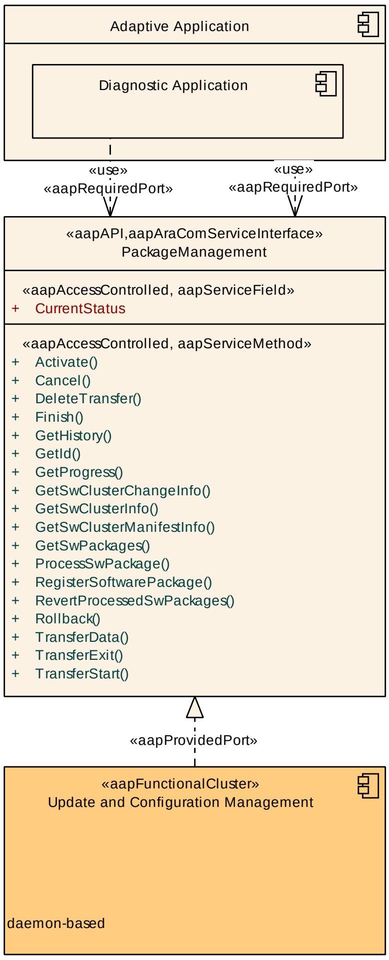

Figure 9.90: General Interfaces of Update and Configuration Management

Name:	PackageManagement	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface provides functionality for managing and transferring `Software Package`s to `Update and Configuration Management`.	
Operations:	Activate	Activates the processed components.
	Cancel	Aborts an ongoing processing of a `Software Package`.
	DeleteTransfer	Delete a transferred `Software Package`.
	Finish	Finishes the processing for the current set of processed `Software Package`s. It does a cleanup of all data of the processing including the sources of the `Software Package`s.
	GetHistory	Retrieve all actions that have been performed by `Update and Configuration Management`.
	GetId	Get the `Update and Configuration Management` Instance Identifier.

△

	GetProgress	Get the progress information of the currently active state in the Package Management state machine.
	GetSwClusterChangeInfo	Get a list of pending changes to the set of Software Clusters on the Adaptive Platform. The returned list includes all Software Clusters that are to be added, updated or removed. The list of changes is extended in the course of processing Software Packages.
	GetSwClusterInfo	Get a list of Software Clusters that are in state kPresent.
	GetSwClusterManifestInfo	Get general information of the Software Clusters present in the platform.
	GetSwPackages	Get the Software Packages that available in Update and Configuration Management.
	ProcessSwPackage	Process a previously transferred Software Package.
	RegisterSoftwarePackage	Registers a package that could already be in Machine's file system or to be downloaded.
	RevertProcessedSwPackages	Revert the changes done by processing (by calling ProcessSwPackage()) of one or several Software Packages.
	Rollback	Rollback the system to the state before the packages were processed.
	TransferData	Block-wise transfer of a Software Package to Update and Configuration Management.
	TransferExit	Finish the transfer of a Software Package to Update and Configuration Management.
	TransferStart	Start the transfer of a Software Package.
Fields:	CurrentStatus	The current status of Update and Configuration Management.

9.7.1.1.2 D-PDU API

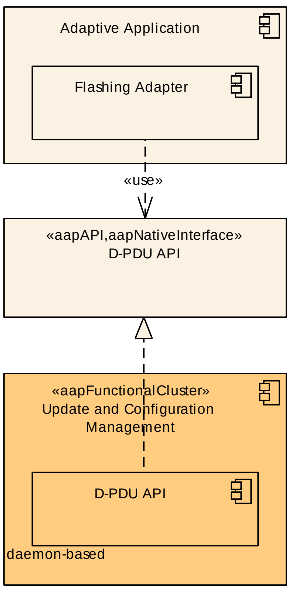

Figure 9.91: D-PDU-API Interfaces

Name:	D-PDU API
Technology:	Native interface
Usage:	Public API
Description:	This interface represents the Diagnostic Protocol Data Unit Application Programming Interface as specified in ISO 22900-2 . This interface is not detailed in this document.

9.7.1.2 Provided interfaces

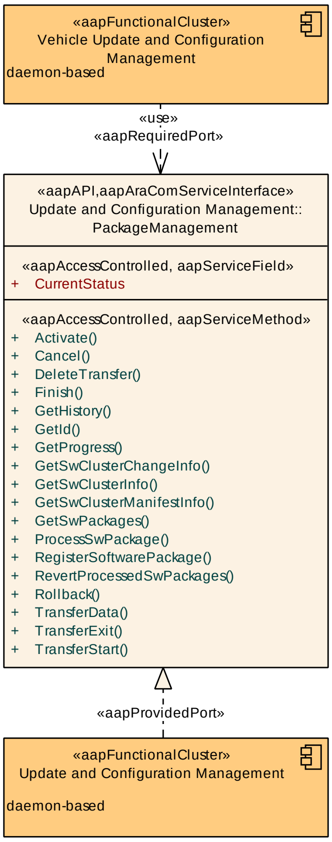

Figure 9.92: Users of the Update and Configuration Management interfaces

Interface	***Requiring functional clusters***
PackageManagement	Vehicle Update and Configuration Management

Table 9.29: Interfaces provided by Update and Configuration Management to other Functional Clusters

9.7.1.3 Required interfaces

Figures 9.93, 9.94, and 9.95 show the interfaces that are required by `Update and Configuration Management`.

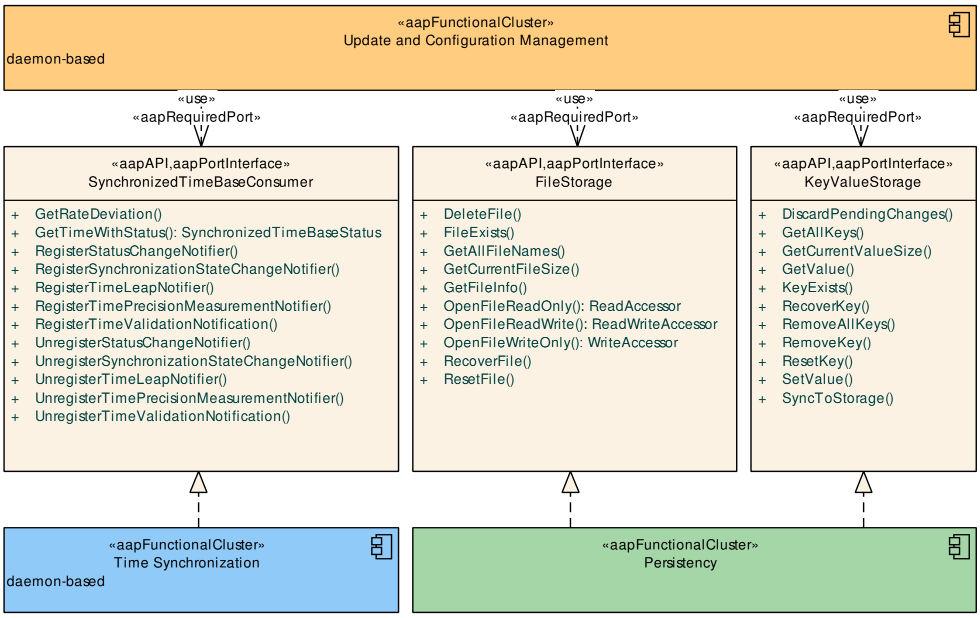

Figure 9.93: Interfaces required by Update and Configuration Management (part 1)

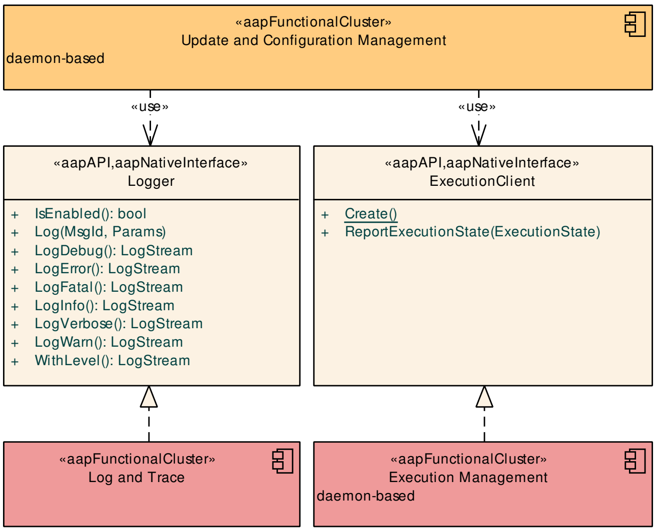

Figure 9.94: Interfaces required by Update and Configuration Management (part 2)

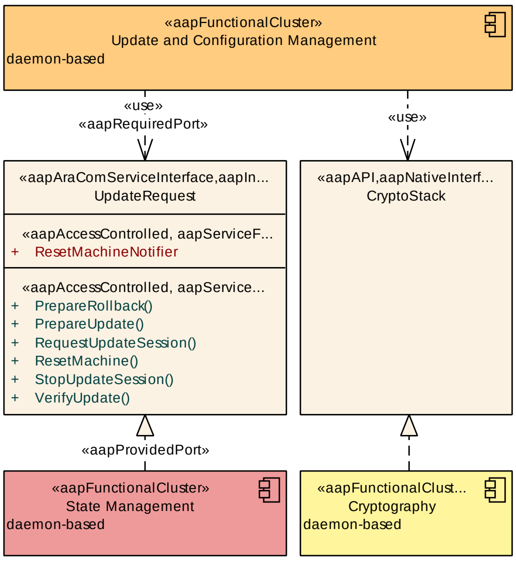

Figure 9.95: Interfaces required by Update and Configuration Management (part 3)

Interface	*Purpose*
ManifestAccessor	Update and Configuration Management shall use this interface to read information about its configuration from the Manifests.
Cryptography::CryptoStack	This interface may be used e.g., to verify the integrity and authenticity of Software Packages.
Execution Management::ExecutionClient	This interface shall be used by the daemon process(es) inside Update and Configuration Management to report their execution state to Execution Management.
Log and Trace::Logger	Update and Configuration Management shall use this interface to log standardized messages.
Persistency::FileStorage	Used to store files of received software packages.
Persistency::KeyValueStorage	Used to store the internal state of Update and Configuration Management.
Platform Health Management::SupervisedEntity	This interface should be used to supervise the daemon process(es) of Update and Configuration Management.
State Management::UpdateRequest	This interface is used to interact with State Management of the Adaptive Platform during an update.
Time Synchronization::SynchronizedTimeBase Consumer	Update and Configuration Management shall use this interface to get latest timestamp.

Table 9.30: Interfaces required by Update and Configuration Management

9.7.2 Vehicle Update and Configuration Management

Name:	Vehicle Update and Configuration Management
Short Name:	vucm
Category:	Configuration
Daemon-based:	Yes
Responsibilities:	Vehicle Update and Configuration Management is responsible for updating, installing, removing and keeping a record of the software installed in an entire vehicle in a safe and secure way. Hereby, Vehicle Update and Configuration Management enables to update the software and its configuration flexibly through over-the-air updates (OTA).

9.7.2.1 Defined interfaces

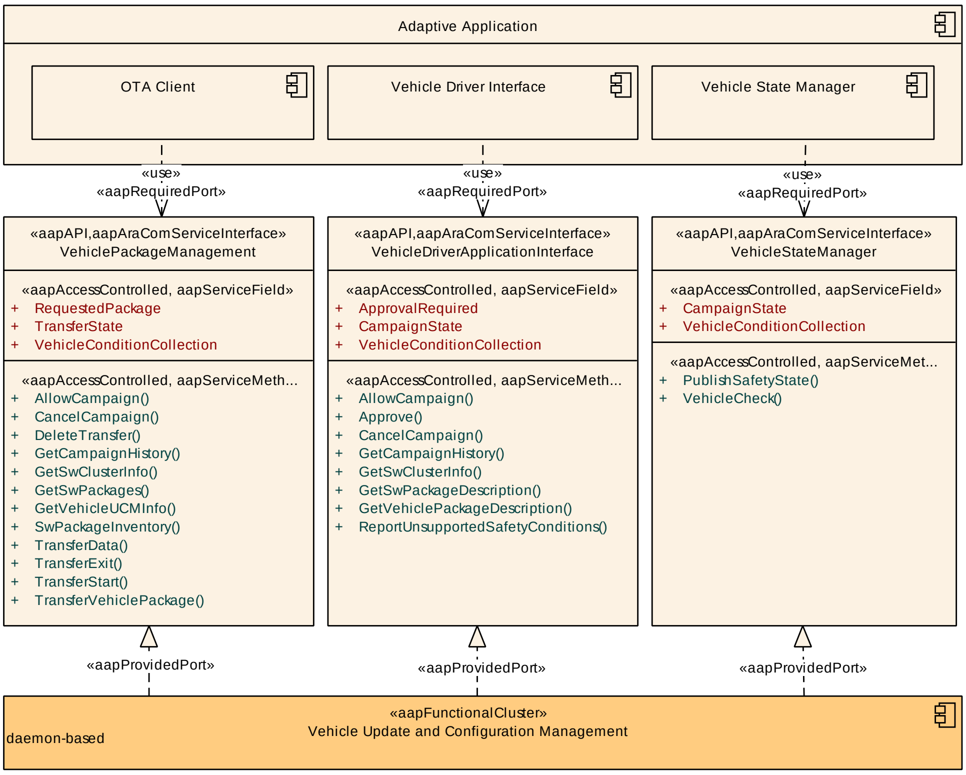

Figure 9.96: Interfaces of Vehicle Update and Configuration Management

Name:	VehiclePackageManagement	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface provides functionality for managing and transferring Vehicle Packages and Software Packages to Vehicle Update and Configuration Management.	
Operations:	AllowCampaign	Allows a new campaign to start.
	CancelCampaign	Aborts an ongoing campaign processing of a Vehicle Package.
	DeleteTransfer	Delete a transferred Software Package or Vehicle Package.
	GetCampaignHistory	Retrieve all actions that have been performed by Vehicle Update and Configuration Management.
	GetSwClusterInfo	Get a list of SoftwareClusters that are in state kPresent.
	GetSwPackages	Get the Software Packages that are part of current campaign handled by Vehicle Update and Configuration Management.
	GetVehicleUCMInfo	Get Vehicle Update and Configuration Management version and a list of Update and Configuration Management identifiers and versions.
	SwPackageInventory	Performs an inventory of all Software Packages.

△

	TransferData	Block-wise transfer of a Software Package or Vehicle Package to Vehicle Update and Configuration Management.
	TransferExit	Finish the transfer of a Software Package or Vehicle Package to Vehicle Update and Configuration Management.
	TransferStart	Start the transfer of a Software Package to Vehicle Update and Configuration Management.
	TransferVehiclePackage	Start the transfer of a Vehicle Package to Vehicle Update and Configuration Management.
Fields:	RequestedPackage	Software Package to be transferred to Vehicle Update and Configuration Management.
	TransferState	The current status of a campaign from an OTA Client perspective.
	VehicleConditionCollection	A set of safety conditions along with their corresponding states.

Name:	VehicleDriverApplicationInterface	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface provides functionality to interact with the vehicle driver, for example to approve updates.	
Operations:	AllowCampaign	Allow a new campaign to start.
	Approve	Inform Vehicle Update and Configuration Management of the driver's notification resolution.
	CancelCampaign	Aborts an ongoing campaign processing of a Vehicle Package.
	GetCampaignHistory	Retrieve all actions that have been performed by Vehicle Update and Configuration Management.
	GetSwClusterInfo	Get general information of the Software Clusters present in the Adaptive Platform.
	GetSwPackageDescription	Get the general information of the Software Packages that are part of current campaign handled by Vehicle Update and Configuration Management.
	GetVehiclePackageDescription	Get the Vehicle Package metadata relevant for communication with driver according to UN ECE 156.
	ReportUnsupportedSafetyConditions	Inform Vehicle Update and Configuration Management on all unsupported safety conditions.
Fields:	ApprovalRequired	Flag to inform an Adaptive Application if approval from a driver is required at current state based on the Vehicle Package Manifest.
	CampaignState	The status of the current campaign.
	VehicleConditionCollection	A set of safety conditions along with their corresponding states.

Name:	VehicleStateManager
Technology:	ara::com service interface
Usage:	Public API

▽

△

Description:	This interface provides functionality for a Vehicle State Manager Adaptive Application to inform Vehicle Update and Configuration Management about the safety state and policy of the vehicle.	
Operations:	PublishSafetyState	Called by the Vehicle State Manager Adaptive Application when safety state is changed.
	VehicleCheck	Called by the Vehicle State Manager Adaptive Application to inform Vehicle Update and Configuration Management of the vehicle check resolution.
Fields:	CampaignState	The status of the current campaign.
	VehicleConditionCollection	A set of safety conditions along with their corresponding computed states.

9.7.2.2 Provided interfaces

Vehicle Update and Configuration Management does not provide any interfaces to other Functional Clusters.

9.7.2.3 Required interfaces

Figures 9.97 and 9.98 show the interfaces that are required by Vehicle Update and Configuration Management from Functional Clusters on the local Machine.

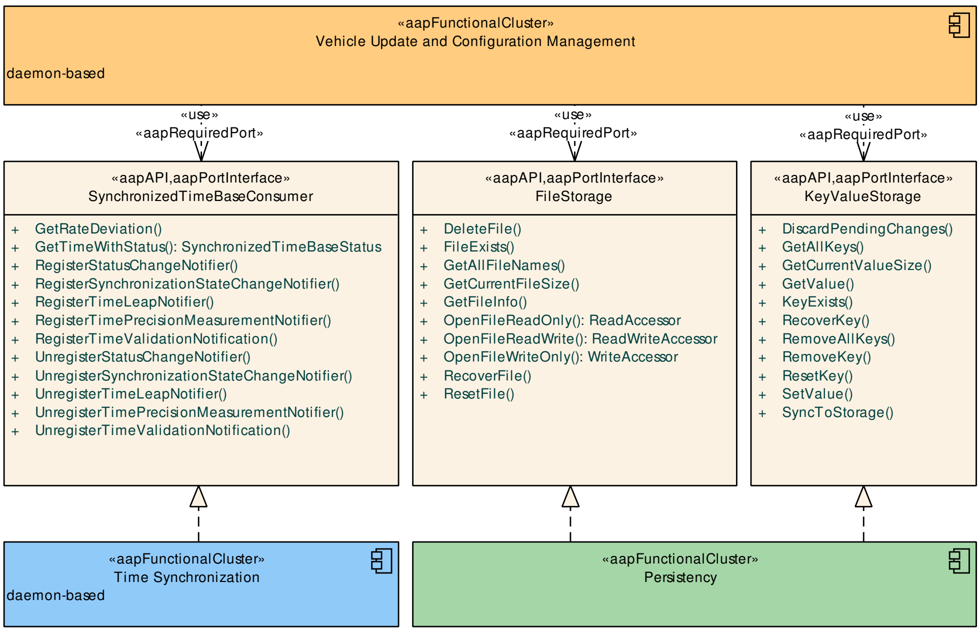

Figure 9.97: Interfaces required by Vehicle Update and Configuration Management (part 1)

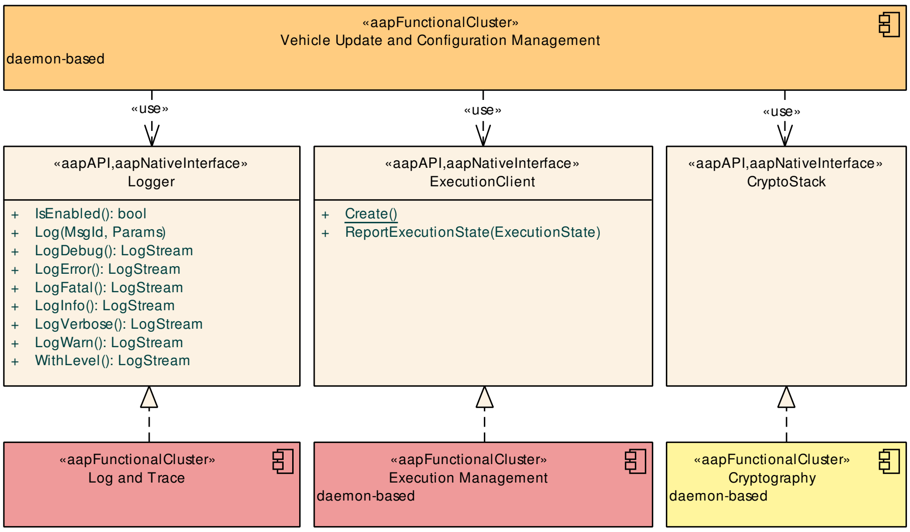

Figure 9.98: Interfaces required by Vehicle Update and Configuration Management (part 2)

Figure 9.99 shows the interfaces that are required by `Vehicle Update and Configuration Management` from `Update and Configuration Management` instances on the local and remote `Machine`(s) or `Flashing Adapter`s that provide the same service for third-party systems.

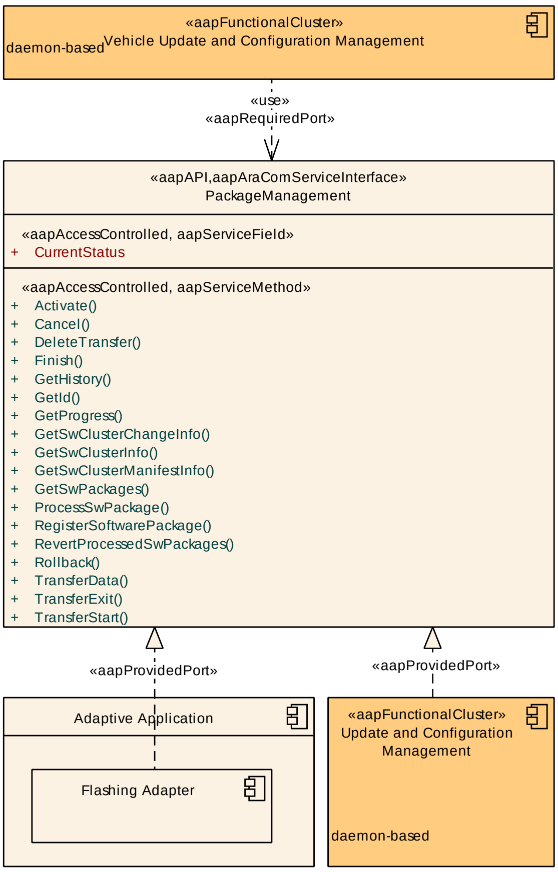

Figure 9.99: Interfaces required by Vehicle Update and Configuration Management (part 3)

Interface	Purpose
ManifestAccessor	Vehicle Update and Configuration Management shall use this interface to read information about its configuration from the Manifests.
Cryptography::CryptoStack	This interface may be used e.g., to verify the integrity and authenticity of Vehicle Packages.
Execution Management::ExecutionClient	This interface shall be used by the daemon process(es) inside Vehicle Update and Configuration Management to report their execution state to Execution Management.
Log and Trace::Logger	Vehicle Update and Configuration Management shall use this interface to log standardized messages.
Persistency::FileStorage	Used to store files of received vehicle packages.
Persistency::KeyValueStorage	Used to store the internal state of Vehicle Update and Configuration Management.
Time Synchronization::SynchronizedTimeBase Consumer	Vehicle Update and Configuration Management shall use this interface to get latest timestamp.
Update and Configuration Management:: PackageManagement	This interface is used to control different Update and Configuration Management instances and e.g., applications implementing the same interface located within the vehicle that act as an adapter to install software packages on third-party systems. Vehicle Update and Configuration Management is able to differentiate between the service instances by matching the result of GetId with an ID provided in a Software Package.

Table 9.31: Interfaces required by Vehicle Update and Configuration Management

9.7.3 Registry

Name:	Registry
Short Name:	n/a
Category:	Configuration
Daemon-based:	No
Responsibilities:	Registry is an internal component of the AUTOSAR Adaptive Platform that provides access information stored in the Manifest. It is not a standardized Functional Cluster intended to be used by Adaptive Applications.

9.7.3.1 Defined interfaces

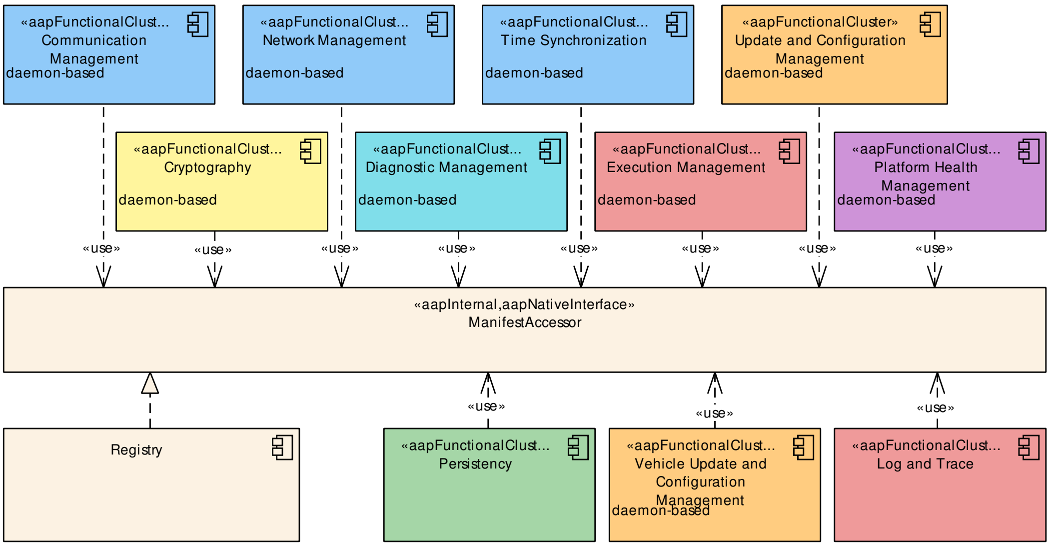

Figure 9.100: Interfaces of Registry

Name:	ManifestAccessor
Technology:	Native interface
Usage:	Internal
Description:	This interface provides functionality to read information that was modeled in the `Manifest`(s). This interface is not detailed in this document.

9.7.3.2 Provided interfaces

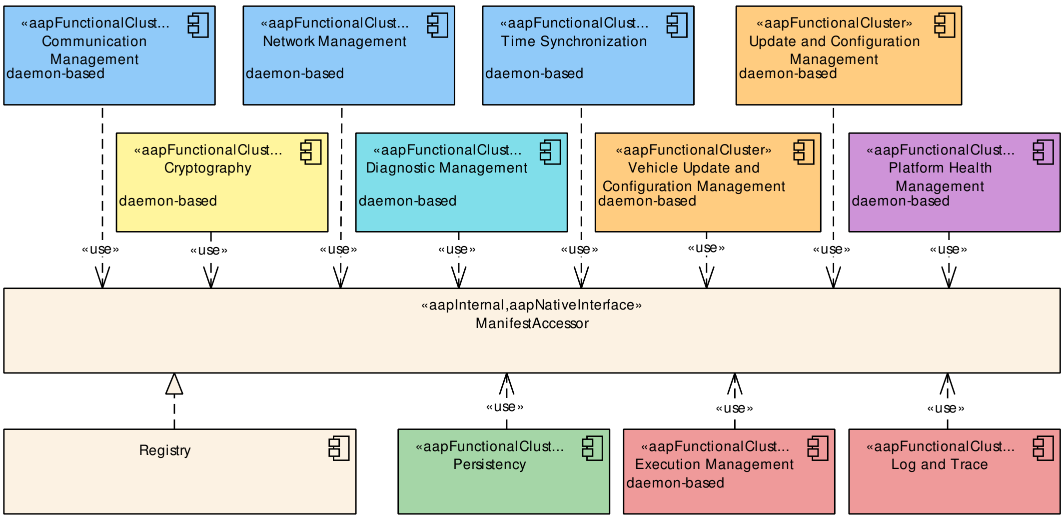

Figure 9.101: Users of the Registry interfaces

Interface	*Requiring functional clusters*
ManifestAccessor	Communication Management
	Cryptography
	Diagnostic Management
	Execution Management
	Log and Trace
	Network Management
	Persistency
	Platform Health Management
	Raw Data Stream
	Time Synchronization
	Update and Configuration Management
	Vehicle Update and Configuration Management

Table 9.32: Interfaces provided by Registry to other Functional Clusters

9.7.3.3 Required interfaces

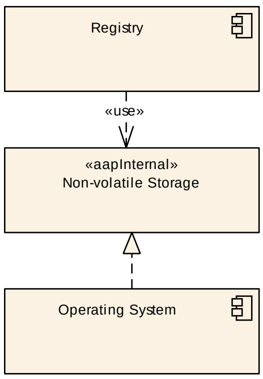

Figure 9.102: Interfaces required by Registry

Interface	*Purpose*
Non-volatile Storage	`Registry` uses this interface to read the information from the `Manifest`(s).

Table 9.33: Interfaces required by Registry

9.8 Diagnostics

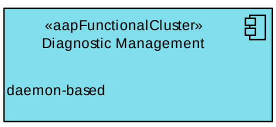

Figure 9.103: Functional Clusters in category Diagnostics

9.8.1 Diagnostic Management

Name:	Diagnostic Management
Short Name:	diag
Category:	Diagnostics
Daemon-based:	Yes
Responsibilities:	Diagnostic Management is responsible for handling diagnostic events produced by the individual Processes running in an AUTOSAR Runtime for Adaptive Applications. Diagnostic Management stores such events and the associated data persistently according to rendition policies. Diagnostic Management also provides access to diagnostic data for external Diagnostic Clients via standardized network protocols (ISO 14229-5 (UDSonIP) which is based on the ISO 14229-1 (UDS) and ISO 13400-2 (DoIP)).

9.8.1.1 Defined interfaces

9.8.1.1.1 Common interfaces

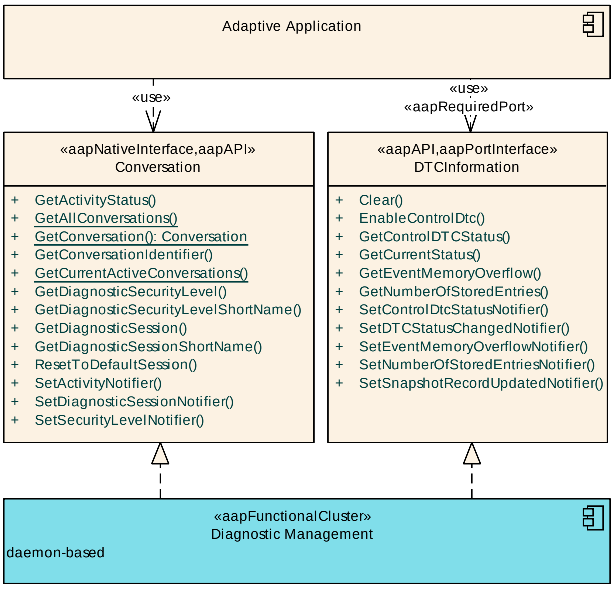

Figure 9.104: Common interfaces of Diagnostic Management (1 of 3)

Name:	Conversation	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface provides functionality to handle diagnostic conversations.	
Operations:	GetActivityStatus	Represents the status of an active conversation.
	GetAllConversations	Get all possible conversations.
	GetConversation	Get one conversation based on given meta information.
	GetConversationIdentifier	Getter for the current identification properties of the active conversation.
	GetCurrentActiveConversations	Get all currently active conversations.

▽

△

	GetDiagnosticSecurityLevel	Represents the current active diagnostic `SecurityLevel` of an active conversation.
	GetDiagnosticSecurityLevelShortName	Converts the given diagnostic `SecurityLevel` into the `ShortName`.
	GetDiagnosticSession	Represents the current active diagnostic session of an active conversation.
	GetDiagnosticSessionShortName	Converts the given diagnostic session into the `ShortName`.
	ResetToDefaultSession	Method to reset the current session to the default session.
	SetActivityNotifier	Register a notifier function which is called if the activity is changed.
	SetDiagnosticSessionNotifier	Register a notifier function which is called if the Session is changed.
	SetSecurityLevelNotifier	Register a notifier function which is called if the `SecurityLevel` is changed.

Name:	DTCInformation	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`DiagnosticDTCInformationInterface`	
Usage:	Public API	
Description:	This interface provides operations on DTC information per configured `DiagnosticMemory` destination.	
Operations:	Clear	Method for Clearing a DTC or a group of DTCs.
	EnableControlDtc	Enforce restoring ControlDTCStatus setting to enabled in case the monitor has some conditions or states demands to do so.
	GetControlDTCStatus	Contains the current status of the ControlDTCStatus.
	GetCurrentStatus	Retrieves the current UDS DTC status byte of the given DTC identifier.
	GetEventMemoryOverflow	Contains the current event memory overflow status.
	GetNumberOfStoredEntries	Contains the number of currently stored fault memory entries.
	SetControlDtcStatusNotifier	Registers a notifier function which is called if the control DTC setting is changed.
	SetDTCStatusChangedNotifier	Register a notifier function which is called if a UDS DTC status is changed.
	SetEventMemoryOverflowNotifier	Register a notifier function which is called if the current event memory overflow status changed.
	SetNumberOfStoredEntriesNotifier	Register a notifier function which is called if the number of currently stored fault memory entries changed.
	SetSnapshotRecordUpdatedNotifier	Register a notifier function which is called if the SnapshotRecord is changed.

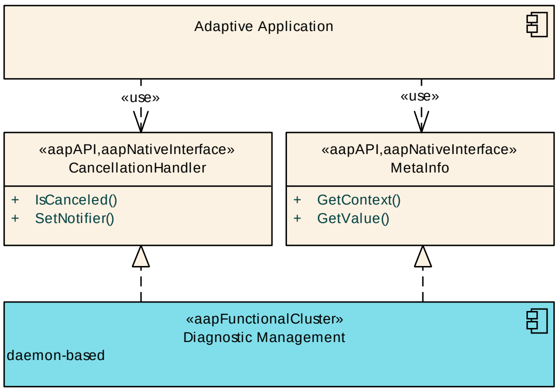

Figure 9.105: Common interfaces of Diagnostic Management (2 of 3)

Name:	CancellationHandler	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface holds a shared state if the processing should be canceled.	
Operations:	IsCanceled	Returns true in if the diagnostic service execution is canceled.
	SetNotifier	Registers a notifier function which is called if the diagnostic service execution is canceled.

Name:	MetaInfo	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface specifies a mechanism to provide meta information, i.e. from transport protocol layer, to an interested application.	
Operations:	GetContext	Get the context of the invocation.
	GetValue	Get the metainfo value for a given key.

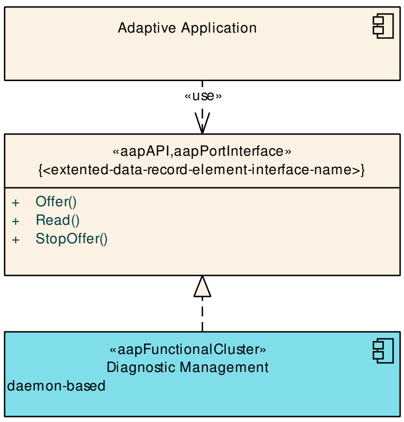

Figure 9.106: Common interfaces of Diagnostic Management (3 of 3)

Name:	{<extented-data-record-element-interface-name>}	
Technology:	Port interface	
Generated:	Yes	
Meta-model interface type:		
Usage:	Public API	
Description:	DiagnosticExtendedDataRecordInterface class	
Operations:	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	Read	Provision of a Read ExtendedDataRecordElement.
	StopOffer	Disables forwarding of request messages from `Diagnostic Management`.

9.8.1.1.2 Interfaces for request handling

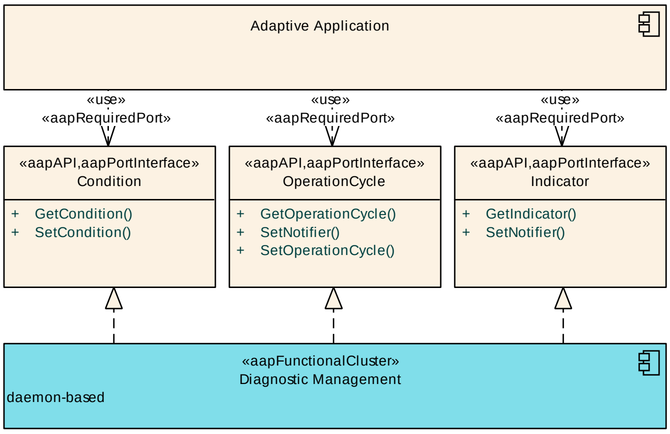

Figure 9.107: Interfaces for request handling

Name:	Condition	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`DiagnosticConditionInterface`	
Usage:	Public API	
Description:	This interface provides functionality for condition management.	
Operations:	GetCondition	Get the current condition.
	SetCondition	Set the current condition.

Name:	OperationCycle
Technology:	Port interface
Generated:	No
Meta-model interface type:	`DiagnosticOperationCycleInterface`

▽

△

Usage:	Public API	
Description:	This interface provides functionality for handling of operation cycles.	
Operations:	GetOperationCycle	Get the current `OperationCycle`.
	SetNotifier	Registers a notifier function which is called if the `OperationCycle` is changed.
	SetOperationCycle	Set the current `OperationCycle`.

Name:	Indicator	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`DiagnosticIndicatorInterface`	
Usage:	Public API	
Description:	This interface provides functionality for handling indicators.	
Operations:	GetIndicator	Get current Indicator.
	SetNotifier	Register a notifier function which is called if the indicator is updated.

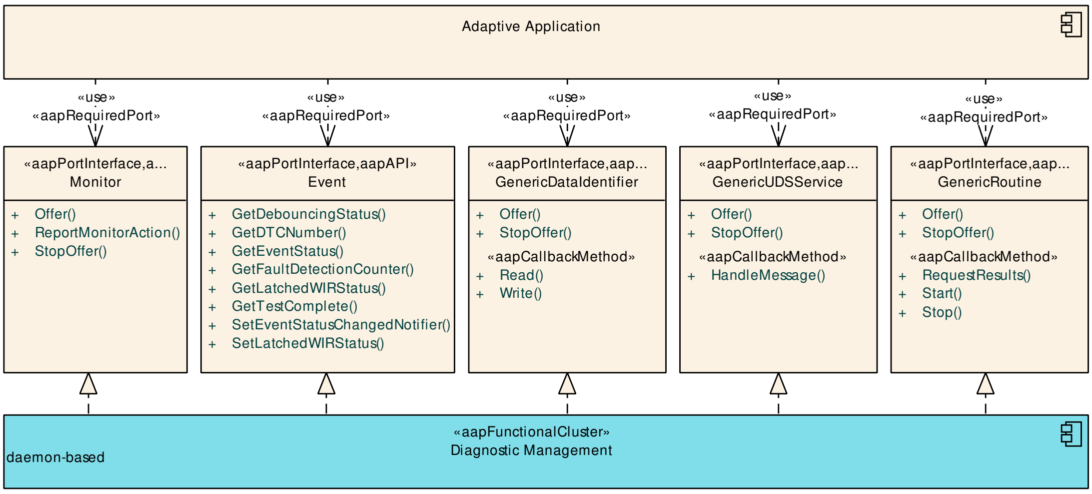

Figure 9.108: Interfaces for generic request handling

Name:	Monitor	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`DiagnosticMonitorInterface`	
Usage:	Public API	
Description:	This interface provides functionality to report qualified and unqualified test results and to control debouncing options.	
Operations:	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	ReportMonitorAction	Report the status information being relevant for error monitoring paths.

▽

△

	StopOffer	Disable forwarding of request messages from Diagnostic Management.

Name:	Event	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	DiagnosticEventInterface	
Usage:	Public API	
Description:	This interface defines functionality for diagnostic events.	
Operations:	GetDTCNumber	Returns the DTC-ID related to this event instance.
	GetDebouncingStatus	Get the current debouncing status.
	GetEventStatus	Returns the current diagnostic event status.
	GetFaultDetectionCounter	Returns the current value of Fault Detection Counter of this event.
	GetLatchedWIRStatus	Returns the current warning indicator status.
	GetTestComplete	Get the status if the event has matured to test completed (corresponds to FDC = -128 or FDC = 127).
	SetEventStatusChangedNotifier	Register a notifier function which is called if a diagnostic event is changed.
	SetLatchedWIRStatus	Set the warning indicator status.

Name:	GenericDataIdentifier	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	DiagnosticDataIdentifierGenericInterface	
Usage:	Public API	
Description:	Generic interface to handle ReadDataByIdentifier and WriteDataByIdentifier requests.	
Operations:	Offer	Enable forwarding of request messages from Diagnostic Management.
	Read	Called for a ReadDataByIdentifier request for this DiagnosticDataIdentifier.
	StopOffer	Disable forwarding of request messages from Diagnostic Management.
	Write	Called for a WriteDataByIdentifier request for this DiagnosticDataIdentifier.

Name:	GenericUDSService	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	DiagnosticGenericUdsInterface	
Usage:	Public API	
Description:	Generic interface to handle UDS messages.	
Operations:	*HandleMessage*	Handles an UDS request message.

▽

	Offer	Enable forwarding of request messages from Diagnostic Management.
	StopOffer	Disable forwarding of request messages from Diagnostic Management.

Name:	GenericRoutine	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`DiagnosticRoutineGenericInterface`	
Usage:	Public API	
Description:	Generic interface to handle `RoutineControl` requests.	
Operations:	Offer	Enable forwarding of request messages from Diagnostic Management.
	RequestResults	Called for `RoutineControl` with `SubFunction RequestResults` request for this `DiagnosticRoutineIdentifier`.
	Start	Called for `RoutineControl` with `SubFunction Start` request for this `DiagnosticRoutineIdentifier`.
	Stop	Called for `RoutineControl` with `SubFunction Stop` request for this `DiagnosticRoutineIdentifier`.
	StopOffer	Disable forwarding of request messages from Diagnostic Management.

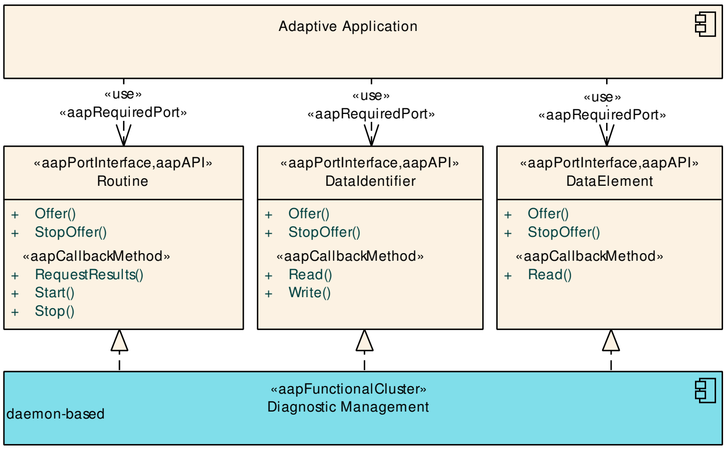

Figure 9.109: Generated interfaces for request handling

Name:	DataIdentifier
Technology:	Port interface
Generated:	Yes
Meta-model interface type:	`DiagnosticDataIdentifierInterface`
Usage:	Public API
Description:	Generated interface to handle `ReadDataByIdentifier` and `WriteDataByIdentifier` requests.

△

Operations:	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	Read	Called for a `ReadDataByIdentifier` request for this `DiagnosticDataIdentifier`.
	StopOffer	Disable forwarding of request messages from `Diagnostic Management`.
	Write	Called for a `WriteDataByIdentifier` request for this `DiagnosticDataIdentifier`.

Name:	DataElement	
Technology:	Port interface	
Generated:	Yes	
Meta-model interface type:	`DiagnosticDataElementInterface`	
Usage:	Public API	
Description:	Generated interface to handle read requests for `DataElement`s.	
Operations:	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	Read	Called for reading a `DataElement`.
	StopOffer	Disable forwarding of request messages from `Diagnostic Management`.

Name:	Routine	
Technology:	Port interface	
Generated:	Yes	
Meta-model interface type:	`DiagnosticRoutineInterface`	
Usage:	Public API	
Description:	Generated interface to handle `RoutineControl` requests.	
Operations:	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	RequestResults	Called for `RoutineControl` with `SubFunction RequestResults` request for this `DiagnosticRoutineIdentifier`.
	Start	Called for `RoutineControl` with `SubFunction Start` request for this `DiagnosticRoutineIdentifier`.
	Stop	Called for `RoutineControl` with `SubFunction Stop` request for this `DiagnosticRoutineIdentifier`.
	StopOffer	Disable forwarding of request messages from `Diagnostic Management`.

9.8.1.1.3 Interfaces for security

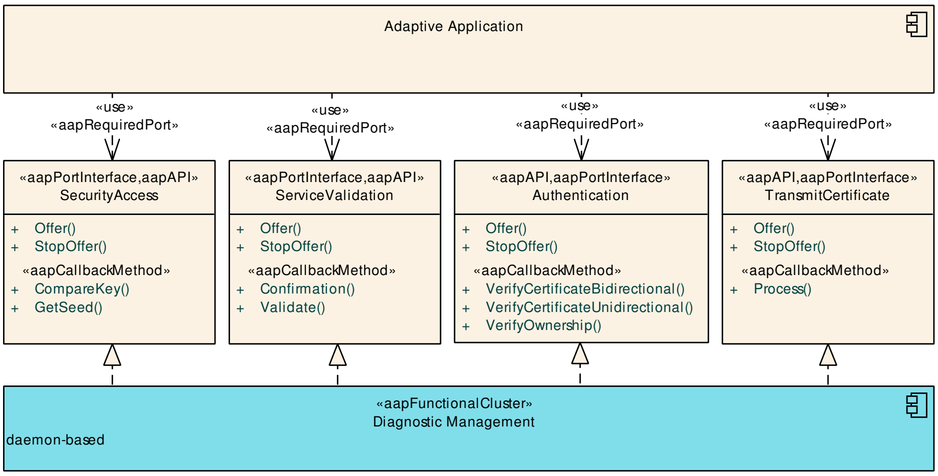

Figure 9.110: Interfaces for security

Name:	SecurityAccess	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`DiagnosticSecurityLevelInterface`	
Usage:	Public API	
Description:	This interface provides functionality for handling `SecurityAccess` requests.	
Operations:	*CompareKey*	This method is called, when a diagnostic request has been finished, to notify about the outcome.
	GetSeed	Called for any request message.
	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	StopOffer	Disable forwarding of request messages from `Diagnostic Management`.

Name:	ServiceValidation	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`DiagnosticServiceValidationInterface`	
Usage:	Public API	
Description:	This interface provides functionality for handling `ServiceValidation` requests.	
Operations:	*Confirmation*	This method is called, when a diagnostic request has been finished, to notify about the outcome.
	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	StopOffer	Disable forwarding of request messages from `Diagnostic Management`.
	Validate	Called for any request message.

Name:	Authentication	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`DiagnosticAuthenticationInterface`	
Usage:	Public API	
Description:	Interface to handle `Authentication` requests.	
Operations:	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	StopOffer	Disable forwarding of request messages from `Diagnostic Management`.
	VerifyCertificateBidirectional	This function accepts the certificate and challenge received from the tester, verifies the certificate, and creates a challenge, ephemeral Public Key and Proof Of Ownership that must be returned to the tester. The function also returns the server certificate that will be used by the tester to verify the Proof Of Ownership.
	VerifyCertificateUnidirectional	This function accepts the certificate received from the tester, verifies it, and creates a challenge and ephemeral public key that must be returned to the tester.
	VerifyOwnership	This function accepts the Proof Of Ownership received from the tester and verifies it with the Public Key of the certificate received in the `VerifyCertificateUnidirectional` / `VerifyCertificateBidirectional` against the server challenge created in the last call to `VerifyCertificateUnidirectional` / `VerifyCertificateBidirectional`.

Name:	TransmitCertificate	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`DiagnosticTransmitCertificateInterface`	
Usage:	Public API	
Description:	Interface to handle subfunction `TransmitCertificate` of UDS service `Authentication`.	
Operations:	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	Process	Process a certificate sent by a diagnostic tester.
	StopOffer	Disable forwarding of request messages from `Diagnostic Management`.

9.8.1.1.4 Interfaces for Upload and Download

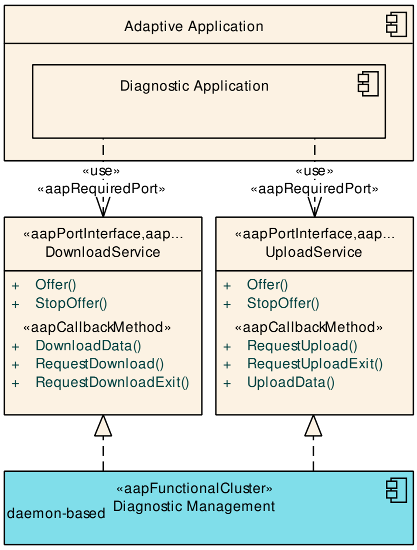

Figure 9.111: Interfaces for Upload and Download

Name:	UploadService	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	DiagnosticUploadInterface	
Usage:	Public API	
Description:	Upload service interface.	
Operations:	Offer	Enable forwarding of request messages from Diagnostic Management.
	RequestUpload	Called for RequestUpload.
	RequestUploadExit	Called for RequestTransferExit.
	StopOffer	Disable forwarding of request messages from Diagnostic Management.
	UploadData	Called for TransferData following a previous RequestUpload.

Name:	DownloadService	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	DiagnosticDownloadInterface	
Usage:	Public API	
Description:	Download service interface.	
Operations:	*DownloadData*	Called for TransferData following a previous RequestDownload.
	Offer	Enable forwarding of request messages from Diagnostic Management.
	RequestDownload	Called for RequestDownload.

▽

	RequestDownloadExit	Called for RequestTransferExit.
	StopOffer	Disable forwarding of request messages from Diagnostic Management.

9.8.1.1.5 Interfaces for Managing Multiple Conditions, Events, and Monitors

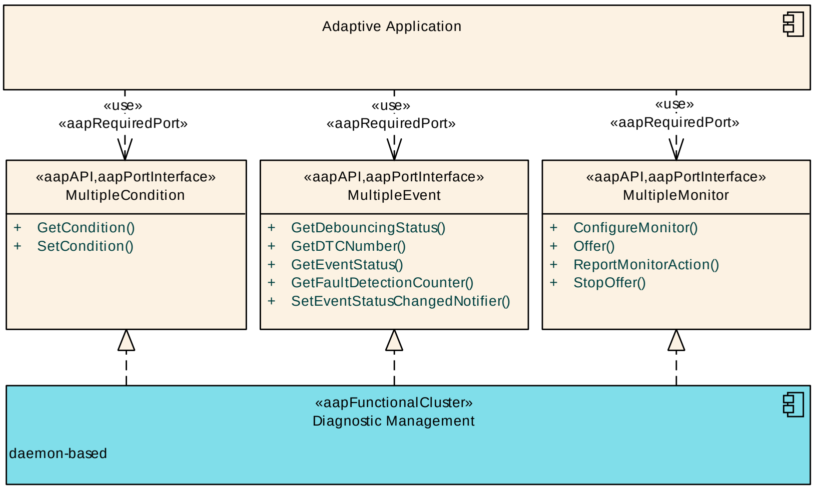

Figure 9.112: Interfaces for Managing Multiple Conditions, Events, and Monitors

Name:	MultipleCondition	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	DiagnosticMultipleConditionInterface	
Usage:	Public API	
Description:	Interface for multiple condition operations.	
Operations:	GetCondition	Get condition state of the requested condition.
	SetCondition	Set condition state of the requested condition.

Name:	MultipleEvent	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	DiagnosticMultipleEventInterface	
Usage:	Public API	
Description:	Interface to implement operations on diagnostic Events with the multiple event interfaces.	
Operations:	GetDTCNumber	Returns the DTC-ID related to requested event.
	GetDebouncingStatus	Get the current debouncing status of the requested event.
	GetEventStatus	Returns the current status of the requested event.

△

	GetFaultDetectionCounter	Returns the current value of Fault Detection Counter of the selected event.
	SetEventStatusChangedNotifier	Register a notifier function which is called if a diagnostic event is changed. A consecutive call of this method will overwrite the previous registered notifier.

Name:	MultipleMonitor	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`DiagnosticMultipleMonitorInterface`	
Usage:	Public API	
Description:	Interface to implement monitor operations on the multiple monitor interfaces.	
Operations:	ConfigureMonitor	Configures a monitor with counter-based debouncing. Any overloaded ConfigureMonitor shall be only called once per monitor. The monitor can only be used after this initialization step.
	Offer	Enable forwarding request messages from `Diagnostic Management` to this handler.
	ReportMonitorAction	Function to report the status information being relevant for error monitoring paths. Calls with invalid monitor Handle are ignored.
	StopOffer	Disable forwarding request messages from `Diagnostic Management` to this handler.

9.8.1.1.6 Interfaces for ECU Reset

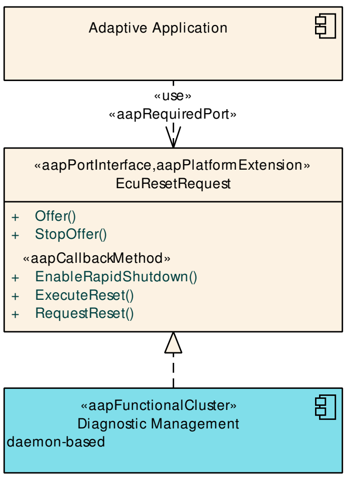

Figure 9.113: Interfaces for ECU Reset

Name:	EcuResetRequest
Technology:	Port interface
Generated:	No

▽

△

Meta-model interface type:	`DiagnosticEcuResetInterface`	
Usage:	Platform extension	
Description:	Service EcuReset Request interface	
Operations:	*EnableRapidShutdown*	Interface for subFunction En-/DisableRapidShutdown
	ExecuteReset	Process which is instantiating this interface, has to execute the requested reset.
	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	RequestReset	Called for any EcuReset subFunction, except En-/DisableRapidShutdown.
	StopOffer	Disables forwarding of request messages from `Diagnostic Management`.

9.8.1.1.7 Interfaces for File Transfer

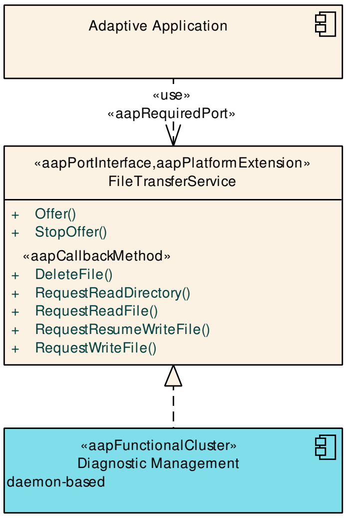

Figure 9.114: Interfaces for File Transfer

Name:	FileTransferService	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`DiagnosticRequestFileTransferInterface`	
Usage:	Platform extension	
Description:	File Transfer service interface	
Operations:	*DeleteFile*	Called for RequestFileTransfer with ModeOfOperation DeleteFile
	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	RequestReadDirectory	Called for RequestFileTransfer with ModeOfOperation ReadDir

▽

	RequestReadFile	Called for RequestFileTransfer with ModeOfOperation ReadFile
	RequestResumeWriteFile	Called for RequestFileTransfer with ModeOfOperation ResumeFile
	RequestWriteFile	Called for RequestFileTransfer with ModeOfOperation WriteFile
	StopOffer	Disable forwarding of request messages from `Diagnostic Management`.

9.8.1.1.8 Interfaces for UDS Transportlayer API

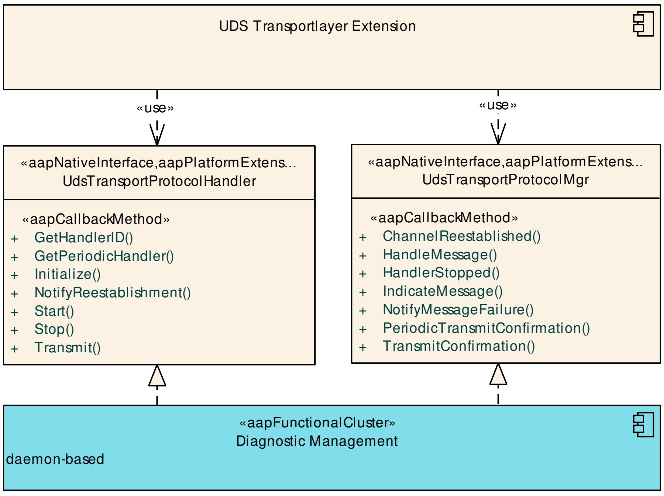

Figure 9.115: Interfaces for the UDS Transportlayer API (1 of 2)

Name:	UdsTransportProtocolHandler	
Technology:	Native interface	
Usage:	Platform extension	
Description:	This interface provides functionality for general Transport Protocol handling.	
Operations:	*GetHandlerID*	Return the `UdsTransportProtocolHandlerID`, which was given to the implementation during construction (ctor call).
	GetPeriodicHandler	Returns the corresponding periodic Transport Protocol handler.
	Initialize	Initializes the handler.
	NotifyReestablishment	Notify the `Diagnostic Management` core if the given channel has been re-established.
	Start	Start processing the implemented UDS Transport Protocol.
	Stop	Indicate that this instance should terminate.
	Transmit	Transmit a UDS message via the underlying UDS Transport Protocol channel.

Name:	UdsTransportProtocolMgr	
Technology:	Native interface	
Usage:	Platform extension	
Description:	This interface provides functionality to manage messages and their handling.	
Operations:	*ChannelReestablished*	Notification call from the given transport channel, that it has been reestablished since the last (Re)Start from the `UdsTransportProtocolHandler` to which this channel belongs.
	HandleMessage	Hands over a valid received UDS message (currently this is only a request type) from transport layer to session layer.
	HandlerStopped	Notification from handler, that it has stopped now (e.g. closed down network connections, freed resources, etc...)
	IndicateMessage	Indicates a message start.
	NotifyMessageFailure	Indicates, that the message indicated via `IndicateMessage()` has failure and will not lead to a final `HandleMessage()` call.
	PeriodicTransmitConfirmation	Confirmation of sent messages and number.
	TransmitConfirmation	Notification about the outcome of a transmit request called by core `Diagnostic Management` at the handler via `Transmit()`

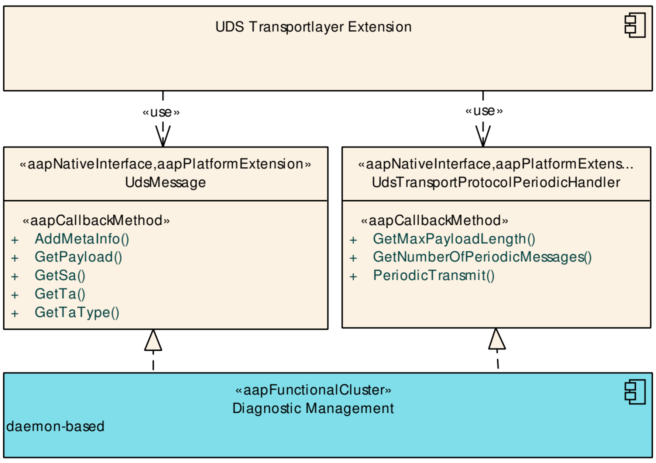

Figure 9.116: Interfaces for the UDS Transportlayer API (2 of 2)

Name:	UdsMessage	
Technology:	Native interface	
Usage:	Platform extension	
Description:	Represents an UDS message exchanged between `Diagnostic Management`'s generic core (`UdsTransportProtocolMgr`) and a specific implementation of `UdsTransportProtocolHandler` on diagnostic request reception path or diagnostic response transmission path.	
Operations:	*AddMetaInfo*	Called by the transport plugin to add channel specific meta-info.
	GetPayload	Get the UDS message data starting with the SID (A_Data as per ISO).
	GetSa	Get the source address of the UDS message.

	GetTa	Get the target address of the UDS message.
	GetTaType	Get the target address type (phys/func) of the UDS message.

Name:	UdsTransportProtocolPeriodicHandler	
Technology:	Native interface	
Usage:	Platform extension	
Description:	This interface provides functionality for Transport Protocol handling of periodic messages.	
Operations:	*GetMaxPayloadLength*	Reports the maximum payload length supported for a single periodic transmission on the channel.
	GetNumberOfPeriodicMessages	Reports the Transport Protocol implementation and connection specific number of periodic messages.
	PeriodicTransmit	Sends all the messages in the list in the given order.

9.8.1.1.9 Interfaces for DoIP API

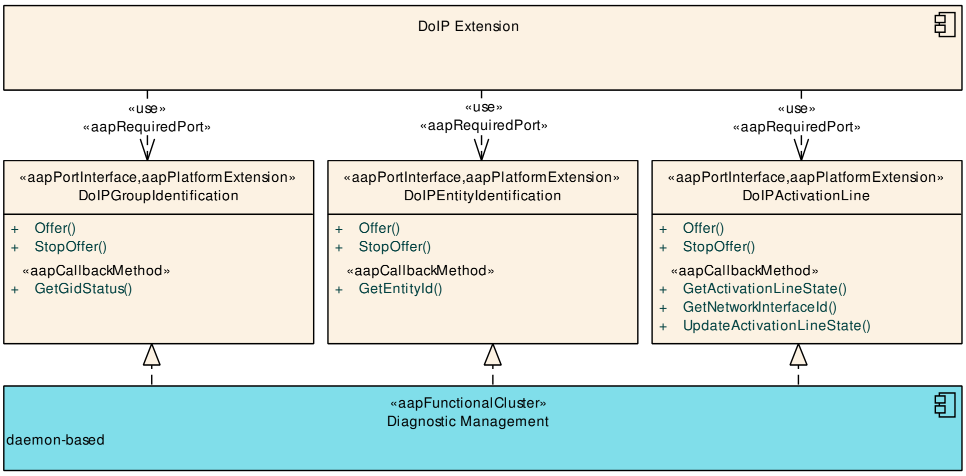

Figure 9.117: Interfaces for the DoIP API (1 of 2)

Name:	DoIPGroupIdentification	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`DiagnosticDoIPGroupIdentificationInterface`	
Usage:	Platform extension	
Description:	This interface provides functionality to get the GID state of the DoIP protocol.	
Operations:	*GetGidStatus*	Called to get the current GID state for the DoIP protocol.
	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	StopOffer	Disables forwarding of request messages from `Diagnostic Management`.

Name:	DoIPEntityIdentification	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:		
Usage:	Platform extension	
Description:		
Operations:	*GetEntityId*	Get the EID for the DoIP protocol.
	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	StopOffer	Disables forwarding of request messages from `Diagnostic Management`.

Name:	DoIPActivationLine	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`DiagnosticDoIPActivationLineInterface`	
Usage:	Platform extension	
Description:	This interface provides functionality to control the DoIP activation line.	
Operations:	*GetActivationLineState*	Get the current activation line state.
	GetNetworkInterfaceId	Get the network interface Id for which this instance is responsible.
	Offer	Enable provision of the activation line state to `Diagnostic Management`.
	StopOffer	Disable provision of the activation line state to `Diagnostic Management`.
	UpdateActivationLineState	Update current activation line state.

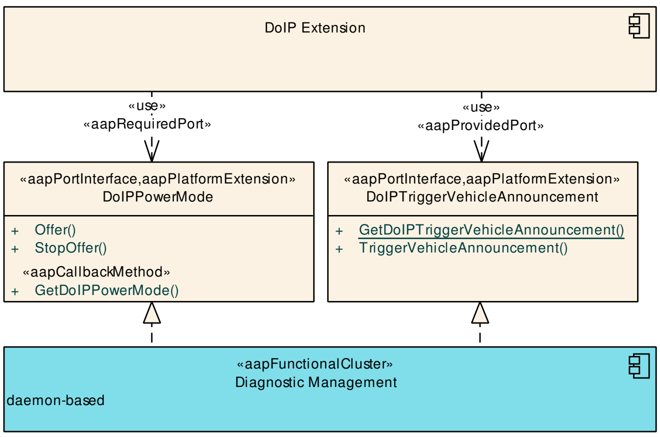

Figure 9.118: Interfaces for the DoIP API (2 of 2)

Name:	DoIPPowerMode
Technology:	Port interface
Generated:	No

△

Meta-model interface type:	`DiagnosticDoIPPowerModeInterface`	
Usage:	Platform extension	
Description:	This interface provides functionality to control the power mode via DoIP.	
Operations:	*GetDoIPPowerMode*	Called to get the current Power Mode for the DoIP protocol.
	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	StopOffer	Disable forwarding of request messages from `Diagnostic Management`.

Name:	DoIPTriggerVehicleAnnouncement	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`DiagnosticDoIPTriggerVehicleAnnouncementInterface`	
Usage:	Platform extension	
Description:	This interface provides functionality to trigger a vehicle announcement via DoIP.	
Operations:	GetDoIPTriggerVehicleAnnouncement	Get the `DoIPTriggerVehicleAnnouncement` interface.
	TriggerVehicleAnnouncement	Send out vehicle announcements on the given network interface Id.

9.8.1.1.10 Interfaces for SOVD data and operations

This chapter lists interfaces to realize SOVD data and SOVD operations.

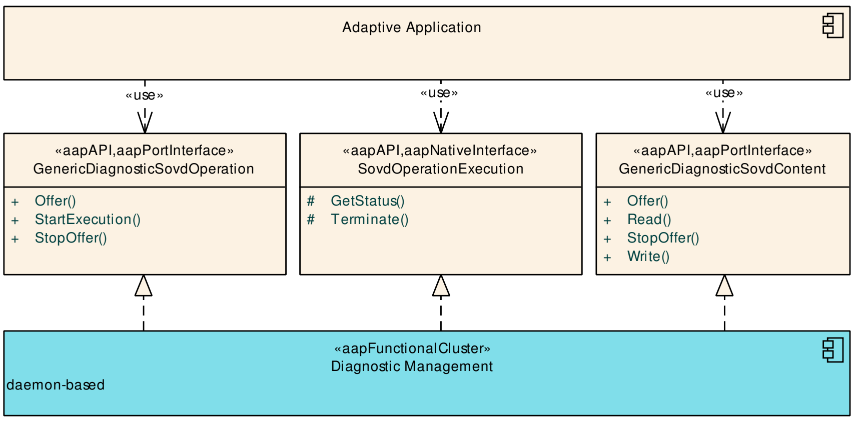

Figure 9.119: Interfaces for SOVD operations

Name:	GenericDiagnosticSovdOperation
Technology:	Port interface
Generated:	No

▽

△

Meta-model interface type:		
Usage:	Public API	
Description:	Generic Diagnostic SOVD Operation interface	
Operations:	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	StartExecution	Called for "Start execution of an operation" for the specified SOVD Operation
	StopOffer	Disables forwarding of request messages from `Diagnostic Management`.

Name:	SovdOperationExecution	
Technology:	Native interface	
Usage:	Public API	
Description:	Interface to handle the SOVD Operation Execution	
Operations:	*GetStatus*	Called for "Get status execution of an operation" for the current execution
	Terminate	Called for "Terminate execution of an operation" for the current execution

Name:	GenericDiagnosticSovdContent	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:		
Usage:	Public API	
Description:	Generic Diagnostic SOVD Content interface	
Operations:	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	Read	Called for "Read Single Data Value request" for the specified SOVD Content
	StopOffer	Disables forwarding of request messages from `Diagnostic Management`.
	Write	Called for "Write a Data Value" request for the specified SOVD Content

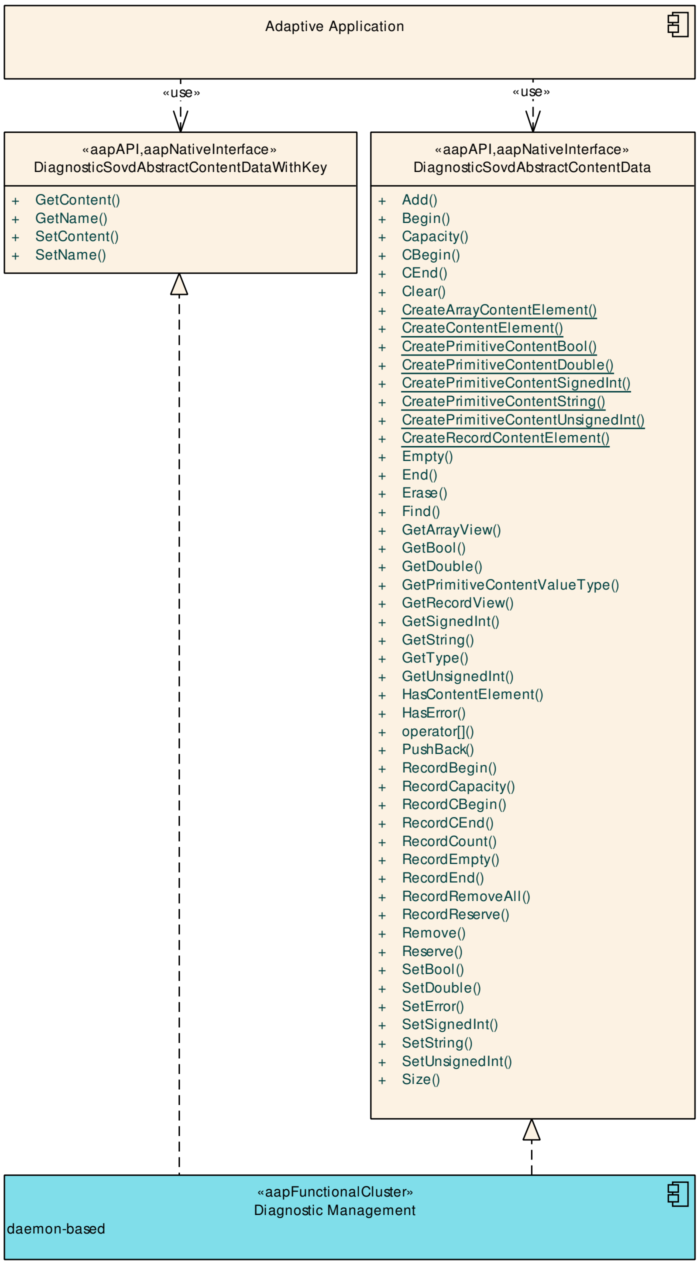

Figure 9.120: Interfaces for SOVD data

Name:	DiagnosticSovdAbstractContentDataWithKey	
Technology:	Native interface	
Usage:	Public API	
Description:	A DiagnosticSovdAbstractContentData with associated key	
Operations:	GetContent	Gets the associated content
	GetName	Gets the name of the key
	SetContent	Sets the associated content
	SetName	Sets the name of the key

Name:	DiagnosticSovdAbstractContentData	
Technology:	Native interface	
Usage:	Public API	
Description:	Diagnostic SOVD Abstract Content Data	
Operations:	Add	Adds an element with the specified key
	Begin	Returns the begin operator of an array
	CBegin	Returns the begin operator of an array
	CEnd	Returns the end operator of an array
	Capacity	Returns the capacity of the Array content element
	Clear	Clears the Array
	CreateArrayContentElement	Create an array content element
	CreateContentElement	Create a content
	CreatePrimitiveContentBool	Create primitive of type bool
	CreatePrimitiveContentDouble	Create primitive of type double
	CreatePrimitiveContentSignedInt	Create primitive of type signed integer
	CreatePrimitiveContentString	Create primitive of type string
	CreatePrimitiveContentUnsignedInt	Create primitive of type unsigned integer
	CreateRecordContentElement	Create a record content element
	Empty	Returns if the Array is empty
	End	Returns the end operator of an array
	Erase	Erases a content element from DiagnosticSovdContent ElementData
	Find	Find an element with the specified key
	GetArrayView	Get a view onto this DiagnosticSovdAbstractContent Data as an array
	GetBool	Get bool
	GetDouble	Get double
	GetPrimitiveContentValueType	Get the underlying value type of the primitive content
	GetRecordView	Get a view onto this DiagnosticSovdAbstractContent Data as a record
	GetSignedInt	Get signed integer
	GetString	Get string
	GetType	Get the underlying type of the abstract content
	GetUnsignedInt	Get unsigned integer
	HasContentElement	Returns if the RecordContentElementData contains an element with the specified key
	HasError	Returns if the abstract content contains an error
	PushBack	Pushes back a DiagnosticSovdContentElementData
	RecordBegin	Returns the begin operator of a RecordContentElement Data
	RecordCBegin	Returns the begin operator of a RecordContentElement Data
	RecordCEnd	Returns the end operator of a RecordContentElement Data
	RecordCapacity	Returns the capacity of the RecordContentElementData
	RecordCount	Returns the count of elements in the RecordContent ElementData

▽

△

	RecordEmpty	Returns if the RecordContentElementData is empty
	RecordEnd	Returns the end operator of a RecordContentElement Data
	RecordRemoveAll	Removes all the content element in the RecordContent ElementData
	RecordReserve	Reserves the capacity of the RecordContentElement Data
	Remove	Removes an element with the specified key
	Reserve	Reserves the capacity of the DiagnosticSovdArray ContentElementData
	SetBool	Set bool
	SetDouble	Set double
	SetError	Sets the abstract content to be erroneous
	SetSignedInt	Set signed integer
	SetString	Set string
	SetUnsignedInt	Set unsigned integer
	Size	Returns the size of the Array content element
	operator[]	Access the content element with specified index of an array

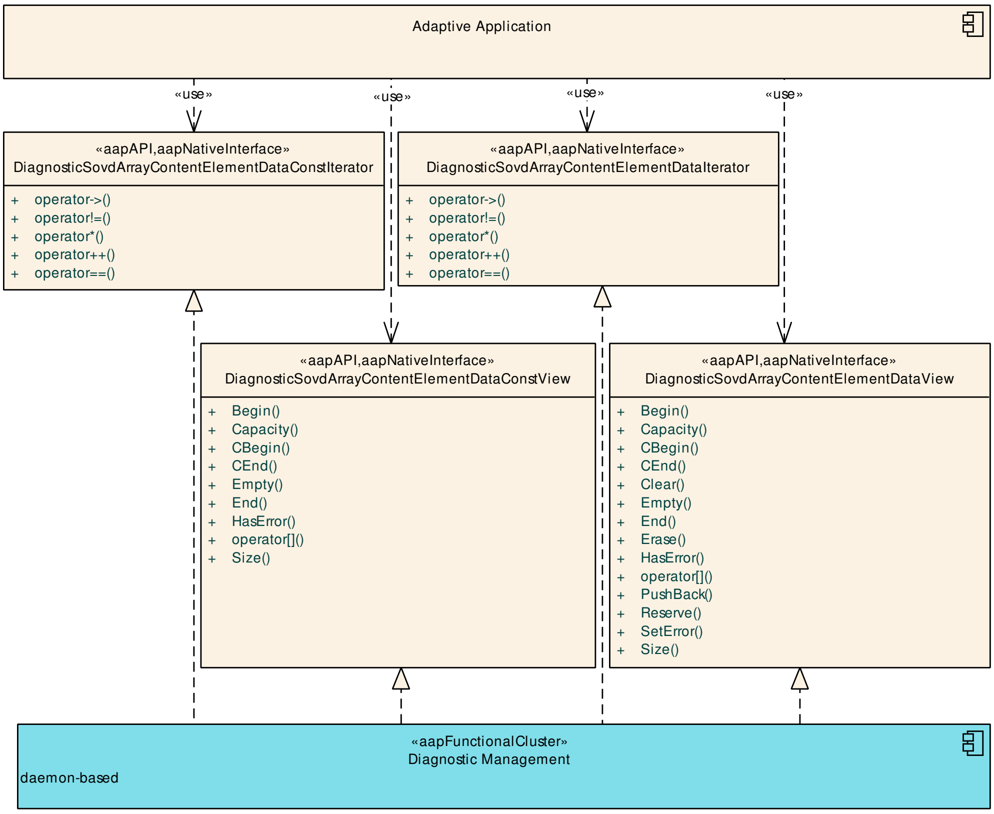

Figure 9.121: Interfaces for SOVD array data

Name:	DiagnosticSovdArrayContentElementDataConstIterator	
Technology:	Native interface	
Usage:	Public API	
Description:	Const iterator for a DiagnosticSovdArrayContentElementData	
Operations:	operator!=	Not equal operator
	operator*	Dereference operator
	operator++	Increment operator
	operator->	Dereference operator
	operator==	Equal operator

Name:	DiagnosticSovdArrayContentElementDataConstView	
Technology:	Native interface	
Usage:	Public API	
Description:	Const view on a DiagnosticSovdAbstractContentData with Array type	
Operations:	Begin	Returns the begin operator of an array
	CBegin	Returns the begin operator of an array
	CEnd	Returns the end operator of an array
	Capacity	Returns the capacity of the Array content element
	Empty	Returns if the Array is empty
	End	Returns the end operator of an array
	HasError	Returns if the abstract content contains an error
	Size	Returns the size of the Array content element
	operator[]	Access the content element with specified index of an array

Name:	DiagnosticSovdArrayContentElementDataIterator	
Technology:	Native interface	
Usage:	Public API	
Description:	Iterator for a DiagnosticSovdArrayContentElementData	
Operations:	operator!=	Not equal operator
	operator*	Dereference operator
	operator++	Increment operator
	operator->	Dereference operator
	operator==	Equal operator

Name:	DiagnosticSovdArrayContentElementDataView	
Technology:	Native interface	
Usage:	Public API	
Description:	View on a DiagnosticSovdAbstractContentData with Array type	
Operations:	Begin	Returns the begin operator of an array
	CBegin	Returns the begin operator of an array
	CEnd	Returns the end operator of an array
	Capacity	Returns the capacity of the Array content element

▽

△

	Clear	Clears the Array
	Empty	Returns if the Array is empty
	End	Returns the end operator of an array
	Erase	Erases a content element from DiagnosticSovdContent ElementData
	HasError	Returns if the abstract content contains an error
	PushBack	Pushes back a DiagnosticSovdContentElementData
	Reserve	Reserves the capacity of the DiagnosticSovdArray ContentElementData
	SetError	Sets the array content to be erroneous.
	Size	Returns the size of the Array content element
	operator[]	Access the content element with specified index of an array

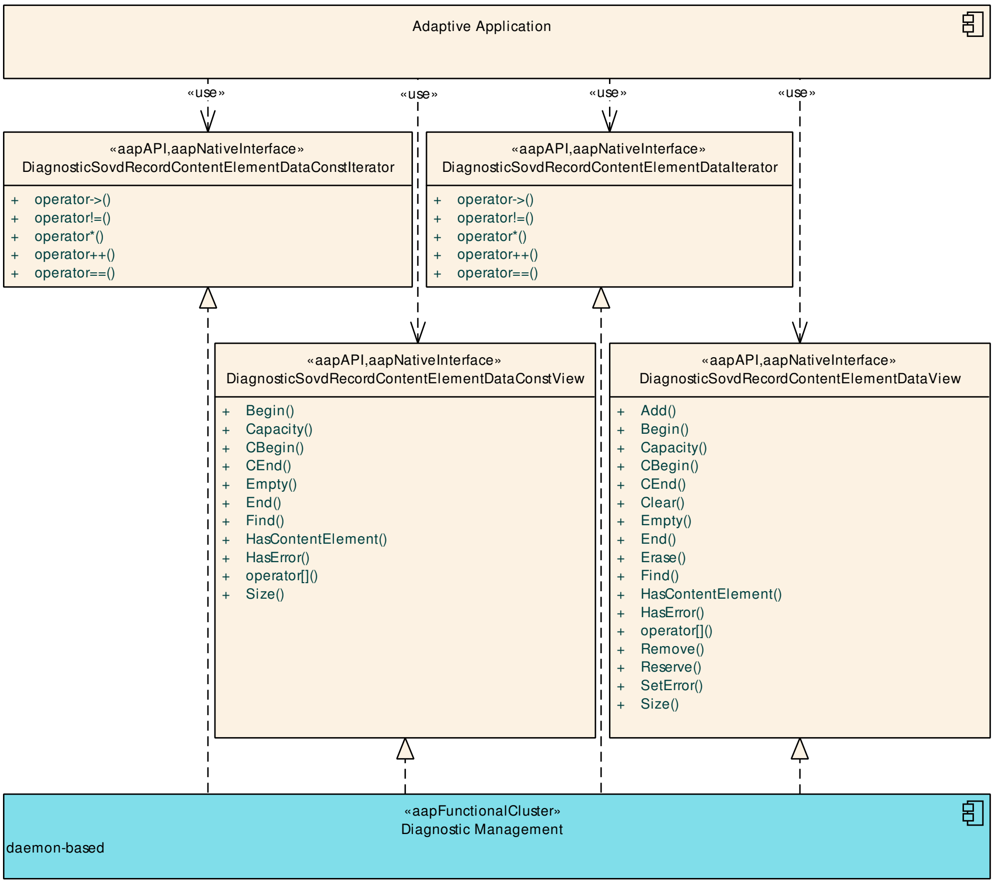

Figure 9.122: Interfaces for SOVD record data

Name:	DiagnosticSovdRecordContentElementDataConstIterator	
Technology:	Native interface	
Usage:	Public API	
Description:	Const iterator for a DiagnosticSovdRecordContentElementData	
Operations:	operator!=	Not equal operator

▽

△

	operator*	Dereference operator
	operator++	Increment operator
	operator->	Dereference operator
	operator==	Equal operator

Name:	DiagnosticSovdRecordContentElementDataConstView	
Technology:	Native interface	
Usage:	Public API	
Description:	Const view on a DiagnosticSovdAbstractContentData with Record type	
Operations:	Begin	Returns the begin operator of a RecordContentElement Data
	CBegin	Returns the begin operator of a RecordContentElement Data
	CEnd	Returns the end operator of a RecordContentElement Data
	Capacity	Returns the capacity of the RecordContentElementData
	Empty	Returns if the RecordContentElementData is empty
	End	Returns the end operator of a RecordContentElement Data
	Find	Find an element with the specified key
	HasContentElement	Returns if the RecordContentElementData contains an element with the specified key
	HasError	Returns if the abstract content contains an error
	Size	Returns the count of elements in the RecordContent ElementData
	operator[]	Access the content element with specified key

Name:	DiagnosticSovdRecordContentElementDataIterator	
Technology:	Native interface	
Usage:	Public API	
Description:	Iterator for a DiagnosticSovdRecordContentElementData	
Operations:	operator!=	Not equal operator
	operator*	Dereference operator
	operator++	Increment operator
	operator->	Dereference operator
	operator==	Equal operator

Name:	DiagnosticSovdRecordContentElementDataView	
Technology:	Native interface	
Usage:	Public API	
Description:	View on a DiagnosticSovdAbstractContentData with Record type	
Operations:	Add	Adds an element with the specified key
	Begin	Returns the begin operator of a RecordContentElement Data

▽

△

	CBegin	Returns the begin operator of a RecordContentElement Data
	CEnd	Returns the end operator of a RecordContentElement Data
	Capacity	Returns the capacity of the RecordContentElementData
	Clear	Removes all the content element in the RecordContent ElementData
	Empty	Returns if the RecordContentElementData is empty
	End	Returns the end operator of a RecordContentElement Data
	Erase	Erases a content element from DiagnosticSovdRecord ContentElementData
	Find	Find an element with the specified key
	HasContentElement	Returns if the RecordContentElementData contains an element with the specified key
	HasError	Returns if the abstract content contains an error
	Remove	Removes an element with the specified key
	Reserve	Reserves the capacity of the RecordContentElement Data
	SetError	Sets the abstract content to be erroneous
	Size	Returns the count of elements in the RecordContent ElementData
	operator[]	Access the content element with specified key

9.8.1.2 Provided interfaces

`Diagnostic Management` does not provide any interfaces to other Functional Clusters.

9.8.1.3 Required interfaces

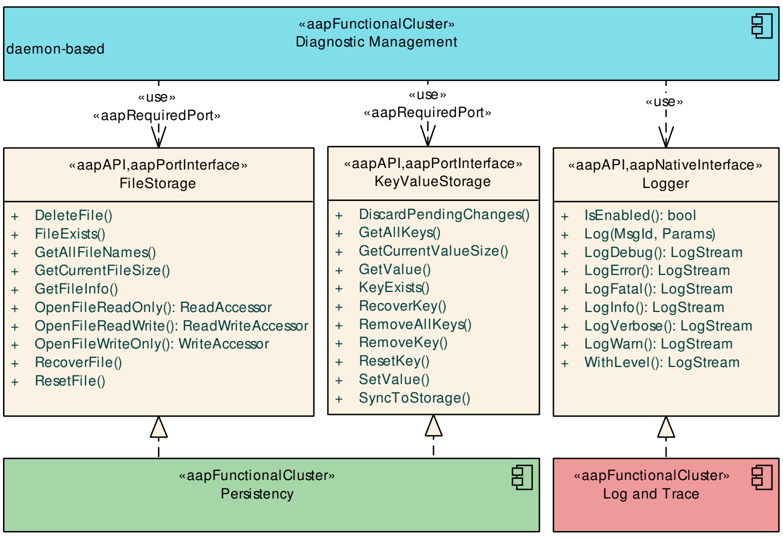

Figure 9.123: Interfaces required by Diagnostic Management

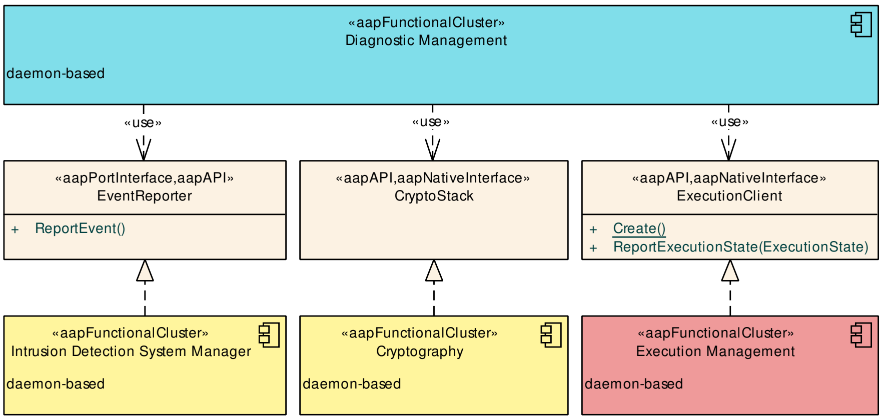

Figure 9.124: Security-related Interfaces required by Diagnostic Management

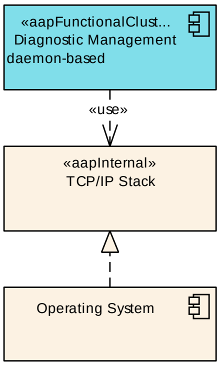

Figure 9.125: Interfaces required by Diagnostic Management from external components

Interface	*Purpose*
ManifestAccessor	Diagnostic Management shall use this interface to read its configuration information from the Manifests.
TCP/IP Stack	This interface is used for DoIP Transport Protocols.
Cryptography::CryptoStack	This interface may be used e.g., to access keys for secure diagnostics.
Diagnostic Management::Communication Control	This interface should be used to realize UDS Service 0x28 - CommunicationControl.
Execution Management::ExecutionClient	This interface is used to report the status of the Diagnostic Management daemon process(es).
Intrusion Detection System Manager::Event Reporter	Diagnostic Management uses this interface to report standardized security events.
Log and Trace::Logger	Diagnostic Management shall use this interface to log standardized messages.
Persistency::FileStorage	Used to store associated data of diagnostic trouble codes (e.g., freeze frames).
Persistency::KeyValueStorage	Used to store properties of diagnostic trouble codes and diagnostic sessions.
Platform Health Management::SupervisedEntity	Diagnostic Management should use this interface to enable supervision of its daemon process(es) by Platform Health Management.

Table 9.34: Interfaces required by Diagnostic Management

10 Use-Case View

This chapter provides an overview of the use cases of the AUTOSAR Adaptive Platform. The use cases are defined on the level of the `Functional Cluster`s in the AUTOSAR Adaptive Platform. The chapter structure corresponds to the Building Block View in Section 9.

The use cases in this chapter are specified in a solution-neutral way. As a consequence, include and extend relationships are used sparingly to avoid "programming" with use cases. Please refer to the individual scenarios in chapter 11 that provide a detailed insight how the use cases could be implemented and how they refer to each other.

The use cases are described using tables that are based on the template shown in Table 10.1. Some rows are optional and will be left out if empty. Each use case has one or more scenarios. The default scenarios show a successful execution of the use case. Other scenarios show error or alternate cases.

Name:	The name of this use case	
Description:	A brief description of this use case.	
Supporting:	If yes, this use case is defined just for factoring out common behavior by means of include or extend relationships only. The use case is not triggered directly by an actor and usually does not have own scenarios. This row is optional.	
Triggering Actors:	The name of an actor that triggers this use case. This row is optional.	A brief description how the actor triggers the use case.
Participating Actors:	The name of an actor that participates in this use case. This row is optional.	A brief description how the actor participates the use case.
Base use case:	The name of the use case that this use cases specializes. This row is optional.	
Included use cases:	The name of an use case that is included by this use case. This row is optional.	A brief description of the reason for including the use case.
Extending use cases:	The name of an use case that extends this use case. This row is optional.	A brief description of the condition for extending this use case.
Preconditions:	A list of preconditions that need to be fulfilled before the use case can be executed.	
Invariants:	A list of invariants that need to be fulfilled before the use case can be executed, while the use case is executed, and after the use case has been executed.	
Postconditions:	A list of postconditions that need to be fulfilled after the use case has been executed.	
Scenarios:	The name of the scenario. This row is optional.	A brief description of the scenario.

Table 10.1: Template for Use Cases

10.1 Runtime

10.1.1 Execution Management

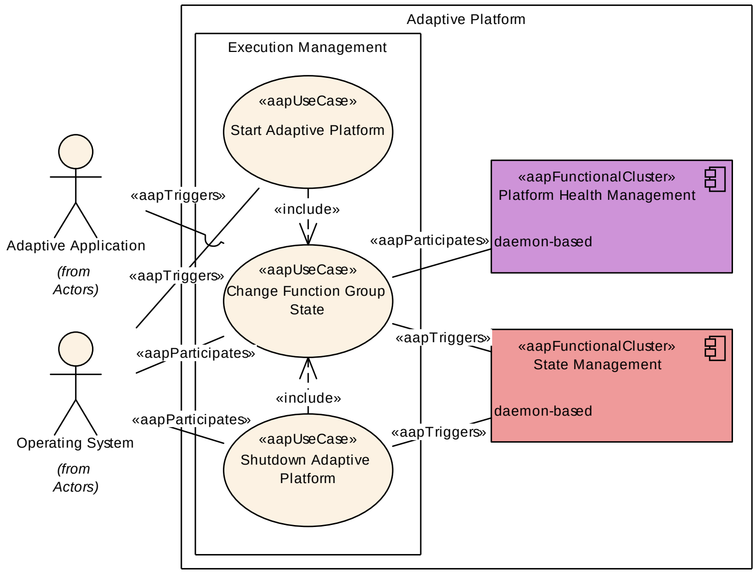

Figure 10.1: Use cases for Execution Management

10.1.1.1 Start Adaptive Platform

Name:	Start Adaptive Platform	
Description:	This use case describes the start of the AUTOSAR Adaptive Platform.	
Triggering Actors:	Operating System	The `Operating System` triggers this use case by starting `Execution Management` as the first process of the AUTOSAR Adaptive Platform. The `Operating System` also participates in this use case by starting processes.
Included use cases:	`Change Function Group State`	This use case is included to perform the transition to the `Machine Function Group State` named `Startup`.
Preconditions:	• The base software, for example a operating system or a hypervisor, is initialized and running.	
Invariants:	• The configuration of the base software remains unchanged, in particular the processes of base software.	
Postconditions:	• The standardized `Machine Function Group State` named `Startup` has been entered.	

▽

△

Scenarios:	Start Adaptive Platform with Supervision of Adaptive Applications	This is the default scenario showing the startup of the Adaptive Platform with an supervision of Adaptive Applications. It therefore includes the startup of Platform Health Management that performs supervision and the startup of an Adaptive Application that is supervised as part of the Startup Machine Function Group State. Additional Functional Clusters and Application processes may need to be started in addition depending on the Adaptive Platform implementation and project-specific needs.

Table 10.2: Use-Case Start Adaptive Platform

10.1.1.2 Shutdown Adaptive Platform

Name:	Shutdown Adaptive Platform	
Description:	This use case describes the shutdown of the AUTOSAR Adaptive Platform.	
Triggering Actors:	State Management	State Management triggers this use case by requesting the standardized Machine Function Group State named Shutdown.
Participating Actors:	Operating System	The Operating System participates in this use case by starting and stopping processes.
Included use cases:	Change Function Group State	This use case is included to perform the transition to the Machine Function Group State named Shutdown.
Preconditions:	None	
Invariants:	None	
Postconditions:	• The AUTOSAR Adaptive Platform has been shutdown.	
Scenarios:	Shutdown Adaptive Platform with Supervision of Adaptive Applications	This is the default scenario showing the regular shutdown of the Adaptive Platform. It includes the shutdown of Platform Health Management that performs supervision of applications.

Table 10.3: Use-Case Shutdown Adaptive Platform

10.1.1.3 Change Function Group State

Name:	Change Function Group State	
Description:	This use case describes the change of a `Function Group State`. The use case is applicable to both, `Function Group State`s defined by an application as well as predefined `Function Group State`s of the AUTOSAR Adaptive Platform (i.e., `Machine Function Group State`). See Sections 13.2 and 13.3 for details.	
Triggering Actors:	State Management	`State Management` triggers this use case by requesting a `Function Group State` from `Execution Management`.
Participating Actors:	Operating System	The `Operating System` participates in this use case by starting / stopping processes as requested by `Execution Management`.
	Adaptive Application	The `Adaptive Application` participates in this use case by reporting its execution state to `Execution Management`.
	Platform Health Management	`Platform Health Management` participates in this use case by using information from `Execution Management` about processes to be newly started or terminated for coordination with supervision checkpoints.
Preconditions:	None	
Invariants:	• The configuration of the base software remains unchanged, in particular the processes of base software.	
Postconditions:	• The requested `Function Group State` has been entered successfully.	
Scenarios:	`Change Function Group State`	This is the default scenario showing the regular change of a `Function Group State`.

Table 10.4: Use-Case Change Function Group State

10.1.2 State Management

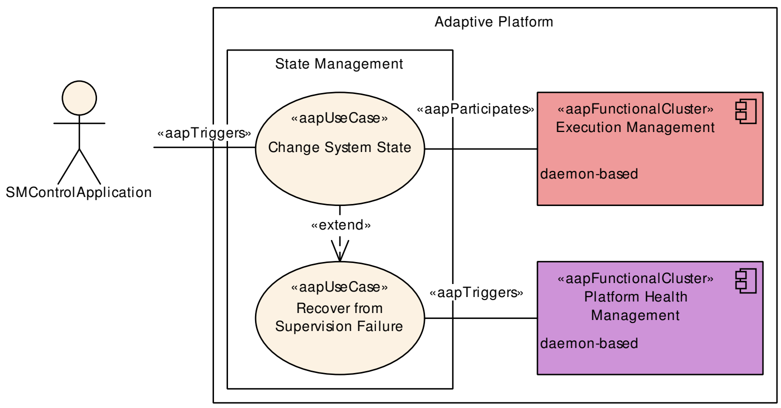

Figure 10.2: Use cases for State Management

10.1.2.1 Change System State

Name:	Change System State	
Description:	This use case describes the change the state of the system (i.e., the configuration and extent of the running Processes).	
Triggering Actors:	SMControlApplication	The SMControlApplication triggers this use case by calling RequestTransition.
Participating Actors:	Execution Management	Execution Management participates in this use case by requesting a start or termination of processes from the Operating System.
Preconditions:	• The system is not in Machine State Shutdown.	
Invariants:	None	
Postconditions:	• The system state (i.e. the configuration and extent of running Processes) has changed.	
Scenarios:	Change System State	This is the default scenario showing a state change reaction by State Management when calling RequestTransition.

Table 10.5: Use-Case Change System State

10.1.2.2 Recover from Supervision Failure

Name:	Recover from Supervision Failure	
Description:	This use case describes the recovery from a supervision failure reported by Platform Health Management.	
Triggering Actors:	Platform Health Management	Platform Health Management triggers this use case.
Extending use cases:	Change System State	Change System State extends this use case if State Management decides to switch the system state to recover from a supervision failure.
Preconditions:	• State Management needs to be registered with the RecoveryAction interface of Platform Health Management.	
Invariants:	None	
Postconditions:	No postconditions defined at the moment.	
Scenarios:	Recover by System State Change	This is the default scenario showing a recovery from a supervision failure by switching the system state.

▽

△

	Recover by Watchdog Reset	This alternate scenario shows a recovery from a supervision failure by triggering a Watchdog reset.

Table 10.6: Use-Case Recover from Supervision Failure

10.2 Communication

10.2.1 Raw Data Stream

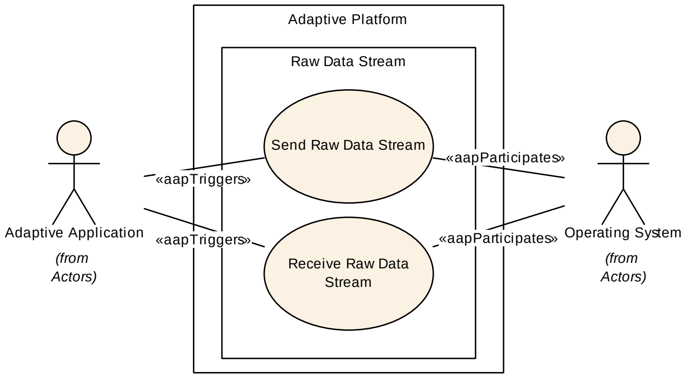

Figure 10.3: Use cases for Raw Data Stream

Raw Data Stream supports IP based protocols (network layer) and the IEEE1722 protocol (data link layer) to send and receive data. However, from a use case perspective, the protocols are not differentiated.

10.2.1.1 Send Raw Data Stream

Name:	Send Raw Data Stream	
Description:	This use case demonstrates sending of raw data.	
Triggering Actors:	Adaptive Application	The Adaptive Application triggers this use case by requesting a the transmission of data from Raw Data Stream.
Participating Actors:	Operating System	The Operating System participates in this use case by handling the lower level protocol(s) used to transmit data packets on the network.

▽

△

Preconditions:	• Internal connection to a network layer or data link layer socket is established • In case the TCP protocol is used for data transmission, a connection between client and server is established	
Invariants:	None	
Postconditions:	• The requested provided data has been transmitted.	
Scenarios:	Send IEEE1722 Stream	This scenario shows sending data via the IEEE1722 protocol (data link layer).
	Send IP Stream	This scenario shows sending data via an IP-based protocol (network layer).

Table 10.7: Use-Case Send Raw Data Stream

10.2.1.2 Receive Raw Data Stream

Name:	Receive Raw Data Stream	
Description:	This use case demonstrates receiving of raw data.	
Triggering Actors:	Adaptive Application	The Adaptive Application triggers this use case by requesting to read data from Raw Data Stream.
Participating Actors:	Operating System	The Operating System participates in this use case by handling the lower level protocol(s) used to receive data packets on the network.
Preconditions:	• Internal connection to a network layer or data link layer socket is established • In case the TCP protocol is used for data transmission, a connection between client and server is established	
Invariants:	None	
Postconditions:	None	
Scenarios:	Receive IEEE1722 Stream	This scenario shows receiving data via the IEEE1722 protocol (data link layer).
	Receive IP Stream	This scenario shows receiving data via an IP-based protocol (network layer).

Table 10.8: Use-Case Receive Raw Data Stream

10.2.2 Communication Management

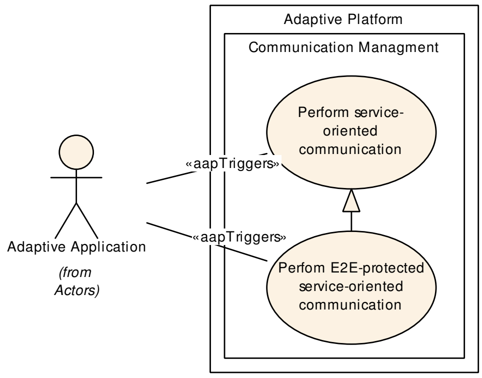

Figure 10.4: Use cases for Communication Management

10.2.2.1 Perform service-oriented communication

Name:	Perform Service-Oriented Communication	
Description:	An adaptive application attempts to send data over a communication channel. That can be data of a remote method invocation, an event, or the value of a field.	
Triggering Actors:	Adaptive Application	The `Adaptive Application` triggers this use case by sending data.
Base use case:	None	
Extending use cases:	None	
Preconditions:	None	
Invariants:	None	
Postconditions:	The received data is available.	
Scenarios:	None	

Table 10.9: Use-Case Perform Service-Oriented Communication

10.2.2.2 Perform E2E-protected service-oriented communication

Name:	Perform E2E-protected Service-Oriented Communication	
Description:	An adaptive application attempts to send data over a (unsafe) communication channel. That can be data of a remote method invocation, an event, or the value of a field. Communication management serializes the data and adds E2E protection. On the receiving side Communication management performs E2E checks on received data. The result of the E2E check is provided together with the received data to the application that is then responsible to handle it.	
Triggering Actors:	Adaptive Application	The `Adaptive Application` triggers this use case by sending data.
Base use case:	`Perform service-oriented communication`	
Extending use cases:	None	
Preconditions:	None	
Invariants:	None	
Postconditions:	E2E evaluation result is available together with the received data.	
Scenarios:	End-to-end communication protection for Events	See [8] Chapter 7.4.1.1.
	End-to-end communication protection for Methods	See [8] Chapter 7.4.1.2.
	End-to-end communication protection for Fields	See [8] Chapter 7.4.1.3.

Table 10.10: Use-Case Perform E2E-protected Service-Oriented Communication

10.2.3 Time Synchronization

The use cases for `Time Synchronization` are separated into use cases for Time Synchronization Master (see Section 10.2.3.1) and Time Synchronization Slave (see Section 10.2.3.2) are described below.

10.2.3.1 Time Synchronization Master

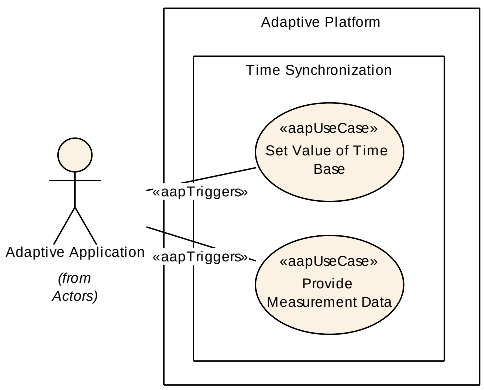

Figure 10.5: Use cases for Time Synchronization Master

10.2.3.1.1 Set Value Of Time Base

Name:	Set Value Of Time Base	
Description:	This use case describes setting the value of a time base.	
Triggering Actors:	Adaptive Application	The Adaptive Application triggers this use case by providing a value for the time base.
Preconditions:	None	
Invariants:	None	
Postconditions:	None	
Scenarios:	Set Value Of Time Base	This is a default scenario showing the update of the time base value.

Table 10.11: Use-Case Set Value Of Time Base

10.2.3.1.2 Provide Measurement Data

Name:	Provide Measurement Data	
Description:	This use case describes providing measurement data such as timing details.	
Triggering Actors:	Adaptive Application	The Adaptive Application triggers this use case by providing detailed timing data.
Preconditions:	None	
Invariants:	None	
Postconditions:	None	
Scenarios:	Provide Measurement Data	This scenario is not detailed in this document due to its simplicity.

Table 10.12: Use-Case Provide Measurement Data

10.2.3.2 Time Synchronization Slave

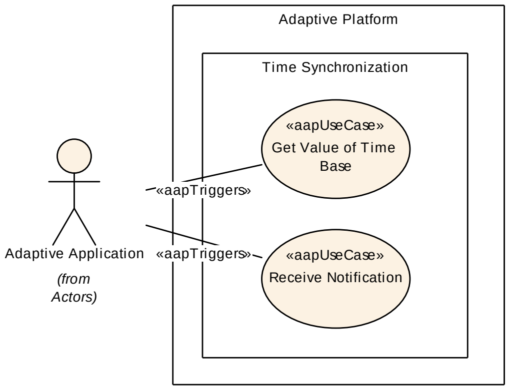

Figure 10.6: Use cases for Time Synchronization Slave

10.2.3.2.1 Get Value Of Time Base

Name:	Get Value Of Time Base	
Description:	This use case describes getting the value of a time base.	
Triggering Actors:	Adaptive Application	The `Adaptive Application` triggers this use case by asking for a time value.
Preconditions:	None	
Invariants:	None	
Postconditions:	None	
Scenarios:	`Get Value Of Time Base`	This is a default scenario showing the retrieval of the time base value by an `Adaptive Application`.

Table 10.13: Use-Case Get Value Of Time Base

10.2.3.2.2 Receive Notification

Name:	Receive Notification	
Description:	This use case describes receiving of a notification.	
Triggering Actors:	Adaptive Application	The `Adaptive Application` triggers this use case by registering status change notifier.
Preconditions:	The `Adaptive Application` needs to be registered with the Time Synchronization Slave.	
Invariants:	None	

▽

△

Postconditions:	None	
Scenarios:	Receive Notification	This is a default scenario showing the reception of a notification by the Adaptive Application in case of either a change of the synchronization status or user data, or a leap jump.

Table 10.14: Use-Case Receive Notification

10.2.4 Automotive API Gateway

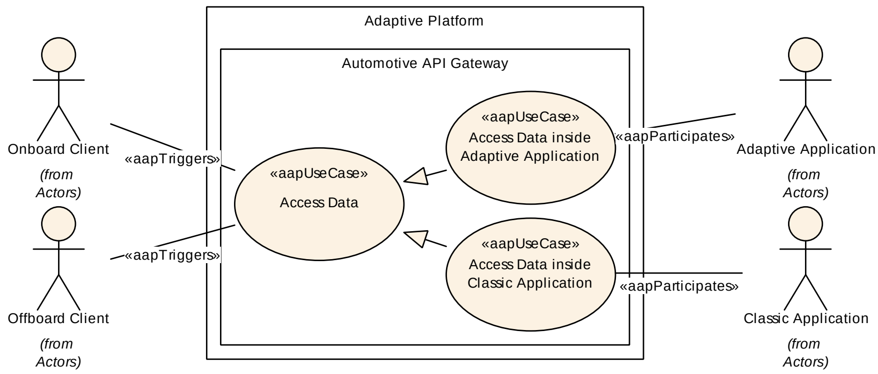

Figure 10.7: Use cases for the Automotive API Gateway

10.2.4.1 Access Data

Name:	Access Data	
Description:	This use case describes accessing data inside the vehicle in a uniform way. This means being able to read, write, and subscribe to data.	
Triggering Actors:	Offboard Client	An Offboard Client triggers this use case by using one of the access functions that the Automotive API Gateway provides.
	Onboard Client	An Offboard Client triggers this use case by using one of the access functions that the Automotive API Gateway provides.
Participating Actors:	Adaptive Application	An Adaptive Application provides access to certain data to make it accessible via the Automotive API Gateway.

▽

△

	Classic Application	A Classic Application provides access to certain data to make it accessible via the Automotive API Gateway.
Preconditions:	None	
Invariants:	None	
Postconditions:	None	
Scenarios:	Access Data inside Adaptive Application	This is a specialization where the data that is accessed is managed by an Adaptive Application.
	Access Data inside Classic Application	This is a specialization where the data that is accessed is managed by a Classic Application.

Table 10.15: Use-Case Access Data

10.3 Storage

10.3.1 Persistency

The use cases for Persistency are organized into three categories. Section 10.3.1.1 lists the use cases for reading persistent data. Section 10.3.1.2 lists the use cases for storing persistent data. Section 10.3.1.3 lists the use cases related to monitoring of persistent storage.

10.3.1.1 Read Persistent Data

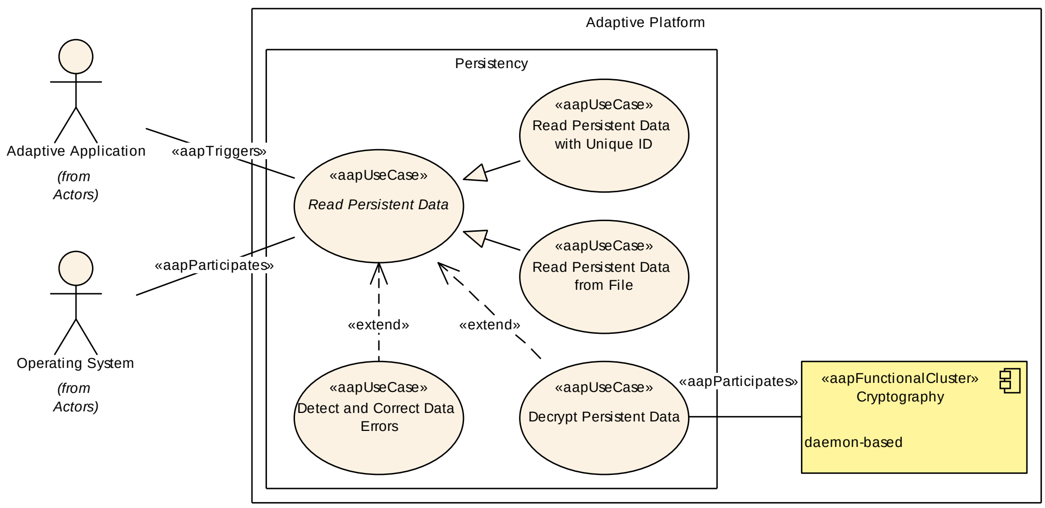

Figure 10.8: Use Cases for Reading Persistent Data

10.3.1.1.1 Read Persistent Data from File

Name:	Read Persistent Data from File	
Description:	This use case describes reading persistent data from a file.	
Triggering Actors:	Adaptive Application	The Adaptive Application triggers this use case.
Participating Actors:	Operating System	The Operating System participates in this use case by providing access to the underlying storage.
Base use case:	Read Persistent Data	
Extending use cases:	Detect and Correct Data Errors	Detect and Correct Data Errors extends this use case if redundancy is configured in the Manifest (see PersistencyRedundancyHandling).
	Decrypt Persistent Data	Decrypt Persistent Data extends this use case if encryption is configured in the Manifest (see PersistencyDeploymentToCryptoKeySlotMapping or PersistencyDeploymentElementToCryptoKeySlotMapping).
Preconditions:	• The requested file exists.	
Invariants:	None	
Postconditions:	None	
Scenarios:	Read Persistent Data from File	This is the default scenario showing how persistent data is read from a file.

Table 10.16: Use-Case Read Persistent Data from File

10.3.1.1.2 Read Persistent Data with Unique ID

Name:	Read Persistent Data With Unique ID	
Description:	This use case describes reading persistent data associated with a unique ID (key).	
Triggering Actors:	Adaptive Application	The Adaptive Application triggers this use case.
Participating Actors:	Operating System	The Operating System participates in this use case by providing access to the underlying storage.
Base use case:	Read Persistent Data	
Extending use cases:	Detect and Correct Data Errors	Detect and Correct Data Errors extends this use case if redundancy is configured in the Manifest (see PersistencyRedundancyHandling).

▽

△

	Decrypt Persistent Data	Decrypt Persistent Data extends this use case if encryption is configured in the Manifest (see PersistencyDeploymentToCryptoKeySlotMapping or PersistencyDeploymentElementToCryptoKeySlotMapping).
Preconditions:	• The requested unique ID (key) exists.	
Invariants:	None	
Postconditions:	None	
Scenarios:	Read Persistent Data with Unique ID	This is the default scenario showing how data is read with a unique ID (key).

Table 10.17: Use-Case Read Persistent Data With Unique ID

10.3.1.1.3 Decrypt Persistent Data

Name:	Decrypt Persistent Data	
Description:	This use case describes the decryption of persistent data. It is part of the scenarios of the concrete use cases Read Persistent Data with Unique ID and Read Persistent Data from File	
Participating Actors:	Cryptography	Cryptography participates in this use case by decrypting data.
Preconditions:	• Encryption/decryption is configured in the Manifest (see PersistencyDeploymentToCryptoKeySlotMapping or PersistencyDeploymentElementToCryptoKeySlotMapping).	
Invariants:	None	
Postconditions:	None	
Scenarios:	None, see use cases Read Persistent Data with Unique ID and Read Persistent Data from File.	

Table 10.18: Use-Case Decrypt Persistent Data

10.3.1.1.4 Detect and Correct Data Errors

Name:	Detect and Correct Data Errors
Description:	An Adaptive Application attempts to read data that is stored safely, using CRCs/hash values and redundant copies. The read access performs a check of data and redundancy information. In case the data is valid, it is returned. Otherwise, if it is corrupted and cannot be restored from an undamaged redundant copy, an error is returned.

▽

△

Preconditions:	• Redundancy is configured in the Manifest (see PersistencyRedundancyHandling).
Invariants:	None
Postconditions:	• Valid data is returned or an error is returned.
Scenarios:	None, see use cases Read Persistent Data with Unique ID and Read Persistent Data from File.

Table 10.19: Use-Case Detect and Correct Data Errors

10.3.1.2 Store Persistent Data

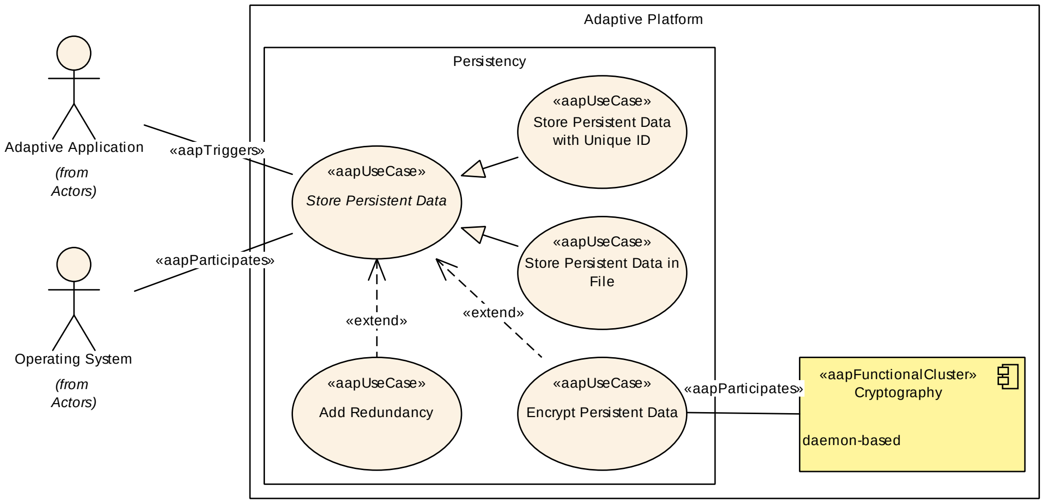

Figure 10.9: Use Cases for Storing Persistent Data

10.3.1.2.1 Store Persistent Data in File

Name:	Store Persistent Data in File	
Description:	This use case describes storing persistent data in a file.	
Triggering Actors:	Adaptive Application	The Adaptive Application triggers this use case.
Participating Actors:	Operating System	The Operating System participates in this use case by providing access to the underlying storage.
Base use case:	Store Persistent Data	

▽

△

Extending use cases:	Encrypt Persistent Data	Encrypt Persistent Data extends this use case if encryption is configured in the Manifest (see PersistencyDeploymentToCryptoKeySlotMapping or PersistencyDeploymentElementToCryptoKeySlotMapping).
	Add Redundancy	Add Redundancy extends this use case if encryption is configured in the Manifest (see PersistencyRedundancyHandling).
Preconditions:	• The requested file exists.	
Invariants:	None	
Postconditions:	None	
Scenarios:	Store Persistent Data in File	This is the default scenario showing how persistent data is stored in a file.

Table 10.20: Use-Case Store Persistent Data in File

10.3.1.2.2 Store Persistent Data with Unique ID

Name:	Store Persistent Data with Unique ID	
Description:	This use case describes storing persistent data associated with an unique ID (key).	
Triggering Actors:	Adaptive Application	The Adaptive Application triggers this use case.
Participating Actors:	Operating System	The Operating System participates in this use case by providing access to the underlying storage.
Base use case:	Store Persistent Data	
Extending use cases:	Encrypt Persistent Data	Encrypt Persistent Data extends this use case if encryption is configured in the Manifest (see PersistencyDeploymentToCryptoKeySlotMapping or PersistencyDeploymentElementToCryptoKeySlotMapping).
	Add Redundancy	Add Redundancy extends this use case if encryption is configured in the Manifest (see PersistencyRedundancyHandling).
Preconditions:	• The requested unique ID (key) exists.	
Invariants:	None	
Postconditions:	None	

▽

△

Scenarios:	Store Persistent Data with Unique ID	This is the default scenario showing how persistent data is stored with a unique ID (key).

Table 10.21: Use-Case Store Persistent Data with Unique ID

10.3.1.2.3 Encrypt Persistent Data

Name:	Encrypt Persistent Data	
Description:	This use case describes the encryption of persistent data. It is part of the scenarios of the concrete use cases Store Persistent Data with Unique ID and Store Persistent Data in File	
Participating Actors:	Cryptography	The Cryptography participates in this scenario by encrypting persistent data.
Preconditions:	• Encryption/decryption is configured in the Manifest (see PersistencyDeploymentToCryptoKeySlotMapping or PersistencyDeploymentElementToCryptoKeySlotMapping).	
Invariants:	None	
Postconditions:	None	
Scenarios:	None, see use cases Store Persistent Data with Unique ID and Store Persistent Data in File.	

Table 10.22: Use-Case Encrypt Persistent Data

10.3.1.2.4 Add Redundancy

Name:	Add Redundancy
Description:	An adaptive application attempts to store data safely by storing it together with CRCs/hash values and/or additional copies of that data.
Preconditions:	• Redundancy is configured in the Manifest (see PersistencyRedundancyHandling).
Invariants:	None
Postconditions:	• Data is stored together with redundant data.
Scenarios:	None, see use cases Store Persistent Data with Unique ID and Store Persistent Data in File.

Table 10.23: Use-Case Add Redundancy

10.3.1.3 Monitoring

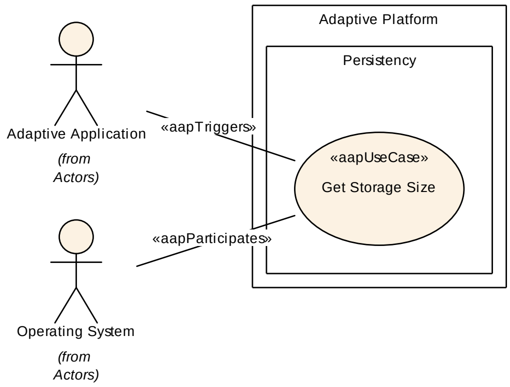

Figure 10.10: Use Cases for Monitoring

10.3.1.3.1 Get Storage Size

Name:	Get Storage Size	
Description:	This use case describes determining the size of the storage space occupied by persistent data.	
Triggering Actors:	Adaptive Application	The `Adaptive Application` triggers this use case.
Participating Actors:	Operating System	The `Operating System` participates in this use case by providing information from underlying storage.
Preconditions:	None	
Invariants:	None	
Postconditions:	None	
Scenarios:	`Get FileStorage Size`	This scenario shows how to obtain information about occupied storage space of a `FileStorage`.
	`Get KeyValueStorage Size`	This scenario shows how to obtain information about occupied storage space of a `KeyValueStorage`.

Table 10.24: Use-Case Get Storage Size

10.4 Security

10.4.1 Firewall

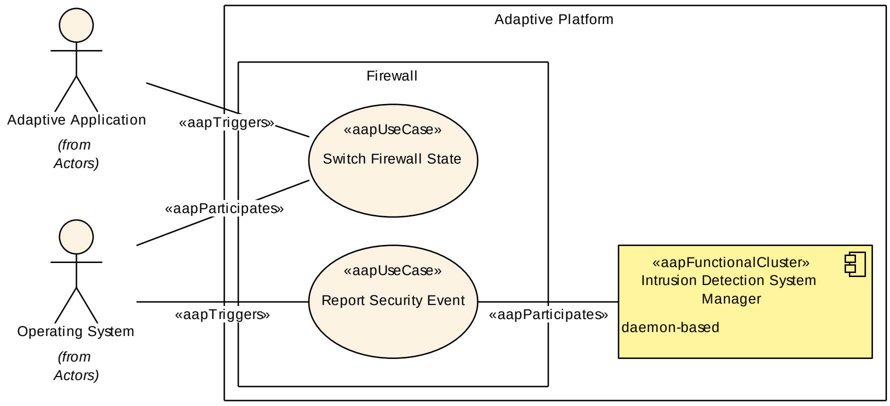

Figure 10.11: Use cases for Firewall

10.4.1.1 Switch Firewall State

Name:	Switch Firewall State	
Description:	This use case describes switching the firewall state.	
Triggering Actors:	Adaptive Application	The `Adaptive Application` triggers this use case.
Participating Actors:	Operating System	The `Operating System` participates in this use case by applying the firewall rules associated with the target firewall state to its TCP/IP stack or, for example, a hardware firewall.
Preconditions:	None	
Invariants:	None	
Postconditions:	• The active firewall state has changed to the requested target state. • The configured firewall rules have been applied to the underlying firewall implementation.	
Scenarios:	`Switch Firewall State`	This is the default scenario showing the switch of the firewall state.

Table 10.25: Use-Case Switch Firewall State

10.4.1.2 Report Security Event

Name:	Report Security Event	
Description:	This use case describes reporting a Security Event to Intrusion Detection System Manager if a message is received that matches a configured rule for blocked messages.	
Triggering Actors:	Operating System	The Operating System triggers this use case by informing the Firewall that a message was received that matches a configured rule for blocked messages.
Participating Actors:	Intrusion Detection System Manager	Intrusion Detection System Manager participates in this use case by handling the generated Security Event.
Preconditions:	• A rule for blocked messages and a corresponding Security Event have been configured in the Manifest.	
Invariants:	None	
Postconditions:	• A Security Event has been generated and forwarded to the Intrusion Detection System Manager.	
Scenarios:	Report Security Event	This is the default scenario showing the reporting of a security event.

Table 10.26: Use-Case Report Security Event

10.5 Safety

10.5.1 Platform Health Management

10.5.1.1 Manage Watchdog

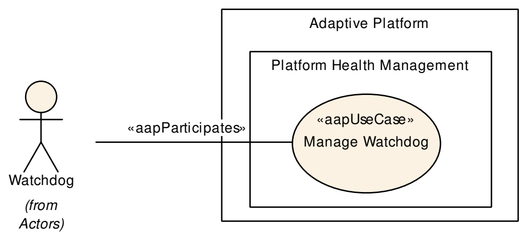

Figure 10.12: Use case Manage Watchdog

Name:	Manage Watchdog
Description:	This use case describes managing the watchdog.
Triggering Event:	The use case is triggered by an internal event (successful supervision cycle).

▽

△

Participating Actors:	Watchdog	The Watchdog participates in this use case by receiving "alive" notification from the Platform Health Management periodically.
Preconditions:	None	
Invariants:	Maximum time to reset the watchdog has not expired.	
Postconditions:	None	
Scenarios:	Manage Watchdog	This scenario shows managing the watchdog.

Table 10.27: Use-Case Manage Watchdog

10.5.1.2 Fire Watchdog Reaction

Name:	Fire Watchdog Reaction	
Description:	This use case describes the firing of the watchdog reaction, which results in a reset of the Machine.	
Triggering Event:	The use case is triggered by an internal event (e.g., a critical supervision failure after which State Management requests a Watchdog reset from Platform Health Management, or State Management does not respond within the configured time to handle a supervision failure).	
Participating Actors:	Watchdog	The Watchdog participates in this use case by executing the watchdog reaction.
Preconditions:	None	
Invariants:	None	
Postconditions:	Watchdog reaction has been initiated.	
Scenarios:	This use case is part of Recover from Critical Supervision Failure in State Management and Recover from Critical Supervision Failure in Execution Management.	

Table 10.28: Use-Case Fire Watchdog Reaction

10.5.1.3 Report Supervision Failure

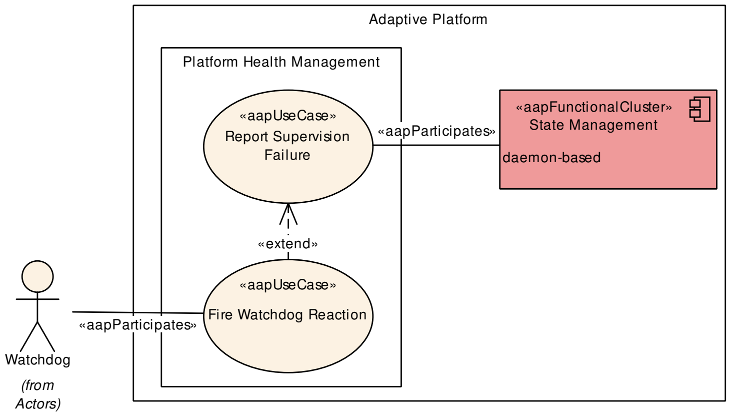

Figure 10.13: Use case Report Supervision Failure

Name:	Report Supervision Failure	
Description:	This use case describes reporting a supervision failure.	
Triggering Event:	The use case is triggered by a supervision failure detected by `Platform Health Management`.	
Participating Actors:	`State Management`	`State Management` participates in this use case by handling recovery.
Extending use cases:	`Fire Watchdog Reaction`	`Fire Watchdog Reaction` extends this use case if the supervision failure cannot be handled by `State Management`.
Preconditions:	• `State Management` has enabled potential invocation of `RecoveryHandler()` by calling the method `Offer()`.	
Invariants:	None	
Postconditions:	None	
Scenarios:	`Report Supervision Failure`	This default scenario shows the reporting of a supervision failure.

Table 10.29: Use-Case Report Supervision Failure

10.5.1.4 Recover from Critical Supervision Failure

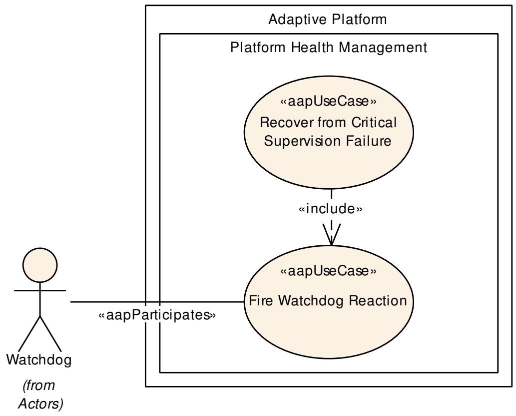

Figure 10.14: Use case Recover from Critical Supervision Failure

Name:	Recover from Critical Supervision Failure	
Description:	This use case describes the recovery from a critical supervision failure, i.e., `Execution Management` or `State Management` processes failed.	
Triggering Event:	The use case is triggered by a critical supervision failure detected by `Platform Health Management`.	
Included use cases:	`Fire Watchdog Reaction`	This use case is included to perform recovery by firing a watchdog reaction upon failure of `Execution Management` or `State Management`.
Preconditions:	• `Execution Management` and `State Management` processes are supervised by `Platform Health Management`.	
Invariants:	None	
Postconditions:	• `Platform Health Management` establishes a supervision on `Execution Management` and `State Management` processes after the Adaptive Platform has restarted.	
Scenarios:	`Recover from Critical Supervision Failure in Execution Management`	This scenario shows a recovery from a supervision failure in `Execution Management`.
	`Recover from Critical Supervision Failure in State Management`	This scenario shows a recovery from a supervision failure in `State Management`.

Table 10.30: Use-Case Recover from Critical Supervision Failure

10.5.1.5 Report Checkpoint

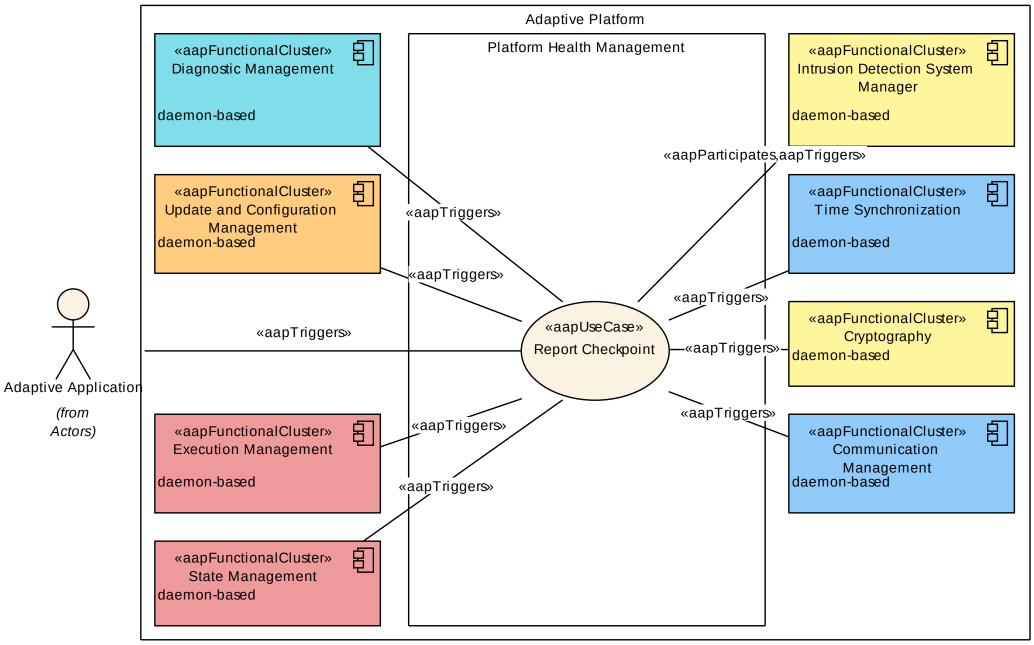

Figure 10.15: Use case Report Checkpoint

Name:	Report Checkpoint	
Description:	This use case describes reporting a checkpoint.	
Triggering Actors:	Adaptive Application	The `Adaptive Application` triggers this use case by reporting a checkpoint.
	Daemon-based Functional Cluster	A daemon-based functional cluster triggers this use case by reporting a checkpoint.
Participating Actors:	Intrusion Detection System Manager	`Intrusion Detection System Manager` participates in this use case if access is denied by generating a security event.
Preconditions:	None	
Invariants:	None	
Postconditions:	None	
Scenarios:	`Report Checkpoint`	This scenario shows how to report a checkpoint.

Table 10.31: Use-Case Report Checkpoint

10.5.2 Safe Hardware Acceleration

Safe Hardware Acceleration assumes two main use-cases for processing data: MIMD - Multiple Instruction, Multiple Data (Chapter 10.5.2.1) and SIMD - Single Instruction, Multiple Data (Chapter 10.5.2.2).

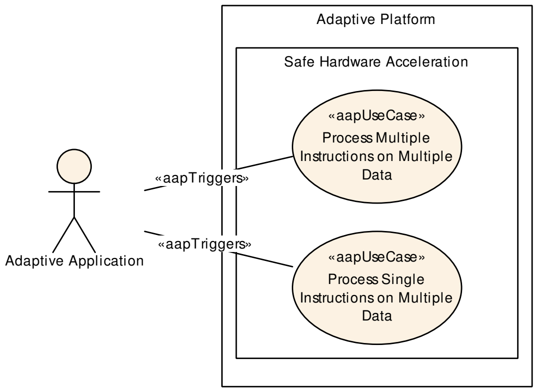

Figure 10.16: Use cases for Safe Hardware Acceleration

10.5.2.1 Process Multiple Instructions on Multiple Data

Name:	Process Multiple Instructions on Multiple Data	
Description:	This use case describes executing different algorithms on multiple datasets simultaneously, for example, in sensor fusion systems that have multiple inputs from different sensors and apply different algorithms over this data in order to fuse it into a single object model.	
Triggering Actors:	Adaptive Application	The `Adaptive Application` triggers this use case.
Preconditions:	None	
Invariants:	None	
Postconditions:	None	
Scenarios:	None	

Table 10.32: Use-Case Process Multiple Instructions on Multiple Data

10.5.2.2 Process Single Instruction on Multiple Data

Name:	Process Single Instruction on Multiple Data	
Description:	This use case describes how to execute the same algorithm over multiple data simultaneously, for example, outlier filtering of a LiDAR point cloud. In this case, the point cloud is split into blocks and outlier filtering is executed in parallel on blocks which will give the same result much faster compared to sequential filtering on a CPU.	
Triggering Actors:	Adaptive Application	The `Adaptive Application` triggers this use case.
Preconditions:	None	

▽

△

Invariants:	None
Postconditions:	None
Scenarios:	None

Table 10.33: Use-Case Process Single Instruction on Multiple Data

10.6 Configuration

10.6.1 Update and Configuration Management

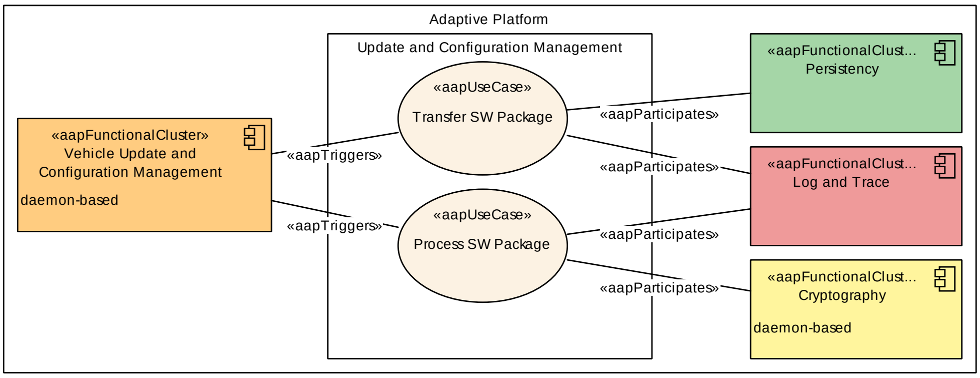

Figure 10.17: Use cases for transferring and processing SW packages

Name:	Transfer SW Package	
Description:	Transfer package data and meta data needed for processing and activation in block wise or in streaming fashion.	
Triggering Actors:	Vehicle Update and Configuration Management	Starting from a vehicle package transfer request, transfer to an individual `Update and Configuration Management` instance is triggered.
Participating Actors:	Persistency	Enables persistent storage for data that is in transfer.
	Log and Trace	Logs of status information related to the Software Package transfer.
Preconditions:	• Sufficient memory is available to transfer the Software Package. • Service API has been successfully subscribed to by triggering applications for initiating the data transfer.	
Invariants:	• Transfer of data has no impact on running processes and existing installations associated with the software modifications.	
Postconditions:	• Transfer of data has been completed or state has been stored for a suspended transfer to enable continuation.	

Table 10.34: Use-Case Transfer SW Package

Name:	Process SW Package	
Description:	Modifies individual Software Package. Use case name is a synonym for any kinds of modification of the software installation, i.e., updating a Software Package, adding a new Software Package, and removing a Software Package.	
Triggering Actors:	Vehicle Update and Configuration Management	Starting from a vehicle package activation request, processing of an individual Software Package is triggered.
Participating Actors:	Cryptography	May provide specific cryptographic services like verification of Software Package authenticity, integrity, and decryption of confidential data.
	Log and Trace	Logs processing related status information.
Preconditions:	• Software Package data is stored completely and consistently in memory or persistent storage. • Triggering actors have subscribed successfully to call service method for processing Software Packages. • `Update and Configuration Management` is currently not involved in execution of any task with an ongoing processing, activation, or rollback.	
Invariants:	• Running processes associated with the Software Packages to be processed are not impacted.	
Postconditions:	• Package processing of all Software Packages in consideration has been completed.	

Table 10.35: Use-Case Process SW Package

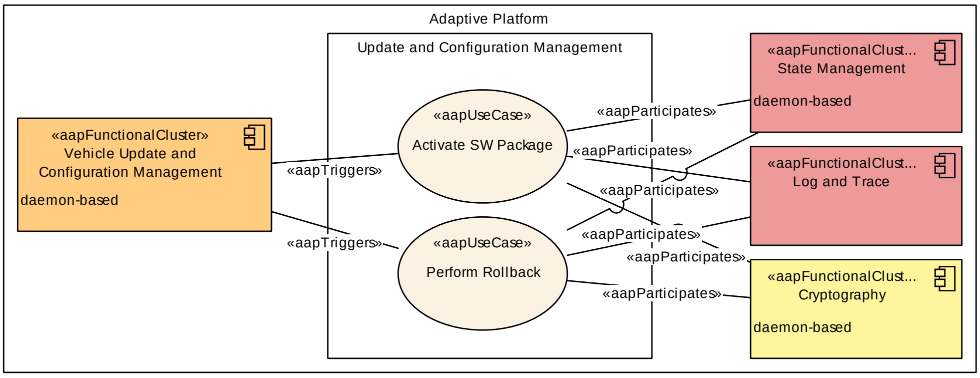

Figure 10.18: Use cases for activating and rolling-back SW packages

Name:	Activate SW Package
Description:	Software Package processed for the local `Machine` shall become active in a next step. This includes shutting down running processes associated with the previous version of the Software Package and potentially removing installed previous version(s) of the Software Package.

▽

△

Triggering Actors:	Vehicle Update and Configuration Management	Starting from a vehicle package activation request, update to individual `Update and Configuration Management` instances is triggered.
Participating Actors:	State Management	For the activation of software, setting of a specific state for the Adaptive Platform instance is required.
	Cryptography	May provide specific cryptographic services like verification of Software Package authenticity, and integrity.
	Log and Trace	Logs activation related status information.
Preconditions:	• Triggering actors have subscribed successfully to call service method for activation of Software Packages. • All Software Packages for activation have been successfully processed. • `Update and Configuration Management` is currently not involved in execution of any task with an ongoing processing, activation, or rollback.	
Invariants:	None	
Postconditions:	• Activation of all Software Packages in consideration has been completed.	

Table 10.36: Use-Case Activate SW Package

Name:	Perform Rollback	
Description:	Request to revert the Software Package processing task and clean up all related data.	
Triggering Actors:	Vehicle Update and Configuration Management	Receiving one of various triggers to initiate Software Package rollback request for an `Update and Configuration Management` instance.
Participating Actors:	State Management	For the rollback of software, setting of a specific state for the Adaptive Platform instance is required.
	Cryptography	May provide specific cryptographic services like verification of Software Package authenticity, and integrity.
	Log and Trace	Logs rollback related status information.
Preconditions:	• Triggering actors have subscribed successfully to call service method for activation of Software Packages. • `Update and Configuration Management` is currently not involved in execution of any task with an ongoing processing, activation, or rollback.	
Invariants:	None	
Postconditions:	• The system has reached the state as before of the processing tasks for the Software Packages in consideration (all relevant processes have been terminated and all data required for operation have been removed).	

Table 10.37: Use-Case Perform Rollback

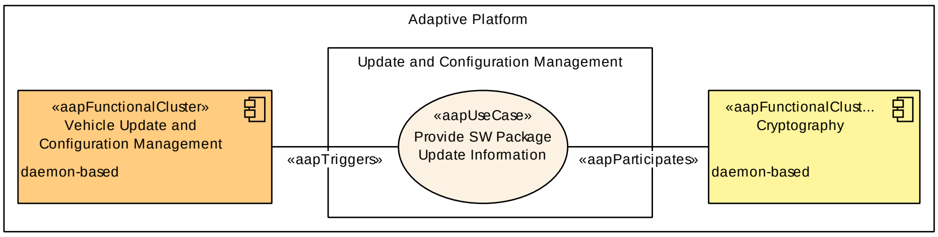

Figure 10.19: Use case for providing update related information

Name:	Provide SW Package Update Information	
Description:	Provide various types of information for various purposes related to software update, for example, about ongoing campaign, history, Software Cluster information, available software for update, etc.	
Triggering Actors:	Vehicle Update and Configuration Management	Vehicle Update and Configuration Management is requested to provide specific information about Software Package update related information on Adaptive Platform.
Participating Actors:	Cryptography	May provide specific cryptographic services like verification of authenticity of an information request.
Preconditions:	• Triggering actors have subscribed successfully to call service method for information request. • Update and Configuration Management is currently not involved in execution of any task with an ongoing processing, activation, or rollback.	
Invariants:	• This use case is supposed to change no data at all.	
Postconditions:	• Data obtained may have a limited (even very short) validity, e.g., those related to an ongoing campaign.	

Table 10.38: Use-Case Provide SW Package Update Information

10.6.2 Vehicle Update and Configuration Management

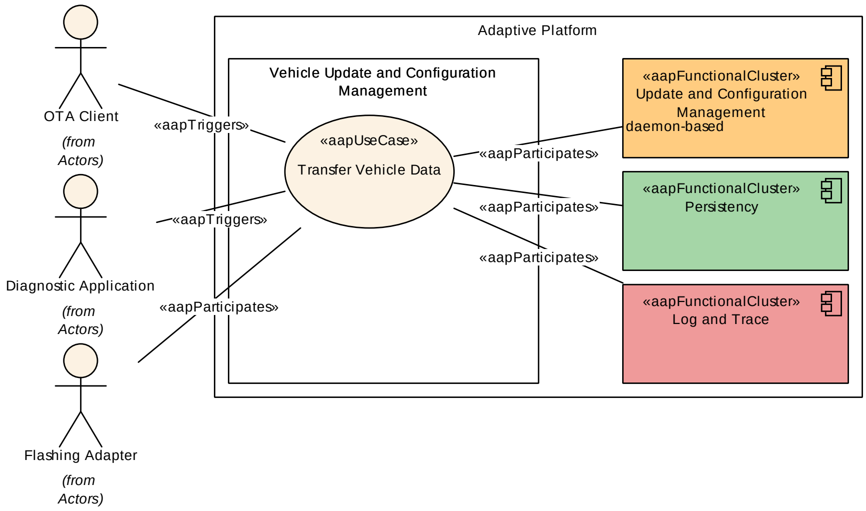

Figure 10.20: Use case Transfer Vehicle Data

Name:	Transfer Vehicle Data	
Description:	Transfer the data for updating SW for the whole vehicle.	
Triggering Actors:	OTA Client	A client that controls package transfer on behalf of a back-end application (external to the vehicle). Vehicle Update and Configuration Management might be the original trigger and query what is available in remote repository/backend and compare with internal state of Software Clusters and versions.
	Diagnostic Application	An application that triggers the package transfer as part of a diagnostic routine on behalf of an external diagnostic client.
Participating Actors:	Update and Configuration Management	Update and Configuration Management optionally transfers software packages to Adaptive Platform target machines (if not done at a later stage).
	Flashing Adapter	Transfers or streams software packages to non-Adaptive Platform target machines (Classic Platform or 3rd-party platform).
	Persistency	Enables persistent storage for data that is in transfer.
	Log and Trace	Logs data transfer related status information.
Preconditions:	• Sufficient memory or persistent storage available to cover transfer of data. • Service APIs for Vehicle Update and Configuration Management have been successfully subscribed to by triggering applications for initiating the data transfer.	

▽

△

Invariants:	• Runtime system (processes, data stored, ...) is not impacted by the transfer.
Postconditions:	• Transfer of data has been completed or state has been stored for a suspended transfer to enable continuation.

Table 10.39: Use-Case Transfer Vehicle Data

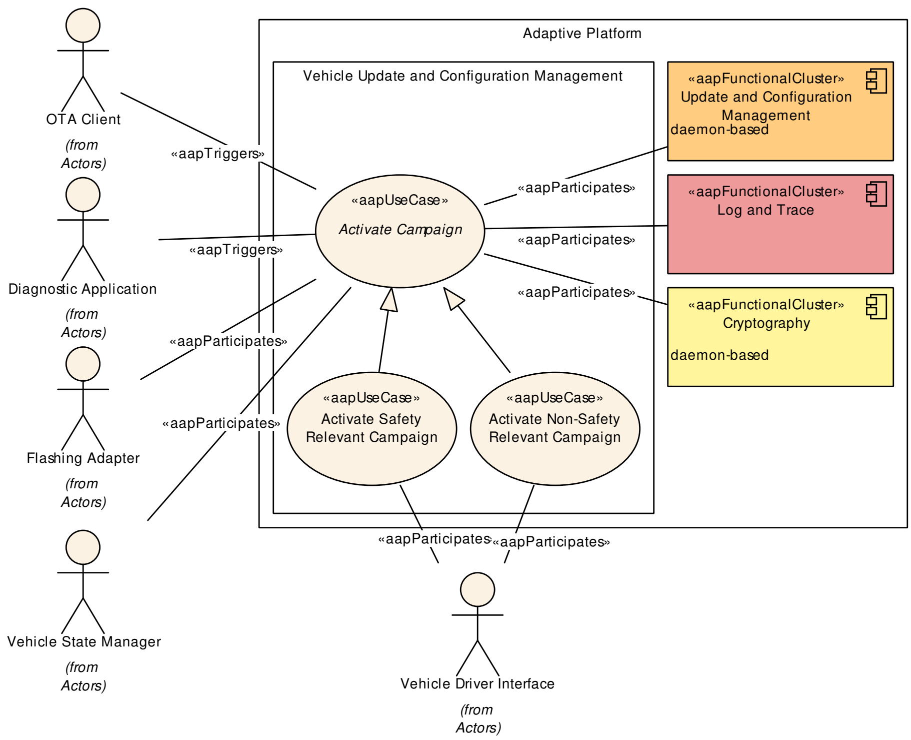

Figure 10.21: Use case Activate Campaign and derived use cases

Name:	Activate Campaign	
Abstract:	Yes	
Description:	Triggers the modification of Software (new installations, updates and removal) on individual machines. This use case is abstract and serves as a base use case for others.	
Triggering Actors:	OTA Client	A client that controls activation in behalf of a trigger from back-end application (external to the vehicle).
	Diagnostic Application	An application that triggers the activation as part of a diagnostic routine in behalf of a diagnostic client.
	Vehicle Driver Interface	An application that triggers the activation in behalf of a user initiated activation.
	Vehicle State Manager	Responds to VehicleCheck request.

▽

△

Participating Actors:	Update and Configuration Management	Update and Configuration Management optionally transfers or streams Software Package(s) (if not done at an earlier stage) and installs and activates it / them on Adaptive Platform target machines.
	Flashing Adapter	Optionally transfers SW Package(s) (if not done at an earlier stage) and installs and activates it / them to non-Adaptive Platform target machines.
	Cryptography	May provide specific cryptographic services like verification of vehicle package authenticity, integrity and decryption of confidential data.
	Log and Trace	Logs activation related status information.
Preconditions:	• Vehicle package data is stored completely and consistently in memory or persistent storage. • Triggering actors have subscribed successfully to call service method for campaign activation. • Currently no other campaign activation or campaign cancellation is executed, which might cause conflicts on any of the local instances (Adaptive Platform, Classic Platform or 3rd-party platform).	
Invariants:	None	
Postconditions:	• All Update and Configuration Management instances involved in the activation have successfully completed the tasks associated with the activation campaign.	

Table 10.40: Use-Case Activate Campaign

Name:	Activate Safety Relevant Campaign	
Description:	Extends the Activate Campaign use case to indicate changes to the safety condition. OEM-specific safe state needs to be ensured for a safety relevant update to take place.	
Participating Actors:	Vehicle State Manager	Checks that the vehicle is in a safe state before allowing safety relevant updates.
	Vehicle Driver Interface	A user optionally gets informed to take some action(s) in order to enable continuation or start of a software modification campaign.
Base use case:	Activate Campaign	
Preconditions:	• Safety relevant update is pending.	
Invariants:	None	
Postconditions:	• Safety relevant update has been completed or cancelled.	

Table 10.41: Use-Case Activate Safety Relevant Campaign

Name:	Activate Non-Safety Relevant Campaign	
Description:	Extends the `Activate Campaign` use case without impact to the safety condition, optionally requiring user interaction.	
Participating Actors:	Vehicle Driver Interface	A user optionally gets informed to take some action(s) in order to enable continuation or start of a software modification campaign.
Base use case:	`Activate Campaign`	
Preconditions:	None	
Invariants:	None	
Postconditions:	None	

Table 10.42: Use-Case Activate Non-Safety Relevant Campaign

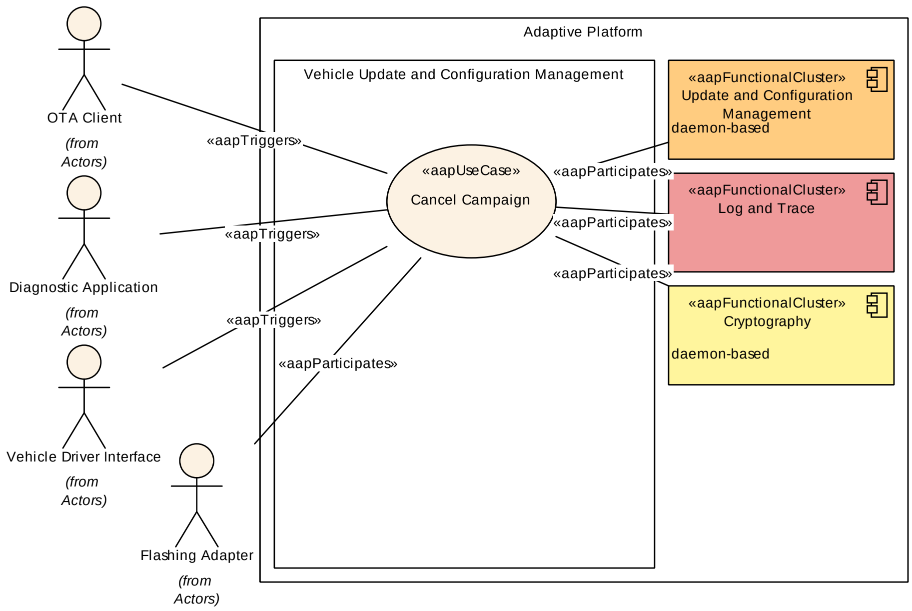

Figure 10.22: Use case Cancel Campaign

Name:	Cancel Campaign	
Description:	Request to cancel an ongoing campaign, which has been requested for activation before.	
Triggering Actors:	OTA Client	A client that triggers the cancellation in behalf of back-end application (external to the vehicle).
	Diagnostic Application	An application that triggers the cancellation as part of a diagnostic routine in behalf of a diagnostic client.
	Vehicle Driver Interface	An application that triggers the cancellation in behalf of a user initiated request.

▽

△

Participating Actors:	Update and Configuration Management	`Update and Configuration Management` is requested to stop any activity associated with the software modification and take actions to reach the state for software package management before the modification of the software package has been triggered.
	Flashing Adapter	Is requested by means of diagnostic routines to revert to the state before software modification has been triggered.
	Cryptography	May provide specific cryptographic services like verification of authenticity of the cancellation request.
	Log and Trace	Logs data transfer related status information.
Preconditions:	• Triggering actors have subscribed successfully to call service method for campaign cancellation. • Currently no other campaign activation or campaign cancellation is executed, which might cause conflicts on any of the local instances (Adaptive Platform, Classic Platform or 3rd-party platform).	
Invariants:	None	
Postconditions:	• All instances involved in the activation have reached the same state as before the software modification has been initiated.	

Table 10.43: Use-Case Cancel Campaign

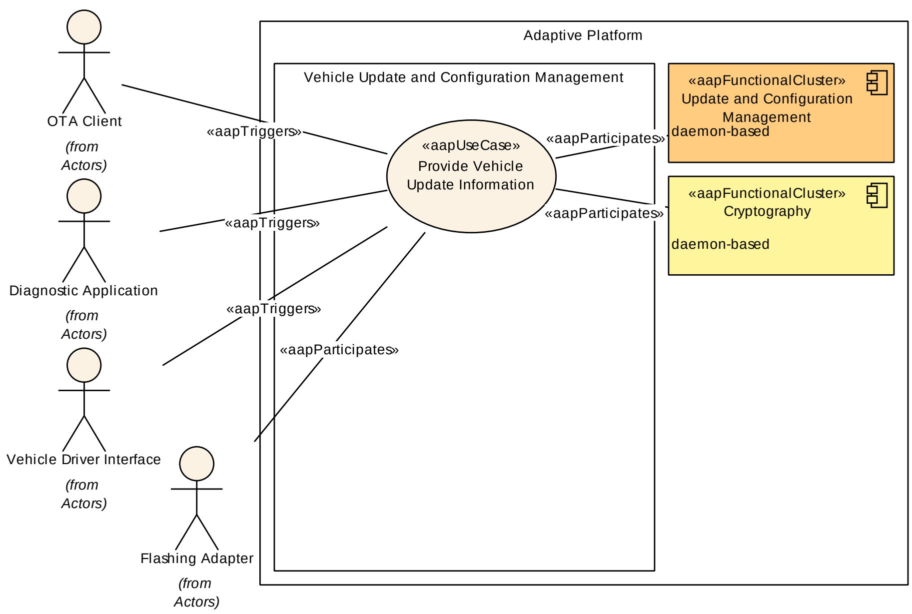

Figure 10.23: Provide Vehicle Update Information

Name:	Provide Vehicle Update Information	
Description:	Provide various types of information for various purposes around vehicle update, for example, about ongoing campaign, history, Software Cluster information, available software for update, etc.	
Triggering Actors:	OTA Client	A client that triggers certain vehicle package update operations may obtain certain type of operational data and may forward this to a back-end application (external to the vehicle).
	Diagnostic Application	An application that triggers certain vehicle package update operations as part of a diagnostic routine in behalf of a diagnostic client.
	Vehicle Driver Interface	An application that receives solicited or unsolicited information related to vehicle package update, that may be displayed to a human user (driver, ...).
Participating Actors:	Update and Configuration Management	`Update and Configuration Management` is requested to provide specific information about software package update related information on Adaptive Platform.
	Flashing Adapter	Is requested to provide specific information about software package update related information on Classic Platform and / or 3rd-party platform.
	Cryptography	May provide specific cryptographic services like verification of authenticity of an information request.
Preconditions:	• Triggering actors have subscribed successfully to call service method for information request.	
Invariants:	• This use case is supposed to change no data at all.	
Postconditions:	• Data obtained may have a limited (even very short) validity, e.g., those related to an ongoing campaign.	

Table 10.44: Use-Case Provide Vehicle Update Information

11 Runtime View

This chapter shows the original design approach of the AUTOSAR Adaptive Platform for implementing selected use cases. The presented use cases currently cover just a small part of the functionality of the AUTOSAR Adaptive Platform. More use cases will be added in future versions of this document. Please note that individual implementations of the AUTOSAR Adaptive Platform may always choose a different design internally. Thus, interaction partners, the type of messages, and their order may differ. Implementation-specific messages are denoted with the stereotype `aapInformative`. All default scenarios show a successful execution of the respective use case. Other scenarios show error cases or alternate cases.

11.1 Runtime

11.1.1 Execution Management

11.1.1.1 Start Adaptive Platform

11.1.1.1.1 Scenario: Start Adaptive Platform with Supervision of Adaptive Applications

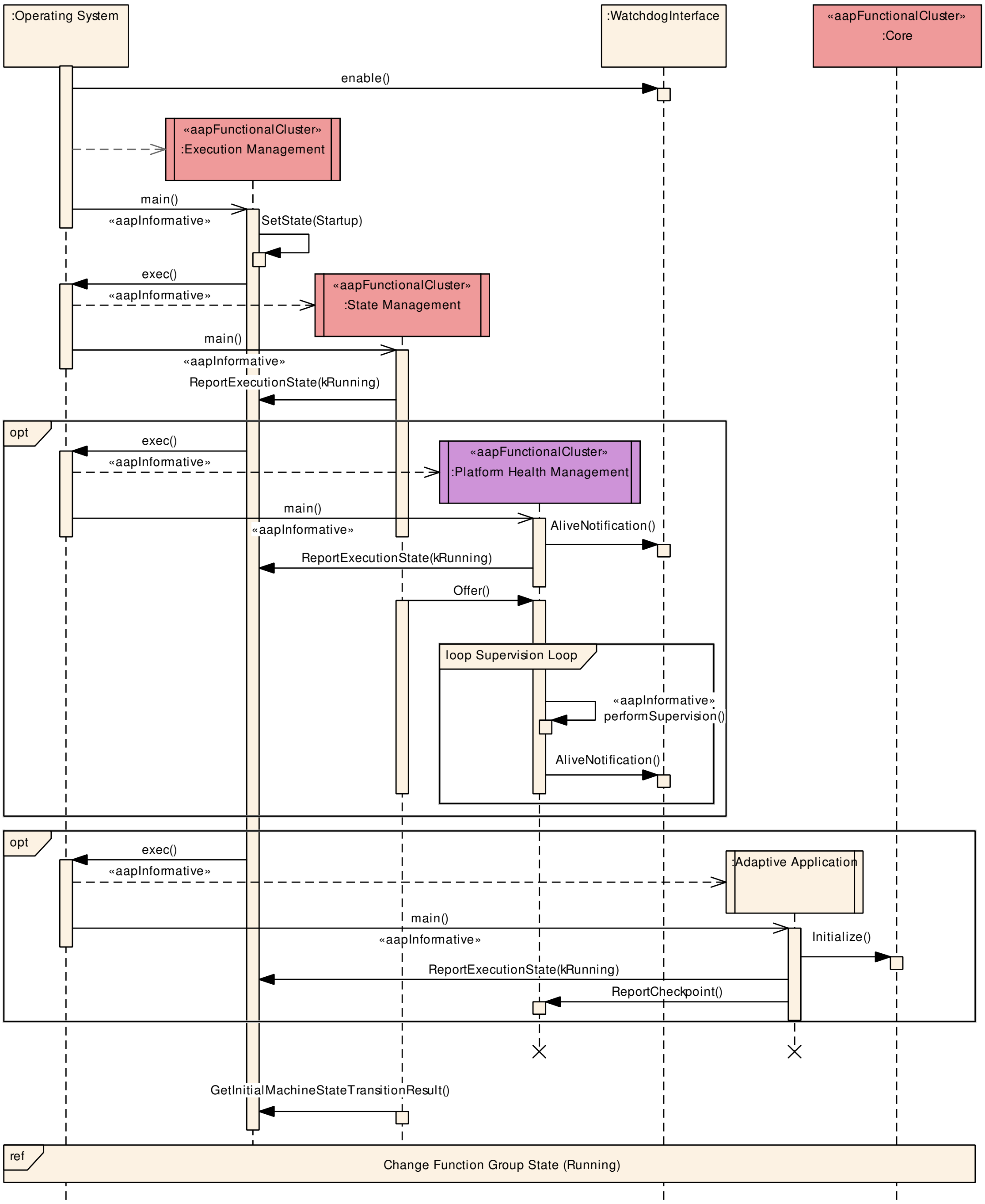

Figure 11.1: Scenario: Start Adaptive Platform with Supervision of Adaptive Applications

Figure 11.1 shows the default scenario for `Start Adaptive Platform` with an supervision of Adaptive Applications. It therefore includes the startup of `Platform Health Management` that performs supervision and the startup of an Adaptive Application that is supervised as part of the Startup Machine Function Group State.

During the preceding startup of the `Machine` the `Operating System` performs initialization steps in an implementation-specific way. These steps include starting any middleware related to the `Operating System`, including device-drivers and services handling low-level middleware. The AUTOSAR Adaptive Platform demands that the Watchdog is enabled prior to the startup of the AUTOSAR Adaptive Platform, for example, the Watchdog could already be enabled by the Bootloader or the Operating System.

Execution Management is started by the Operating System as the first process of the AUTOSAR Adaptive Platform. Execution Management then controls the startup of the AUTOSAR Adaptive Platform by activating the standardized Function Group State called Startup of the Machine Function Group State. This triggers the start of additional processes that are configured to run in the Startup state. It is mandatory that State Management is part of the Startup state. Other processes of the AUTOSAR Adaptive Platform, for example Platform Health Management and application processes may also be part of the Startup state (see Figure 11.1).

Platform Health Management is responsible to service the Watchdog. Thus, the time between enabling the Watchdog during the start of the Machine and the start of Platform Health Management needs to be less than the Watchdog timeout. The integrator needs to fulfill this constraint.

After the Startup state has been reached, State Management takes over control to determine the desired Function Group States.

11.1.1.2 Shutdown Adaptive Platform

11.1.1.2.1 Scenario: Shutdown Adaptive Platform with Supervision of Adaptive Applications

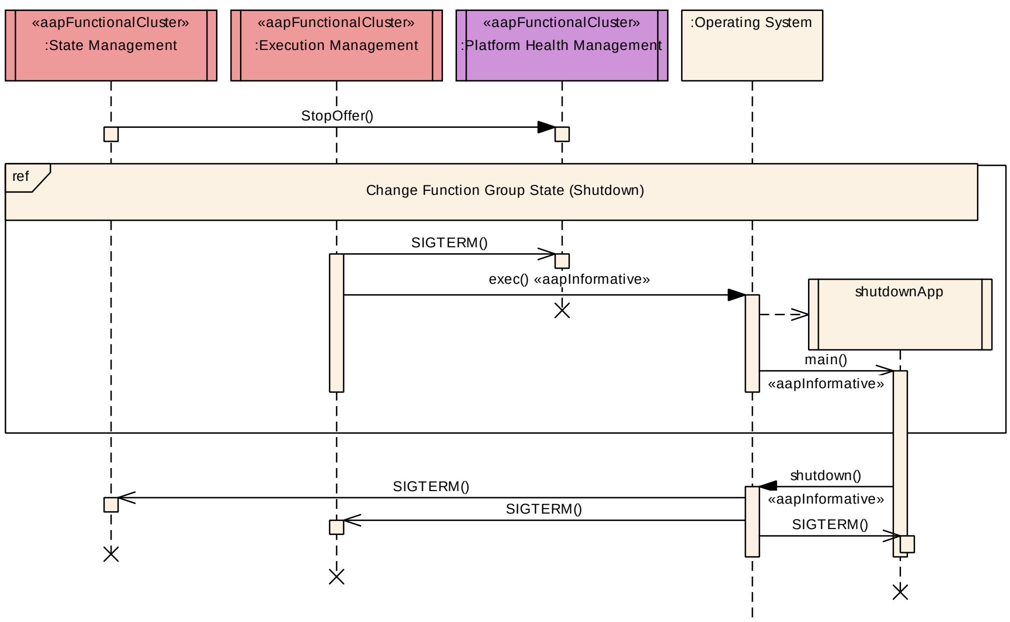

Figure 11.2: Scenario: Shutdown Adaptive Platform with Supervision of Adaptive Applications

Figure 11.2 shows the default scenario for Shutdown Adaptive Platform with a supervision of Adaptive Applications. It therefore includes the shutdown of Platform Health Management that performs supervision.

The shutdown is triggered by State Management by requesting the standardized Machine Function Group State called Shutdown. In general, it is assumed that the only processes configured to run in the Shutdown state are Execution Management, State Management, and an application that issues a shutdown request towards the Operating System (shutdownApp in Figure 11.2). That application is usually specific for the Operating System and/or project and therefore not an Adaptive Application. The use of an external application that performs the shutdown keeps Execution Management free from project-specific functionality. Execution Management will perform an orderly shutdown of the other running application and platform processes (including Platform Health Management) before starting the application process that issues a shutdown request towards the Operating System. The Operating System terminates the remaining processes of the AUTOSAR Adaptive Platform (i.e. State Management, Execution Management) and shuts down the Machine in an implementation-specific way.

11.1.1.3 Change Function Group State

11.1.1.3.1 Scenario: Change Function Group State

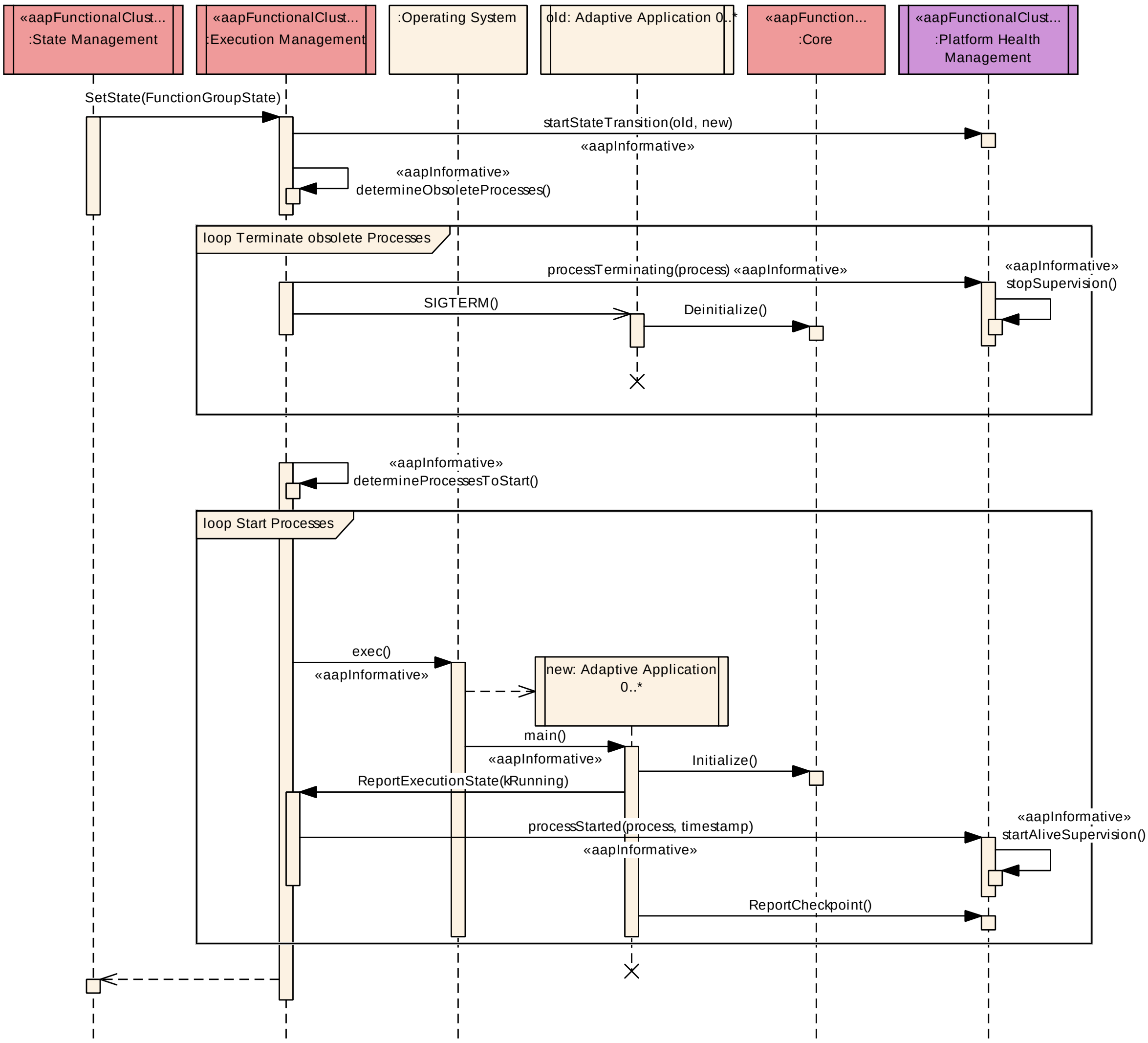

Figure 11.3: Scenario: Change Function Group State

Figure 11.3 shows the default scenario for Change Function Group State. The scenario is triggered by State Management. Execution Management will terminate all processes that are not part of the requested target Function Group State or that have a different StateDependentStartupConfig.

Just before terminating a process (with a SIGTERM signal), Execution Management notifies Platform Health Management that will stop all supervisions of the process. Consequently, State Management will not receive any information about failed supervisions during the shutdown of the process. The shutdown is monitored by Execution Management by means of StartupConfig.timeout configured in the Manifest.

Afterwards, Execution Management starts the processes of the target Function Group State in the order imposed by their StateDependentStartupConfig. When a processes reports its execution state as kRunning, Execution Management notifies Platform Health Management to start Alive Supervision for that process. Other kinds of supervisions (Deadline Supervision, and Logical Supervision) are started when the first checkpoint is reported for them.

11.1.2 State Management

11.1.2.1 Change System State

11.1.2.1.1 Scenario: Change System State

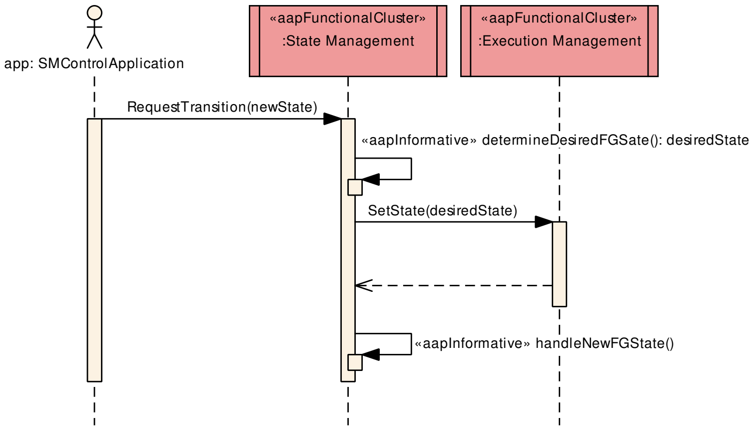

Figure 11.4: Scenario: Change System State

Figure 11.4 shows the default scenario for Change System State. An SMControlApplication calls RequestTransition with a parameter indicating the requested transition. Based on the requested transition, State Management determines a new desired system state, that is a set of desired Function Group States, and requests these Function Group States from Execution Management by calling SetState().

11.1.2.2 Recover from Supervision Failure

11.1.2.2.1 Scenario: Recover by System State Change

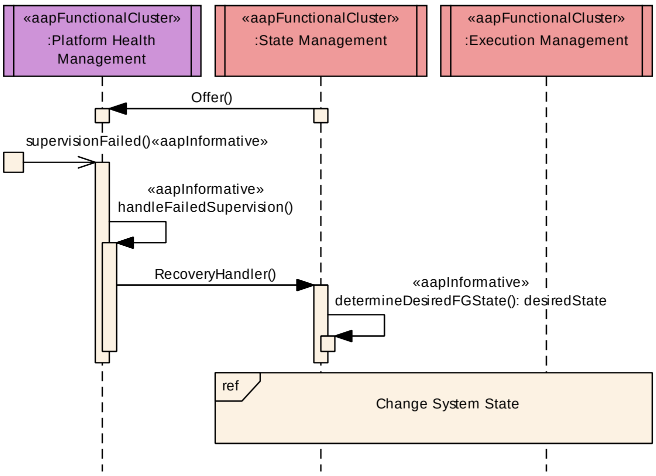

Figure 11.5: Scenario: Recover by System State Change

Figure 11.5 shows the default scenario for Recover from Supervision Failure. After a supervision failure has been detected by Platform Health Management, Platform Health Management notifies State Management by invoking the call-back method RecoveryHandler(). State Management then determines a new desired state (which may be the same as the current state) and requests corresponding Function Group State transitions from Execution Management.

11.1.2.2.2 Scenario: Recover by Watchdog Reset

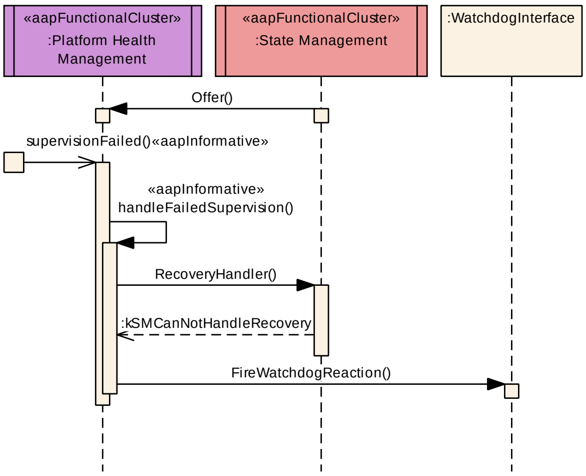

Figure 11.6: Scenario: Recover by Watchdog Reset

Figure 11.6 shows an alternate scenario for Recover from Supervision Failure. After a supervision failure has been detected by Platform Health Management, Platform Health Management notifies State Management by invoking the call-back method RecoveryHandler(). In this scenario, State Management cannot handle the failure and returns the error code kSMCanNotHandleRecovery from the call-back. Platform Health Management then handles this failure by triggering a Watchdog reset.

11.2 Communication

11.2.1 Raw Data Stream

11.2.1.1 Send Raw Data Stream

11.2.1.1.1 Scenario: Send IEEE1722 Stream

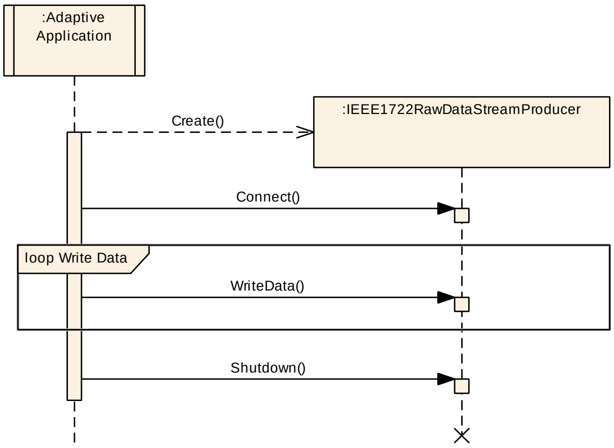

Figure 11.7: Scenario: Send IEEE1722 Stream

Figure 11.7 shows the scenario for Send Raw Data Stream that makes use of the IEEE1722 protocol. The Adaptive Application needs to connect to an IEEE1722 stream using Connect(). Afterwards, the Adaptive Application may transmit any number of datagrams by calling WriteData(). If the stream is no longer needed, the Adaptive Application may shutdown the connection using Shutdown().

11.2.1.1.2 Scenario: Send IP Stream

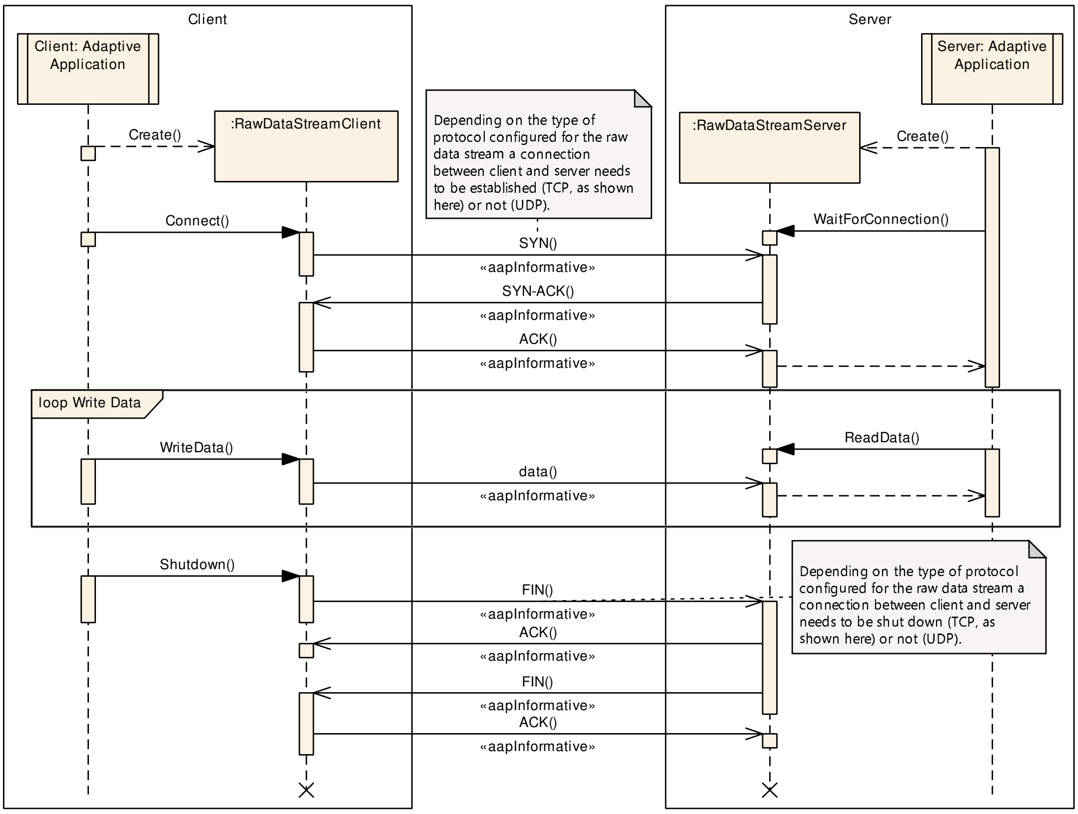

Figure 11.8: Scenario: Send IP Stream

Figure 11.8 shows the scenario for Send Raw Data Stream that makes use of an IP-based protocol (i.e., TCP or UDP). The Client Adaptive Application needs to connect to a Server using Connect(). If TCP is configured as a protocol, the underlying TCP connection will be established with the Server. Afterwards, the Client Adaptive Application may transmit data to the Server by calling WriteData(). The Server Adaptive Application may receive the data transmitted by the Client by calling ReadData(). The segmentation of data is handled internally. Please note that Client and Server are both able to receive and transmit via the same stream for some types of protocol configurations (e.g., TCP). If the stream is no longer needed, the Client Adaptive Application may shutdown the connection using Shutdown().

11.2.1.2 Receive Raw Data Stream

11.2.1.2.1 Scenario: Receive IEEE1722 Stream

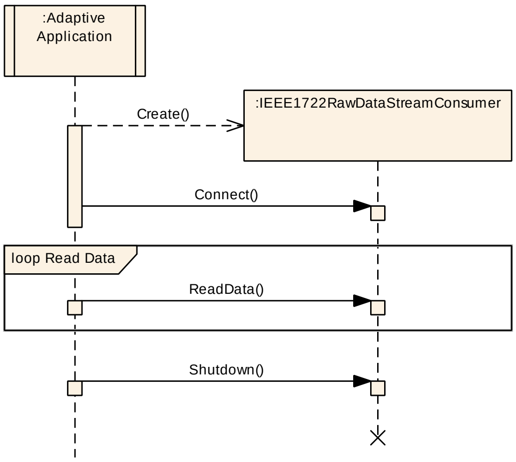

Figure 11.9: Scenario: Receive IEEE1722 Stream

Figure 11.9 shows the scenario for Receive Raw Data Stream that makes use of the IEEE1722 protocol. The Adaptive Application needs to connect to an IEEE1722 stream using Connect(). Afterwards, the Adaptive Application may receive any number of datagrams by calling ReadData(). If the stream is no longer needed, the Adaptive Application may shutdown the connection using Shutdown().

11.2.1.2.2 Scenario: Receive IP Stream

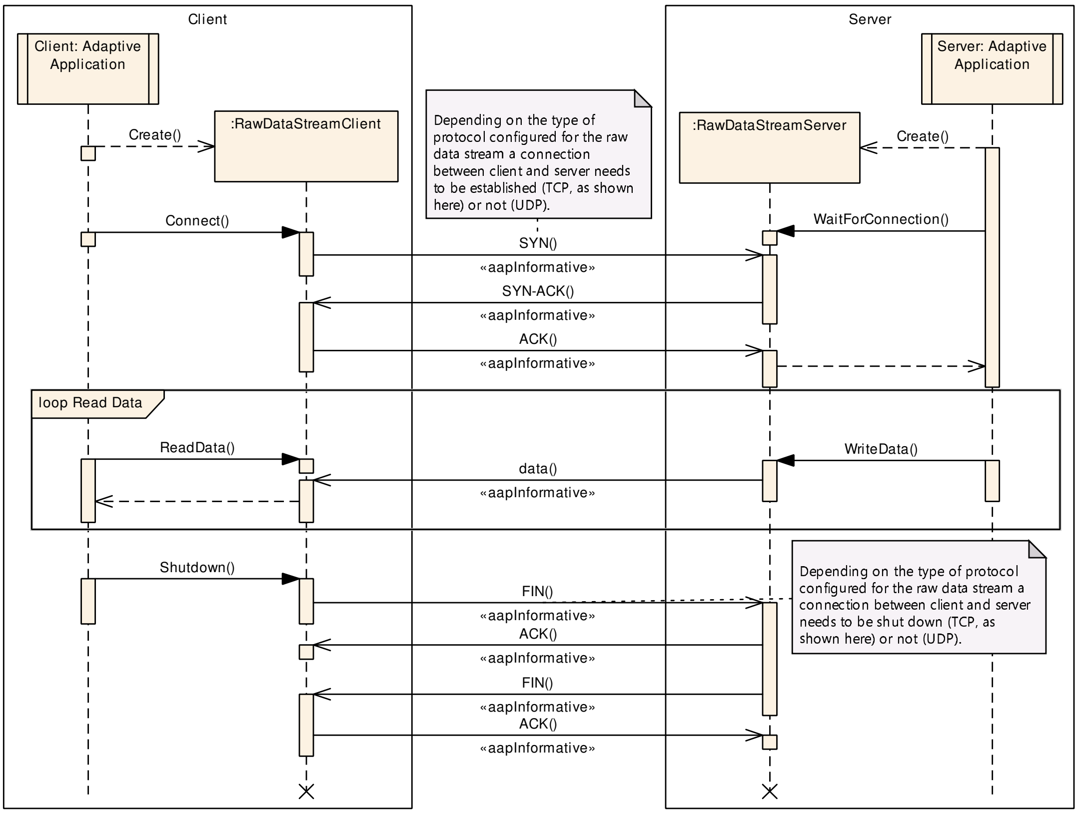

Figure 11.10: Scenario: Receive IP Stream

Figure 11.10 shows the scenario for `Receive Raw Data Stream` that makes use of an IP-based protocol (i.e., TCP or UDP). The Client `Adaptive Application` needs to connect to a Server using `Connect()`. If TCP is configured as a protocol, the underlying TCP connection will be established with the Server. Afterwards, the Server `Adaptive Application` transmits data by calling `WriteData()`. The Client `Adaptive Application` may receive the data transmitted by the Server by calling `ReadData()`. The segmentation of data is handled internally. Please note that Client and Server are both able to receive and transmit via the same stream for some types of protocol configurations (e.g., TCP). If the stream is no longer needed, the Client `Adaptive Application` may shutdown the connection using `Shutdown()`.

11.2.2 Time Synchronization

This section shows interactions for Time Synchronization Master (see Section 11.2.2.1) and Time Synchronization Slave (see Section 11.2.2.2).

11.2.2.1 Time Synchronization Master

11.2.2.1.1 Set Value of Time base

11.2.2.1.1.1 Scenario: Set Value Of Time Base

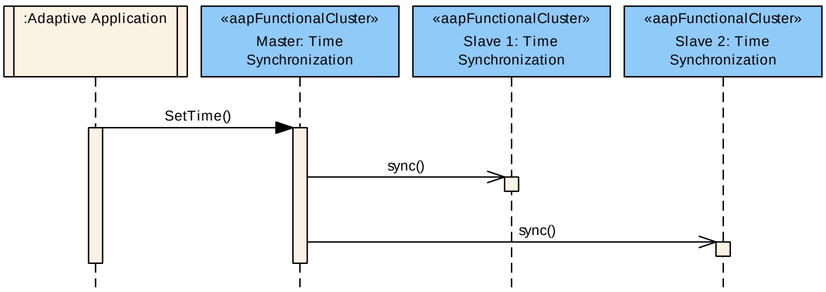

Figure 11.11: Scenario: Set Value Of Time Base

Figure 11.11 shows the default scenario for `Set Value Of Time Base`. The `Adaptive Application` sets a value for the Time Base in Time Synchronization Master by calling `SetTime()`. Then, Time Synchronization Master distributes the updated time value with all other connected Time Synchronization Slaves.

11.2.2.2 Time Synchronization Slave

11.2.2.2.1 Get Value of Time base

11.2.2.2.1.1 Scenario: Get Value Of Time Base

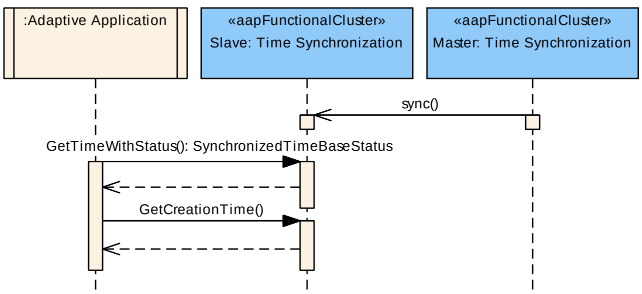

Figure 11.12: Scenario: Get Value of Time Base

Figure 11.12 shows the default scenario for `Get Value of Time Base`. The Time Synchronization Slave already knows the current time due to a prior synchronization message from Time Synchronization Master. Once the `Adaptive Application` calls the method `GetTimeWithStatus()`, the Time Synchronization Slave provides an `SynchronizedTimeBaseStatus` object that reflects the time (`GetCreationTime()`) and the synchronization status (`GetSynchronizationStatus()`) when that object was created.

11.2.2.2.2 Receive Notification

11.2.2.2.2.1 Scenario: Receive Notification

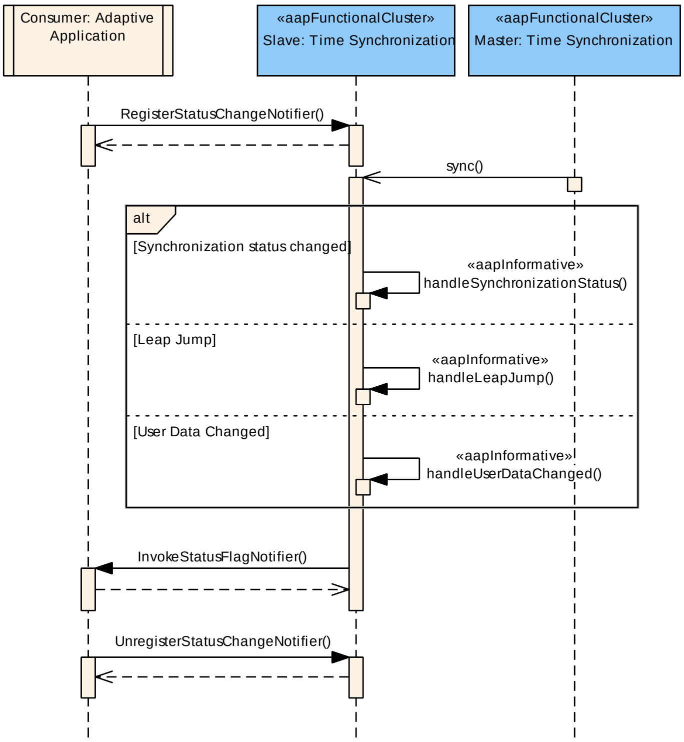

Figure 11.13: Scenario: Receive Notification

Figure 11.13 shows the default scenario for `Receive Notification`. A notifier must be registered with the Time Synchronization Slave by calling `RegisterStatusChangeNotifier()` as a precondition for `Adaptive Application` to be able to receive a notification. The Time Synchronization Slave has the current time available thanks to the synchronization from the Time Synchronization Master. Then, one of 3 cases can occur:

1. The synchronization status changed in Time Synchronization Slave.
2. A leap jump was detected.
3. The user data associated with the time base changed.

After the occurrence one of these 3 cases, the Time Synchronization Slave informs the `Adaptive Application` about the change by invoking the call-back function. If the `Adaptive Application` no longer wants to receive notifications it may unregister by calling the method `UnregisterStatusChangeNotifier()`.

11.3 Storage

11.3.1 Persistency

11.3.1.1 Read Persistent Data from File

11.3.1.1.1 Scenario: Read Persistent Data from File

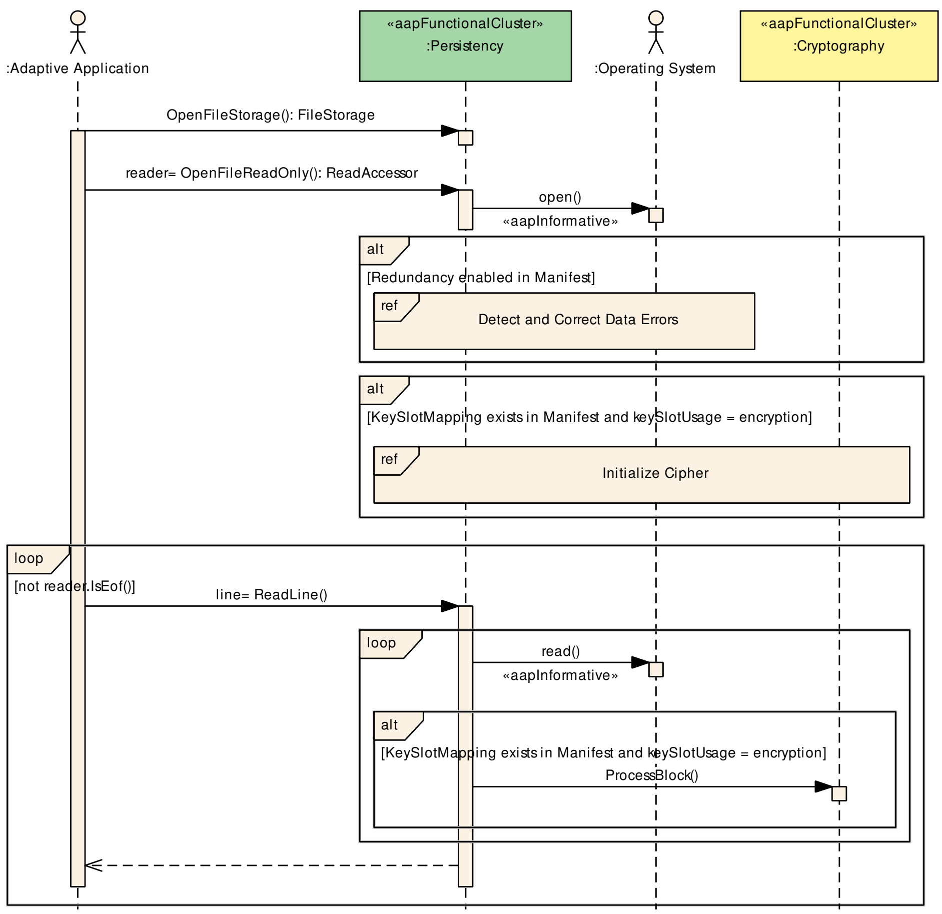

Figure 11.14: Scenario: Read Persistent Data from File

Figure 11.14 shows the default scenario for `Read Persistent Data from File`. The `Adaptive Application` needs to open a `FileStorage` by calling `OpenFileStorage()`. Afterwards, the `Adaptive Application` needs to open an individual file by calling `OpenFileReadOnly()` or `OpenFileReadWrite()` depending if the `Adaptive Application` needs to write data to the file as well. Then, the `Adaptive Application` can read data via the methods provided by `ReadAccessor` either as binary data or text data.

If enabled in the `Manifest`, `Persistency` will detect and try to correct data errors when reading from a file. If the file or `FileStorage` is configured to use encryption, the contents of the file will be transparently decrypted during read.

11.3.1.2 Read Persistent Data with Unique ID

11.3.1.2.1 Scenario: Read Persistent Data with Unique ID

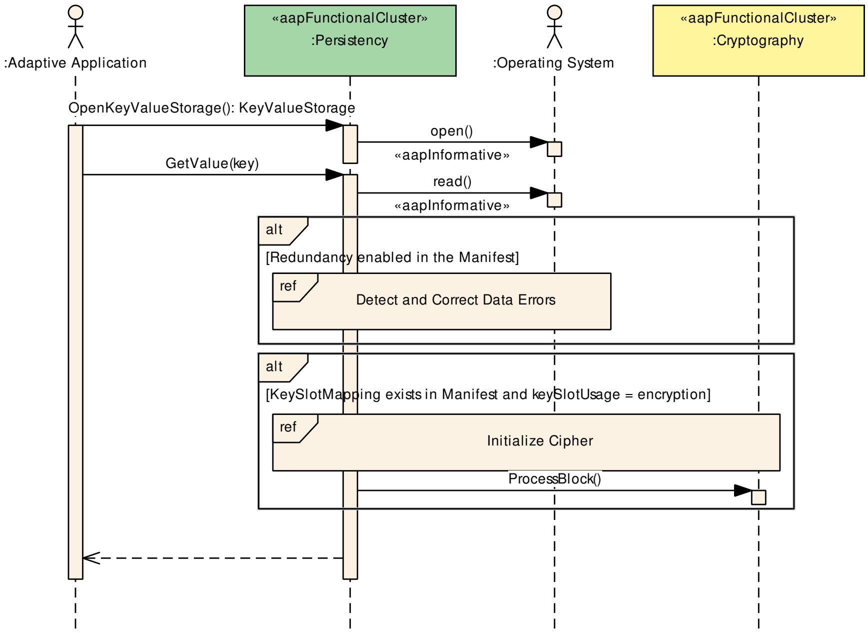

Figure 11.15: Scenario: Read Persistent Data with Unique ID

Figure 11.15 shows the default scenario for `Read Persistent Data with Unique ID`. The `Adaptive Application` needs to open a `KeyValueStorage` by calling `OpenKeyValueStorage()`. Then, the `Adaptive Application` can read data associated to a key by calling `GetValue()`.

If enabled in the `Manifest`, `Persistency` will detect and try to correct data errors when reading data. If the individual key or `KeyValueStorage` is configured to use encryption, data will be transparently decrypted during read.

11.3.1.3 Store Persistent Data in File

11.3.1.3.1 Scenario: Store Persistent Data in File

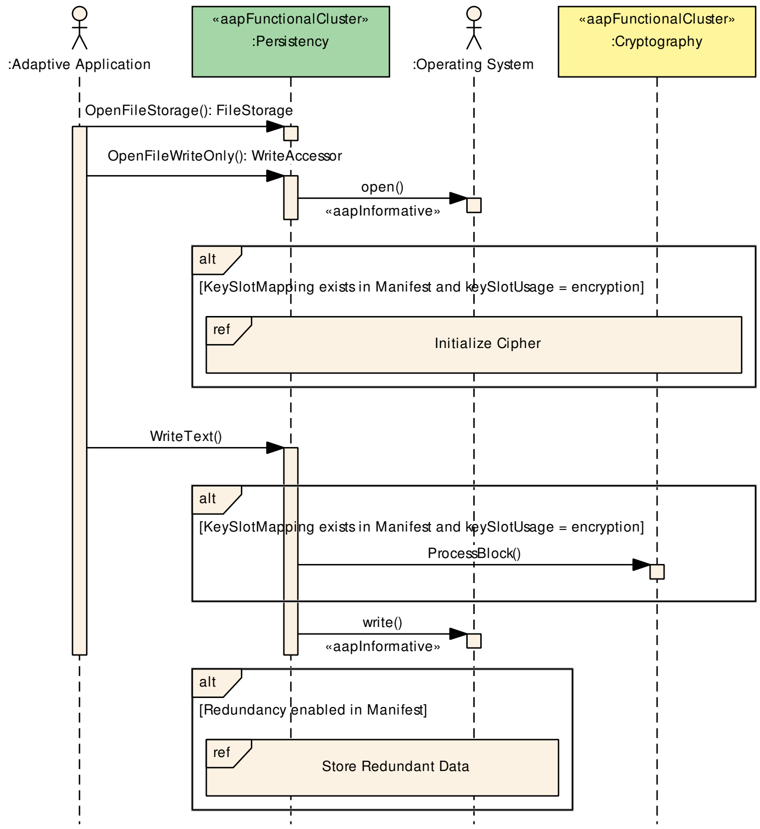

Figure 11.16: Scenario: Store Persistent Data in File

Figure 11.16 shows the default scenario for `Store Persistent Data in File`. The `Adaptive Application` needs to open a `FileStorage` by calling `OpenFileStorage()`. Afterwards, the `Adaptive Application` needs to open an individual file for writing by calling `OpenFileWriteOnly()` or `OpenFileReadWrite()` depending if the `Adaptive Application` needs to read data from the file as well. Then, the `Adaptive Application` can store data via the methods provided by `WriteAccessor` or `ReadWriteAccessor` either as binary data or text data.

If enabled in the `Manifest`, `Persistency` stores redundant data to detect and correct errors when reading from a file later. If the file or `FileStorage` is configured to use encryption, data will be encrypted before it is written to the underlying storage.

11.3.1.4 Store Persistent Data with Unique ID

11.3.1.4.1 Scenario: Store Persistent Data with Unique ID

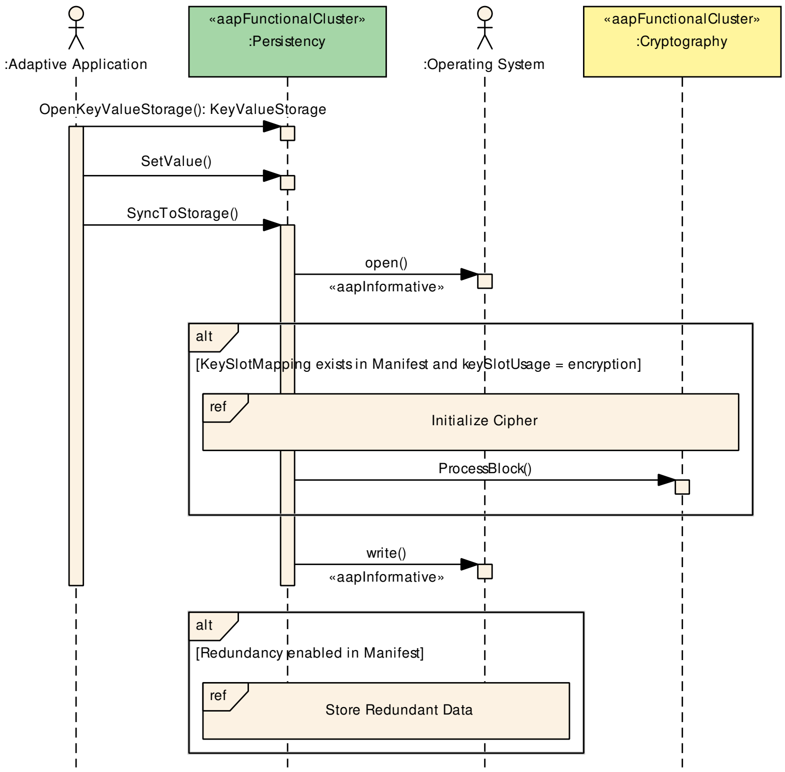

Figure 11.17: Scenario: Store Persistent Data with Unique ID

Figure 11.17 shows the default scenario for Store Persistent Data with Unique ID. The Adaptive Application needs to open a KeyValueStorage by calling OpenKeyValueStorage(). Then, the Adaptive Application can store data associated to a key by calling SetValue(). The data is updated in memory by calling SetValue() but not written to persistent storage. The Adaptive Application needs to call SyncToStorage() to write one or more such changes to persistent storage.

If enabled in the Manifest, Persistency stores redundant data to detect and correct errors when reading from a KeyValueStorage later. If the key or KeyValueStorage is configured to use encryption, data will be encrypted before it is written to the underlying storage.

11.3.1.5 Get Storage Size

11.3.1.5.1 Scenario: Get FileStorage Size

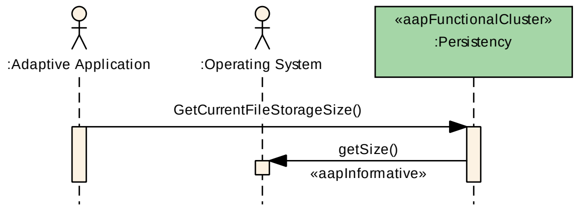

Figure 11.18: Scenario: Get FileStorage Size

Figure 11.18 shows the scenario of the use case `Get Storage Size` in case of a `FileStorage`. The `Adaptive Application` needs to call `GetCurrentFileStorageSize()` to determine the current size.

11.3.1.5.2 Scenario: Get KeyValueStorage Size

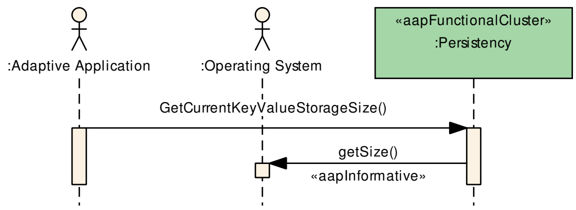

Figure 11.19: Scenario: Get KeyValueStorage Size

Figure 11.19 shows the scenario of the use case `Get Storage Size` in case of a `KeyValueStorage`. The `Adaptive Application` needs to call `GetCurrentKeyValueStorageSize()` to determine the current size.

11.4 Security

11.4.1 Firewall

11.4.1.1 Switch Firewall State

11.4.1.1.1 Scenario: Switch Firewall State

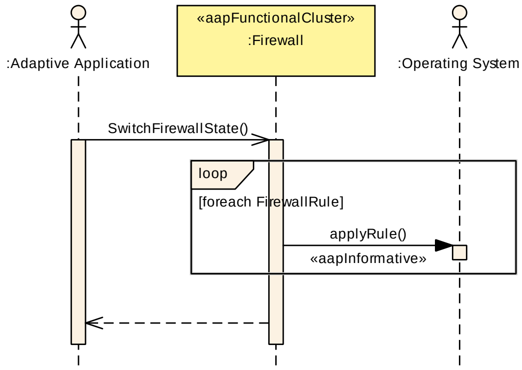

Figure 11.20: Scenario: Switch Firewall State

Figure 11.20 shows the default scenario for Switch Firewall State. The Adaptive Application triggers the state switch by calling SwitchFirewallState() . The Firewall will then apply the FirewallRules configured for the request state in an implementation-specific way (e.g., using tools provided with the TCP/IP stack of the Operating System).

11.4.1.2 Report Security Event

11.4.1.2.1 Scenario: Report Security Event

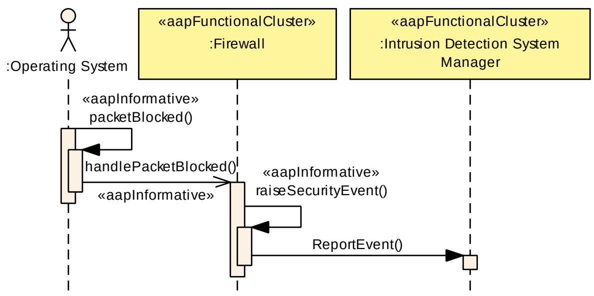

Figure 11.21: Scenario: Report Security Event

Figure 11.21 shows the default scenario for Report Security Event. The Operating System (or another component that implements the actual firewall) reports that a packet has been blocked by a specific rule to the Firewall. If a Security Event has been configured for that rule in the Manifest, the Firewall will create a corresponding Security Event and report it to Intrusion Detection System Manager by calling ReportEvent(). Intrusion Detection System Manager will then handle the Security Event accordingly.

11.5 Safety

11.5.1 Platform Health Management

11.5.1.1 Manage Watchdog

11.5.1.1.1 Scenario: Manage Watchdog

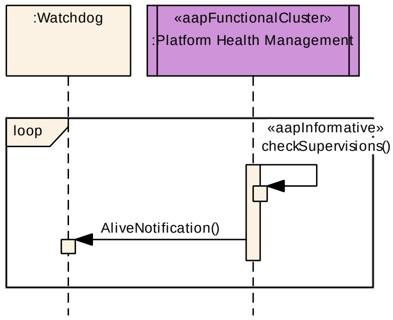

Figure 11.22: Scenario: Manage Watchdog

Figure 11.22 shows the default scenario for `Manage Watchdog`. `Platform Health Management` regularly checks its supervisions. `Platform Health Management` also calls the method `AliveNotification()` cyclically in order to report aliveness to the `Watchdog`.

11.5.1.2 Report Supervision Failure

11.5.1.2.1 Scenario: Report Supervision Failure

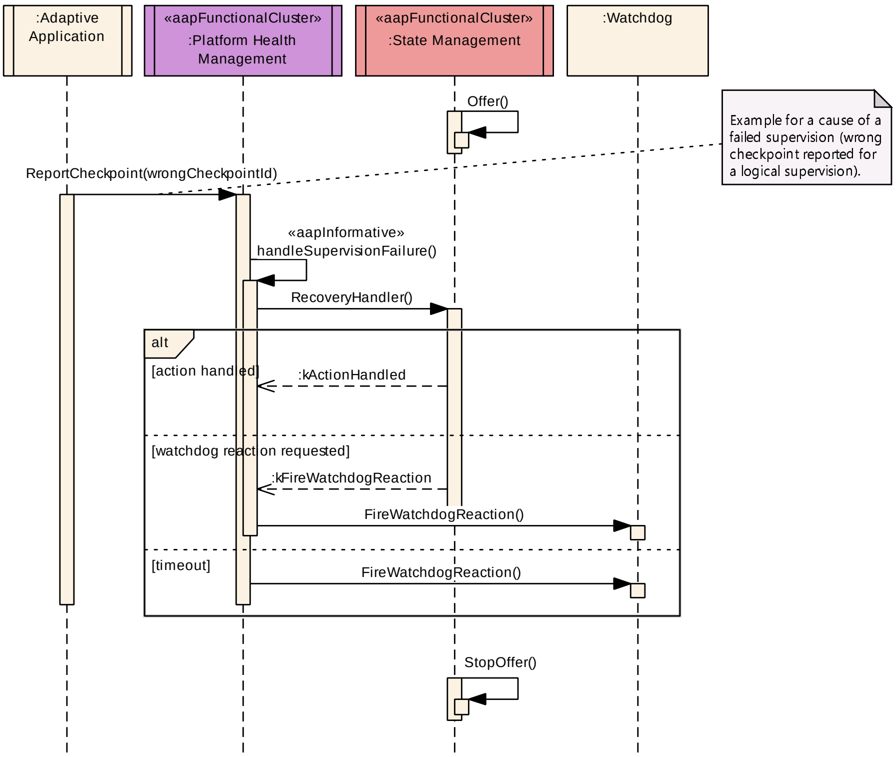

Figure 11.23: Scenario: Report Supervision Failure

Figure 11.23 illustrates the default scenario for Report Supervision Failure. As a precondition, State Management needs to call the method Offer() in order to register for notifications.

Whenever Platform Health Management detects a supervision failure it notifies State Management by invoking the callback method RecoveryHandler(). If the call to State Management does not return within the timeout configured in the Manifest, Platform Health Management needs to call the method FireWatchdogReaction() to perform a restart of the Machine.

During its shutdown State Management needs to call the method StopOffer() to disable further invocations of RecoveryHandler().

11.5.1.3 Recover from Critical Supervision Failure

11.5.1.3.1 Scenario: Recover from Critical Supervision Failure in Execution Management

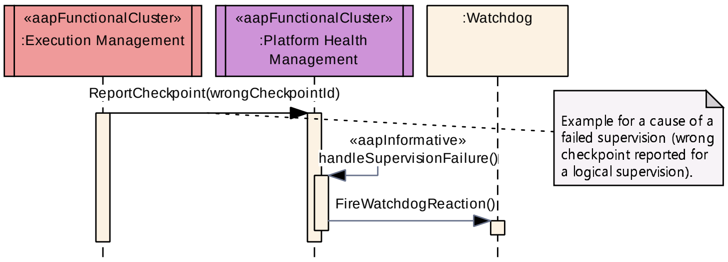

Figure 11.24: Scenario: Recover from Critical Supervision Failure in Execution Management

The figure 11.24 shows a scenario for Report Supervision Failure that handles recovery from a critical supervision failure in Execution Management. Such failures are considered as critical failures because Execution Management is required to recover from supervision failures. Thus, a supervision failure in Execution Management results in Platform Health Management calling the method FireWatchdogReaction() to perform a restart of the Machine.

11.5.1.3.2 Scenario: Recover from Critical Supervision Failure in State Management

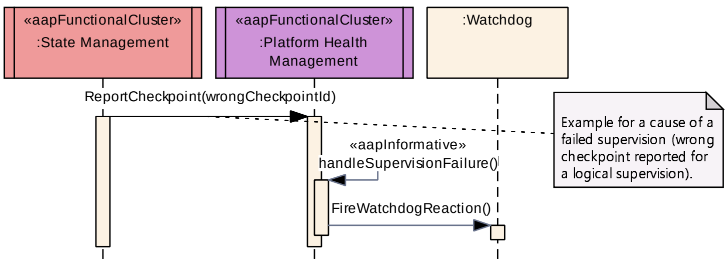

Figure 11.25: Scenario: Recover from Critical Supervision Failure in State Management

The figure 11.25 shows a scenario for Report Supervision Failure that handles recovery from a critical supervision failure in State Management. Such failures are considered as critical failures because State Management is required to recover from supervision failures. Thus, a supervision failure in State Management results in Platform Health Management calling the method FireWatchdogReaction() to perform a restart of the Machine.

11.5.1.4 Report Checkpoint

11.5.1.4.1 Scenario: Report Checkpoint

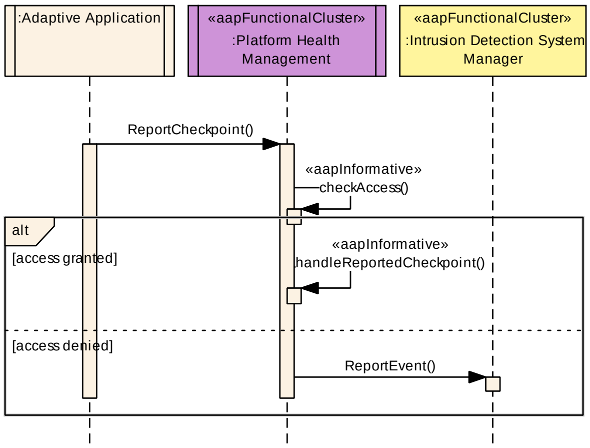

Figure 11.26: Scenario: Report Checkpoint

Figure 11.26 shows the default scenario for Report Checkpoint. The Adaptive Application reports a checkpoint to Platform Health Management by calling the method ReportCheckpoint(). After that, Platform Health Management checks if the Adaptive Application is allowed to report the checkpoint. Then, Platform Health Management handles the reported checkpoint internally depending on the type of the supervision mode.

If access is denied, Platform Health Management reports this event to Intrusion Detection System Manager by calling the method ReportEvent().

12 Deployment View

This chapter provides an overview of exemplary deployment scenarios for an AUTOSAR Adaptive Platform. Since the AUTOSAR Adaptive Platform is highly configurable in its deployment, this section rather provides constraints on supported deployments and a selection of relevant deployment scenarios.

12.1 Vehicle Software Deployment

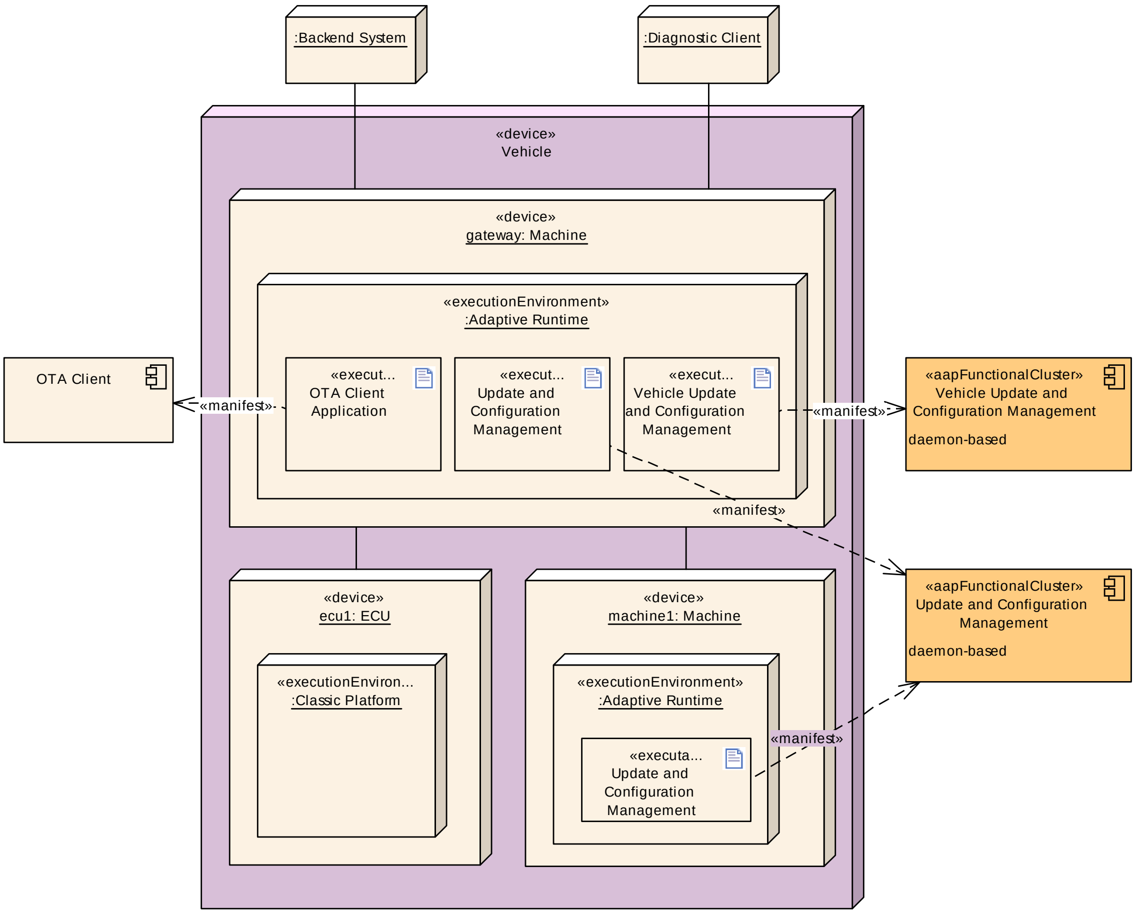

Figure 12.1: Exemplary vehicle software update scenario

`Vehicle Update and Configuration Management` allows to install and update software within an entire vehicle. The vehicle packages are transferred (over-the-air) by an application `OTA Client` to `Vehicle Update and Configuration Management`. `Vehicle Update and Configuration Management` then controls the update process in the vehicle and distributes individual software packages to the `Machine`s and `ECU`s within a vehicle.

12.2 Deployment of Software Packages on a Machine

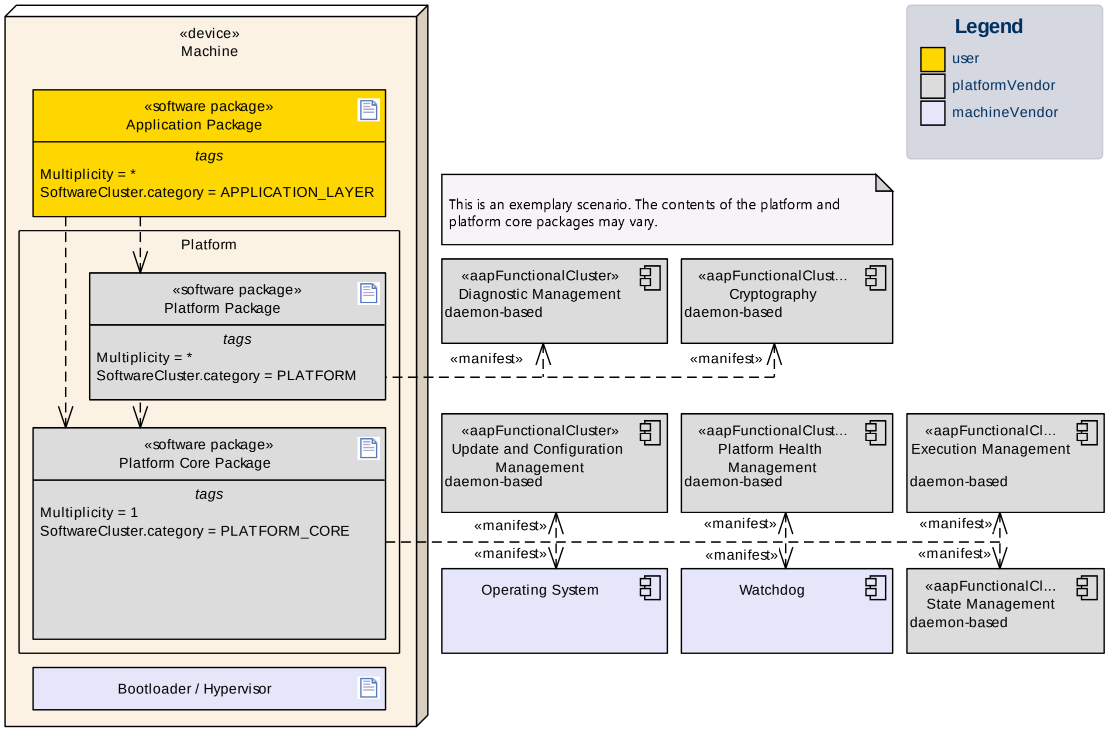

Figure 12.2: Exemplary deployment of Software Packages on a Machine

Update and Configuration Management manages the software installed on a Machine using one or more Software Packages. The configuration of the contents of a Software Package is done in the Manifest by means of a Software Cluster. Software Cluster defines the enumeration attribute category with the values APPLICATION_LAYER, PLATFORM, and PLATFORM_CORE. Some rules apply on the use of the Software Cluster category and corresponding Software Packages which are outlined in the following.

For a Machine there shall exist exactly one Software Cluster with category PLATFORM_CORE. The corresponding Platform Core Package needs to include all required core components provided by the Platform Vendor and Machine Vendor, for example the operating system, device drivers, and the required Functional Clusters of the Adaptive Platform (see Figure 12.2). It is assumed that at least the Functional Clusters Execution Management, State Management, and Update and Configuration Management are part of the Platform Core Package since they are required to install any additional Software Packages. However, the concrete extend of a Platform Core Package is vendor and/or project specific. A Platform Core Package cannot be removed. The Software Cluster with category PLATFORM_CORE needs to be self-contained and therefore shall not have dependencies to other Software Clusters.

Optional Functional Clusters of the Adaptive Platform may be distributed as part of any number of additional Software Clusters with category PLATFORM within Platform Packages. Such Software Clusters may depend on the the

Software Cluster with category PLATFORM_CORE (see Figure 12.2). Platform Packages may be installed and removed as needed.

Application components should be distributed as part of additional Software Clusters with category APPLICATION_LAYER within Application Packages. Such Software Clusters may depend on the on Software Clusters with any kind of category (see Figure 12.2). Application Packages may be installed and removed as needed.

The Platform Core Package and Platform Packages shall include all required Functional Cluster daemons and their respective configuration (e.g., startup configuration). This frees applications (distributed as Application Packages) from taking care of the configuration of the platform.

13 Cross-cutting Concepts

This section provides an overview of cross-cutting concepts and patterns used in the AUTOSAR Adaptive Platform.

13.1 Overview of Platform Entities

The AUTOSAR Adaptive Platform defines design entities that several Functional Clusters depend on. Figure 13.1 provides an overview of these entities, their logical relationships, and the Functional Clusters that depend on them. For the sake of brevity, this overview uses simplifications and abstractions over the actual specifications in the [9, manifest specification].

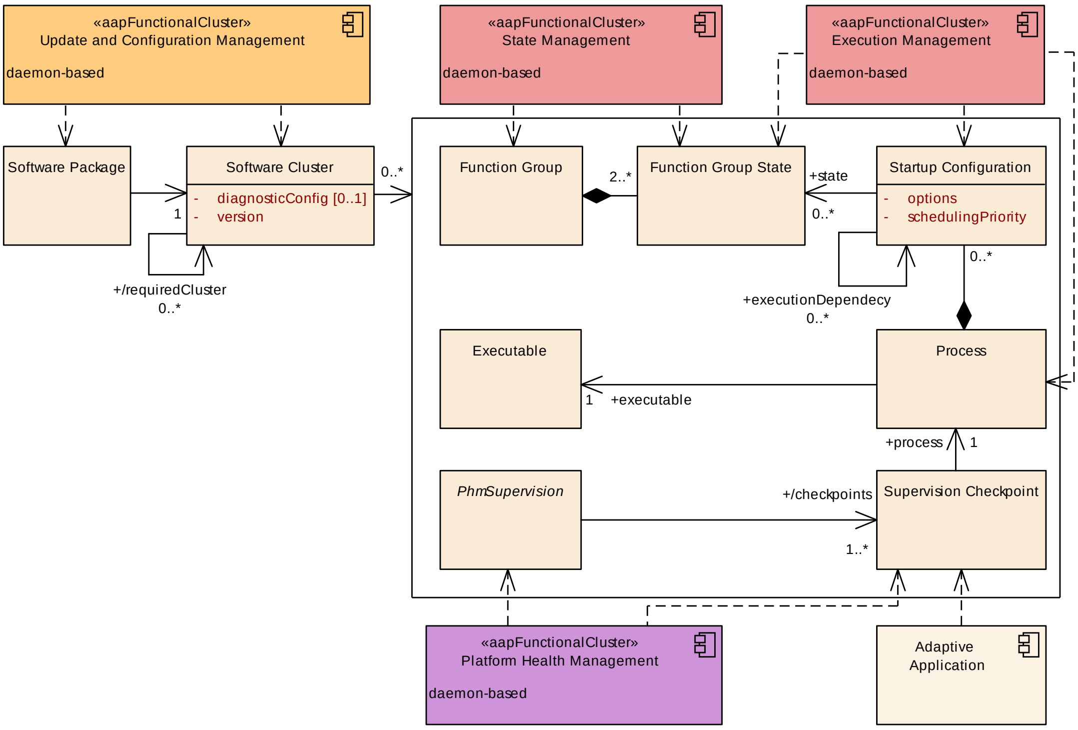

Figure 13.1: Overview of platform entities and their logical relationships

A Software Package is a digitally signed package that can be installed/uninstalled via Update and Configuration Management. A Software Package contains exactly one Software Cluster (see Section 13.4 for details). A Software Cluster refers to a set of Executables (and other files). The corresponding executable file then holds the executable code for the Machine that the AUTOSAR Adaptive Platform runs on.

Additionally, a Software Cluster configuration collects a set of Processes (cf. Section 13.4) and related entities. A Process refers to an Executable and provides different Startup Configuration values, for example parameters, a scheduling priority, and resource constraints. A Startup Configuration of a Process applies to one or more Function Group States. Function Group States belong to a Function Group.

During runtime, State Management requests to enter a Function Group State from Execution Management. Execution Management then terminates and starts the Processes accordingly using the underlying Operating System.

For safety-critical systems, Platform Health Management performs supervision of Processes according to rules (logical sequence, deadlines) defined in PhmSupervisions. A PhmSupervision refers to a number of Supervision Checkpoints. During runtime, a process reports whenever it has reached such a checkpoint in its control flow.

13.2 Function Group

A Function Group (meta-class ModeDeclarationGroup) is (logically) made up a set of modeled Processes that provide a certain functionality. For example, a Function Group could be an application, or a service. A special Function Group is the Machine State that groups the Process of the AUTOSAR Adaptive Platform itself. A Function Group contains a set of Function Group States.

13.3 Function Group State

A Function Group State (meta-class ModeDeclaration) defines which Processes of a Function Group with what configuration parameters shall be running or not. The Machine State (that refers to the Processes of the AUTOSAR Adaptive Platform itself) defines at least the following Function Group States: Off, Startup, Shutdown, and Restart.

13.4 Software Cluster

A Software Cluster configuration refers to a set of modeled Processes. Those Processes are (transitively) used by one or more Function Group(s). Hereby, a Function Group and its associated entities shall be part of only one Software Cluster. In other words, Function Groups that span several Software Clusters are invalid. A Software Cluster is packaged into one Software Package - the atomic installable/updateable unit managed by Update and Configuration Management. A Software Cluster may depend on other Software Clusters. Such dependencies are expressed by version constraints. Please note that a Software Cluster is only used to configure deployment aspects. A Software Cluster is not a runtime entity.

A Software Cluster may specify a diagnostic configuration (DiagnosticContributionSet). A DiagnosticContributionSet specifies a single physical diagnostic address (and any number of functional addresses). All Software Clusters referencing the same DiagnosticContributionSet belong to the same physical diagnostic address. The diagnostic configuration applies to Processes that are contained in a Software Cluster. That means, at runtime, any diagnostic event produced by those Processes will be associated with the diagnostic address. Please refer to Section 13.11 for further details on the diagnostic deployment.

An exemplary Software Cluster during application design is shown in Figure 13.2. The application Software Cluster(s) are modeled/configured in the same way as the platform Software Cluster by defining Function Groups with Function Group States and associating StartupConfigurations of Processes to them.

A Software Cluster serves as a grouping entity during application design. As a result, entities within a Software Cluster, in particular the Function Groups, do not need to have a unique (simple) name within the overall model because their path is still unique. This allows to design Software Clusters independently, for example, by external suppliers.

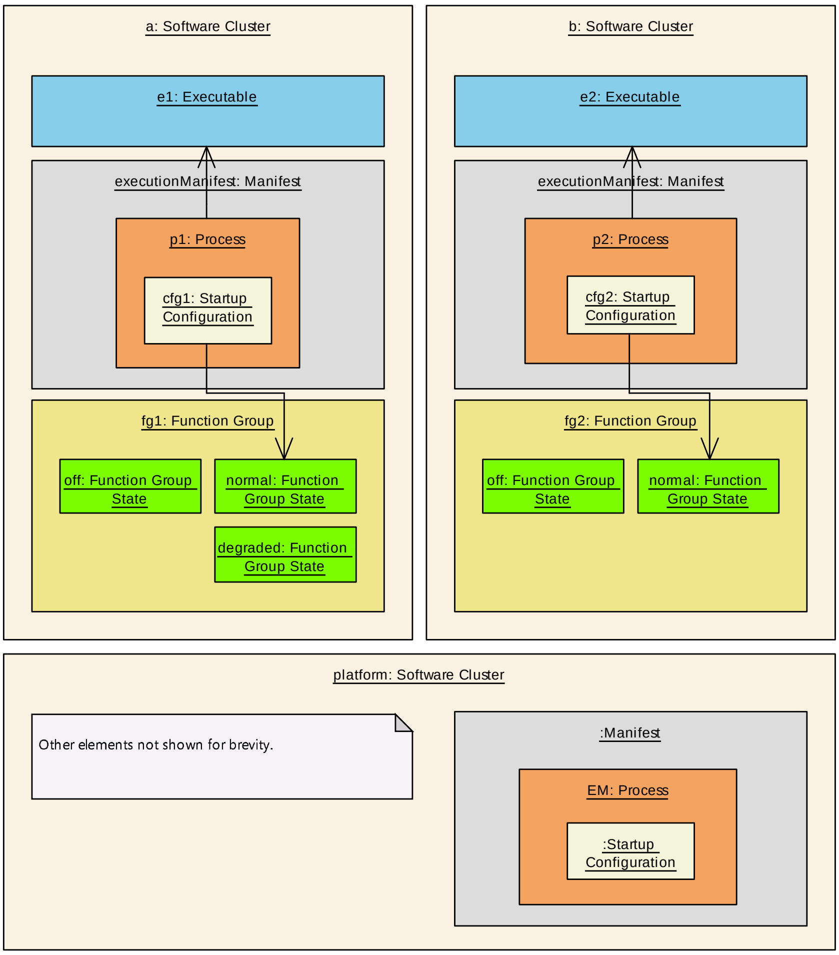

Figure 13.2: Exemplary Software Cluster during application design

From such a standardized model, an equivalent implementation-specific configuration for Execution Management is derived that is used during runtime (see Figure 13.3). That configuration advises Execution Management to start and configure processes accordingly when a Function Group State is requested. Hereby, Execution Management (logically) merges configurations contributed by all installed Software Packages. Other Functional Clusters that depend on configuration provided in the Manifests merge the configurations contributed by all installed Software Packages in the same way. Please also note that there is no corresponding runtime entity for a Software Cluster (see Figure 13.3).

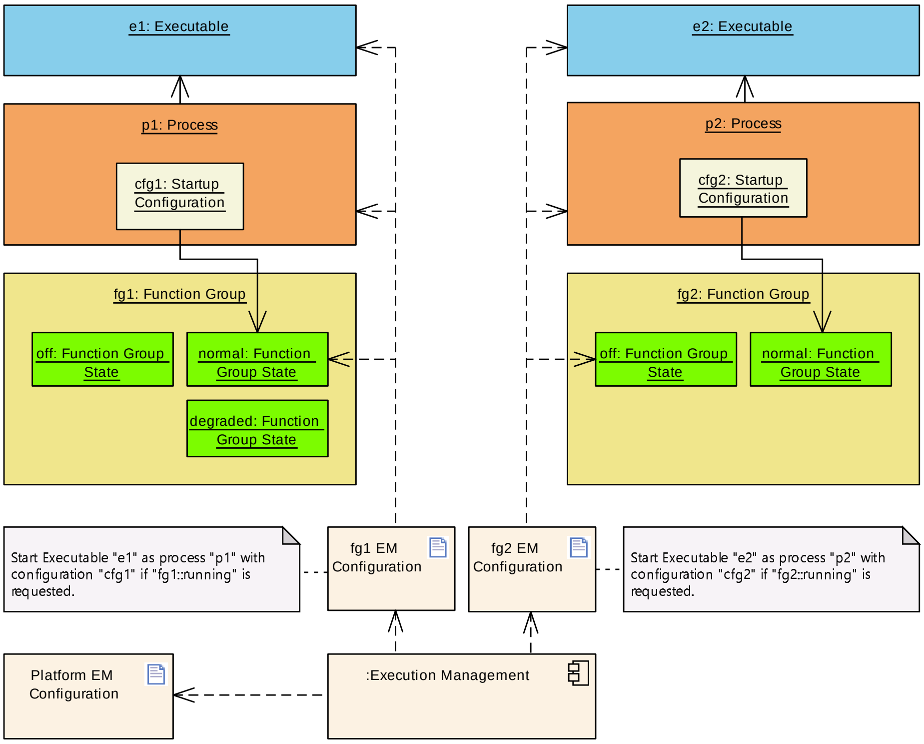

Figure 13.3: Impact of exemplary Software Cluster during runtime

All Processes related to the Functional Clusters of the AUTOSAR Adaptive Platform should be referenced only in Software Clusters of category PLATFORM_CORE or PLATFORM. This allows for platform-independent development of Software Clusters of category APPLICATION_LAYER.

In case a Functional Cluster may need multiple logical instances (for example, Diagnostic Management has a logical instance per diagnostic address), an implementation of the Functional Cluster should still use a single physical (daemon) process.

An AUTOSAR Adaptive Platform vendor may deviate from this design guide but should provide additional countermeasures to keep Adaptive Applications portable.

13.5 Machine

The AUTOSAR Adaptive Platform regards hardware it runs on as a Machine. The rationale behind that is to achieve a consistent platform view regardless of any virtualization technology which might be used. The Machine might be a real physical machine, a fully-virtualized machine, a para-virtualized OS, an OS-level-virtualized container or any other virtualized environment.

On hardware, there can be one or more Machine, and only a single instance of AUTOSAR Adaptive Platform runs on a machine. It is generally assumed that this hardware includes a single chip, hosting a single or multiple Machines. However, it is also possible that multiple chips form a single Machine if the AUTOSAR Adaptive Platform implementation allows it.

13.6 Manifest

A Manifest represents a piece of AUTOSAR model description that is created to support the configuration of an AUTOSAR Adaptive Platform product and which is uploaded to the AUTOSAR Adaptive Platform product, potentially in combination with other artifacts (like binary files) that contain executable code to which the Manifest applies. Please note that a typical Adaptive Application will make use of several distinct but interrelated Manifests. Hereby, the individual Manifests contribute information to the complete application model. For example, each Software Cluster may contribute a self-contained set of Manifests to configure its functionality.

The usage of a Manifest is limited to the AUTOSAR Adaptive Platform. This does not mean, however, that all ARXML produced in a development project that targets the AUTOSAR Adaptive Platform is automatically considered a Manifest. In fact, the AUTOSAR Adaptive Platform is usually not exclusively used in a vehicle project. A typical vehicle will most likely be also equipped with a number of ECUs developed on the AUTOSAR Classic Platform and the system design for the entire vehicle will, therefore, have to cover both, ECUs built on top of the AUTOSAR Classic Platform and Machines created on top of the AUTOSAR Adaptive Platform.

In principle, the term Manifest could be defined such that there is conceptually just one "Manifest" and every deployment aspect would be handled in this context. This does not seem appropriate because it became apparent that Manifest-related model-elements exist that are relevant in entirely different phases of a typical development project.

This aspect is taken as the main motivation that next to the application design it is necessary to subdivide the definition of the term Manifest in three different partitions:

Application Design This kind of description specifies all design-related aspects that apply to the creation of application software for the AUTOSAR Adaptive Platform. It is not necessarily required to be deployed to the adaptive platform machine, but the application design aids the definition of the deployment of application software in the

Execution Manifest and Service Instance Manifest. See Section 13.7 for details.

Execution Manifest This kind of Manifest is used to specify the deployment-related information of applications running on the AUTOSAR Adaptive Platform. An Execution Manifest is bundled with the actual executable code to support the integration of the executable code onto the machine. See Section 13.8 for details.

Service Instance Manifest This kind of Manifest is used to specify how service-oriented communication is configured in terms of the requirements of the underlying transport protocols. A Service Instance Manifest is bundled with the actual executable code that implements the respective usage of service-oriented communication. See Section 13.9 for details.

Machine Manifest This kind of Manifest is supposed to describe deployment-related content that applies to the configuration of just the underlying machine (i.e. without any applications running on the machine) that runs an AUTOSAR Adaptive Platform. A Machine Manifest is bundled with the software taken to establish an instance of the AUTOSAR Adaptive Platform. See Section 13.10 for details.

The temporal division between the definition (and usage) of different kinds of Manifest leads to the conclusion that in most cases different physical files will be used to store the content of the three kinds of Manifest. In addition to the Application Design and the different kinds of Manifest, the AUTOSAR Methodology supports a System Design with the possibility to describe Software Components of both AUTOSAR Platforms that will be used in a system in one single model. The Software Components of the different AUTOSAR platforms may communicate in a service-oriented way with each other. But it is also possible to describe a mapping of signals to services to create a bridge between the service-oriented communication and the signal-based communication.

13.7 Application Design

The application design describes all design-related modeling that applies to the creation of application software for the AUTOSAR AP. Application Design focuses on the following aspects:

- Data types used to classify information for the software design and implementation
- Service interfaces as the pivotal element for service-oriented communication
- Definition how service-oriented communication is accessible by the application
- Persistency Interfaces as the pivotal element to access persistent data and files
- Definition how persistent storage is accessible by the application
- Definition how files are accessible by the application

- Definition how crypto software is accessible by the application
- Definition how the Platform Health Management is accessible by the application
- Definition how Time Bases are accessible by the application
- Serialization properties to define the characteristics of how data is serialized for the transport on the network
- Description of client and server capabilities
- Grouping of applications in order to ease the deployment of software.

The artifacts defined in the application design are independent of a specific deployment of the application software and thus ease the reuse of application implementations for different deployment scenarios.

13.8 Execution Manifest

The purpose of the execution manifest is to provide information that is needed for the actual deployment of an application onto the AUTOSAR AP. The general idea is to keep the application software code as independent as possible from the deployment scenario to increase the odds that the application software can be reused in different deployment scenarios. With the execution manifest the instantiation of applications is controlled, thus it is possible to

- instantiate the same application software several times on the same machine, or to
- deploy the application software to several machines and instantiate the application software per machine.

The Execution manifest focuses on the following aspects:

- Startup configuration to define how the application instance shall be started. The startup includes the definition of startup options and access roles. Each startup may be dependent on machines states and/or function group states.
- Resource Management, in particular resource group assignments.

13.9 Service Instance Manifest

The implementation of service-oriented communication on the network requires configuration which is specific to the used communication technology (e.g. SOME/IP). Since the communication infrastructure shall behave the same on the provider and the requesters of a service, the implementation of the service shall be compatible on both sides.

The Service Instance Manifest focuses on the following aspects:

- Service interface deployment to define how a service shall be represented on the specific communication technology.
- Service instance deployment to define for specific provided and required service instances the required credentials for the communication technology.
- The configuration of E2E protection
- The configuration of Security protection
- The configuration of Log and Trace

13.10 Machine Manifest

The machine manifest allows to configure the actual adaptive platform instance running on specific hardware (machine).

The Machine Manifest focuses on the following aspects:

- Configuration of the network connection and defining the basic credentials for the network technology (e.g. for Ethernet this involves setting of a static IP address or the definition of DHCP).
- Configuration of the service discovery technology (e.g. for SOME/IP this involves the definition of the IP port and IP multi-cast address to be used).
- Definition of the used machine states.
- Definition of the used function groups.
- Configuration of the adaptive platform functional cluster implementations (e.g. the operating system provides a list of OS users with specific rights).
- The configuration of the Crypto platform Module.
- The configuration of Platform Health Management.
- The configuration of Time Synchronization.
- Documentation of available hardware resources (e.g. how much RAM is available; how many processor cores are available).

13.11 Diagnostics deployment

Each `Software Cluster` may reference a diagnostic configuration (`DiagnosticContributionSet`). Nevertheless, multiple `DiagnosticContributionSet`s can be utilized in a `Machine`. Each `DiagnosticContributionSet` may be used in several `Software Cluster`s. This concept provides a lot of flexibility in the assignment of diagnostic addresses to `Software Cluster`s. For example, in one extreme this allows to use a single diagnostic address for the whole `Machine` (see Figure 13.4), in another extreme a single diagnostic address per `Software Cluster` could be used (see Figure 13.5). Deployment scenarios in between those extremes are also possible.

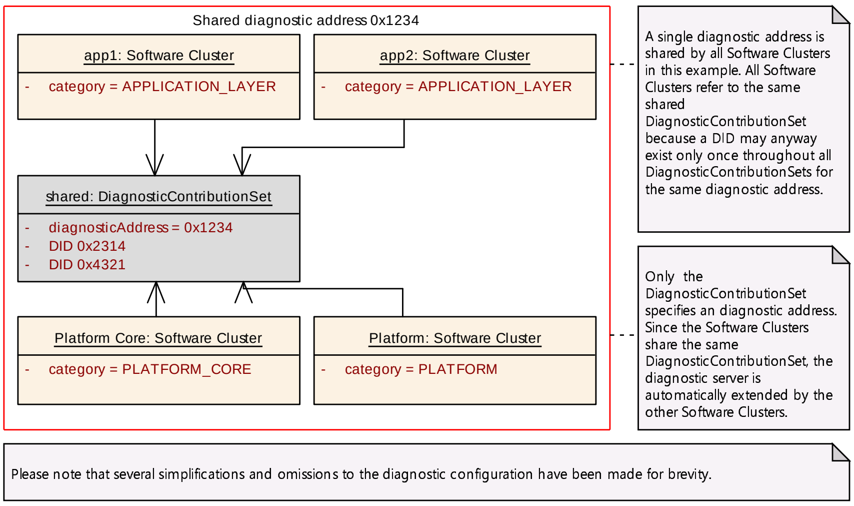

Figure 13.4: Example defining a single diagnostic address for the whole `Machine`

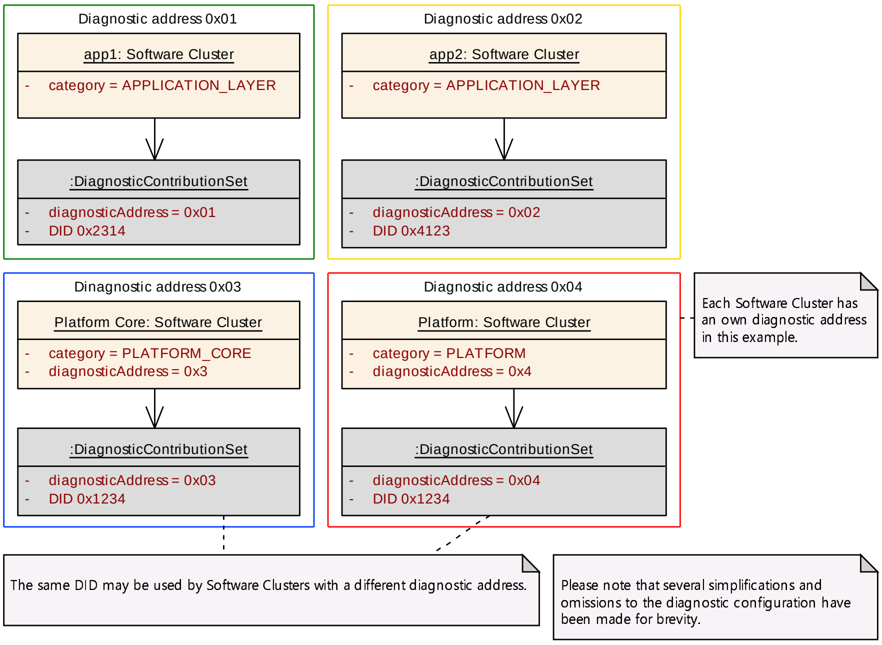

Figure 13.5: Example using one diagnostic address for each `Software Cluster`

In the case of a distributed `DiagnosticContributionSet`, each `Software Cluster` shall include its related diagnostic configuration objects (for example, the `Data Identifier` configuration). The merge of such a split `DiagnosticContributionSet` is done internally by the AUTOSAR Adaptive Platform (e.g. during installation or during start up the `Diagnostic Management` daemon).

13.12 Error Handling

Proper handling of errors during runtime is an important aspect to build safe and secure systems. The AUTOSAR Adaptive Platform does provide means for raising and handling of such errors on different levels in the platform.

Platform Health Management detects errors (errors in the logical control flow, missed deadlines, and missed liveness reporting) at the level of Processes and performs recovery actions (for example, degradation) according to rules defined in the Manifest. Execution Management detects unexpected termination of Processes and reports to State Management for handling of such errors.

During execution of a Process of an Adaptive Application, different abnormal conditions might be detected and need to be handled and/or reported. The following types of unsuccessful operations are distinguished within the AUTOSAR Adaptive Platform:

- An Error is the inability of an AUTOSAR Runtime for Adaptive Applications API function to fulfill its specified purpose. An Error it is often a consequence of invalid and/or unexpected input data. An Error is considered to be recoverable and therefore shall be handled by applications.
- A Violation is the consequence of failed pre- or post-conditions of internal state of the AUTOSAR Runtime for Adaptive Applications. A Violation is considered to be non-recoverable.
- A Corruption is the consequence of the corruption of a system resource, e.g. stack or heap overflow, or a hardware memory flaw (for example, a detected bit flip). A Corruption is considered to be non-recoverable.
- A failed default allocation is the inability of the AUTOSAR Runtime for Adaptive Applications's default memory allocation mechanism to satisfy an allocation request (for example, there is not enough free memory available).

Violations and Corruptions are non-recoverable (by the application itself) and will lead to an immediate termination of the application process. It is expected that a Violation or Corruption will very rarely be experienced by an application in a production environment. A Corruption is usually caused by faulty hardware. A Violation is usually caused by misuse of the ARA APIs or inconsistencies between the code and the Manifest (e.g., InstanceSpecifier referring to an element in the Manifest that does not exist) which should have been detected by quality assurance mechanisms upfront.

13.13 Trusted Platform

To guarantee the correct function of the system, it is crucial to ensure that the code executed on the AUTOSAR Adaptive Platform is unaltered (integrity) and has legitimate origin (authenticity). Keeping this property allows the integrator to build a Trusted

Platform. A key property of a system that implements a Trusted Platform is a Trust Anchor (also called Root of Trust). A Trust Anchor is often realized as a public key that is stored in a secure environment, e.g. in non-modifiable persistent memory or in an Hardware Security Module. A system designer is responsible to ensure that the system starts beginning with a Trust Anchor and that the chain of trust is kept until the Execution Management is launched. Depending on the mechanism that is chosen by the system designer to establish the chain of trust, the integrity and authenticity of the entire system (including all executables) may be checked during system start-up. Alternatively, the system designer might only ensure that the already executed software has been checked regarding integrity and authenticity and Execution Management takes over responsibility on continuing the chain of trust when it takes over control of the system. In the latter case, the system integrator is responsible to ensure that the Execution Management is configured accordingly.

Passing trust requires that a trusted entity checks (using trusted functionality) that the entity down the chain is authentic. The Trust Anchor (the first entity in the chain) is authentic by definition. An example of such a chain of trust could look like this: The Trust Anchor authenticates the bootloader before the bootloader is being started. In each subsequent step in the boot process, the to-be-started executable shall be authenticated first, for example by the executable started previously or by some external entity like an Hardware Security Module. After the relevant parts of the Operating System have been authentically started, it shall launch Execution Management as one of its first processes in the same manner passing trust to the AUTOSAR Adaptive Platform. Then, Execution Management takes over the responsibility of authenticating Adaptive Applications before launching them.

As stated above, if authenticity is not checked by the functionality of the Trust Anchor itself, which is authentic by definition, the functionality that shall be applied to verify authenticity of an executable has to be authenticated as well before it is applied. For instance, if the Crypto Functional Cluster shall be used to verify authenticity of executables, the Crypto Functional Cluster itself shall be authenticated by some trusted entity before it is used.

13.14 Communication Mechanisms

As communication is a fundamental principle of AP, this subsection provides an overview of the communication protocols and APIs in use and the intended purpose. Structuring of the protocol overview has been organized according to ISO OSI model into lower layer protocols (up to data link layer), network and transport layer protocols and application layer protocols (session, presentation and application layer). The communication APIs include the communication interfaces offered to the Adaptive Applications, i.e. communication related interfaces of the ara framework (see [10, platform design]). There is no implication that protocol deployment is mandatory for a proper setup of an Adaptive Machine. For some protocols this may depend on the project specific requirements.

For a better understanding of the AUTOSAR context following categories of protocol usage are introduced and included in the overview:

- native: Protocol (fields) has / have been specified especially for AUTOSAR usage. There is a protocol specification available.
- adapted: AP includes features of a protocol (suite) defined by 3rd party and may modify some of them and / or extend them with AUTOSAR specific extensions as part of the standard. The extension is part of a protocol specification.
- supported: AP includes features of a protocol (suite) defined by 3rd party in native form in the standard. The protocol is referred to within requirement specification items and optionally SW specification item(s) and / or the Manifest to define protocol specific properties.
- external standard: In contrast to 3rd party protocol (suite) mentioned in category 'supported', protocols within this category are not part of any AUTOSAR specification items (but may be mentioned in technical reports, explanatory documents or added as notes and alike). Their usage is not binding from AUTOSAR perspective, but some potential opportunities may be recognized in combination with functionality covered by the AUTOSAR standard. Specific usage examples are provided where suitable. Potentially listing external standard protocols could be a non-exhaustive task. Therefore, inclusion in the overview should be justified due to high potential in Automotive context and consequently for the ecosystem of Adaptive Platform. A rationale should provide further explanation.

The protocols listed in this subsection in general don't consider support for external vehicle connectivity (lower layers up to ISO OSI transport layer) due to completely different ecosystems, such as the following examples:

- Wireless connectivity for long and mid-range communication (e.g. C-V2X standard based on 3GPP Rel. 14 and beyond or IEEE 802.11p)
- Wireless nearfield connectivity e.g. for the purpose of sensor and user device connectivity (e.g. IEEE 802.11 a/b/g, Bluetooth LE, ..)
- Connectivity via plug-in user devices (e.g. USB, AUX, AV, SD„ ..)

13.14.1 Lower Layer Protocols

Protocol	***AUTOSAR Context***	***Rationale***	***AUTOSAR Reference***
Ethernet	supported	Ethernet MAC layer is a prerequisite for service-oriented communication. Configuration parameters are defined for access of Ethernet type media in Automotive context.	EthernetCommunicationConnector in [9, Specification of Manifest]

▽

△

Protocol	*AUTOSAR Context*	*Rationale*	*AUTOSAR Reference*
Ethernet over CAN XL	supported	Support of CAN XL media is included in the standard to offer tunneling of Ethernet frames. For this case configuration parameters are defined to access CAN XL controller.	"CanXlProps" in [9, Specification of Manifest]
MACsec	supported	MACsec provides cryptographic protection for MAC frames according to IEEE 802.1AE. AUTOSAR standard defines creation and maintenance of secure channel and association (such as the MACsec key agreement).	[11, Explanation of MACsec and MKA Protocols implementation and configuration guidelines]
TimeSync Protocol	adopted	The time synchronization protocol (derived from IEEE 802.1 AS) is used to synchronize time bases and the corresponding Ethernet messages and supports measurement of time differences between Ethernet frames. It is supposed to be applicable for time-critical and safety-related automotive applications. Support for VLAN tagging is based on IEEE 802.1Q.	[12, Time Synchronization over Ethernet Protocol Specification]

Table 13.1: Lower layer protocols relevant to AP

13.14.2 Network and Transport Layer Protocols

Protocol	*AUTOSAR Context*	*Rationale*	*AUTOSAR Reference*
IPv4 & Companion Protocols (ARP, ICMPv4)	supported	Network protocol to exchange IP datagrams if IPv4 endpoint option (addressing scheme) is configured. Even though not mentioned by AUTOSAR specification ARP (address resolution protocol) is implicitly included for an IPv4 endpoint. The protocol is used to provide the association between the IPv4 network address and the MAC address. This mechanism is needed to determine the correct MAC address for IP datagram delivery. Due to the different format of the ▽	"Ipv4Props" in [9, Specification of Manifest]

▽

△

Protocol	*AUTOSAR Context*	*Rationale*	*AUTOSAR Reference*
		△ addressing schemes static mapping without resolution is usually not feasible. In addition, ICMPv4 should be considered in a network with IPv4 endpoints. This protocol is used to exchange IPv4 network related status and error messages between these endpoints.	
IPv6 & Companion Protocols (NDP, ICMPv6)	supported	Network protocol to exchange IP datagrams if IPv6 endpoint option (addressing scheme) is configured. Even though not mentioned by AUTOSAR specification NDP (Network Discovery Protocol) is implicitly included for IPv6 endpoints. NDP replaces ARP in an IPv6 network. It is responsible for discovery and IPv6 address configuration tasks. In addition, ICMPv6 (Internet Control Message Protocol v6) should be considered in a network with IPv6 endpoints. This protocol is used to exchange IPv6 network related status and error messages between these endpoints.	"Ipv6Props" in [9, Specification of Manifest]
TCP	supported	Transmission Control Protocol is used as a reliable transport protocol to exchange service-oriented communication messages between two endpoints. For this purpose, a TCP port mapping can be configured.	"TcpProps" in [9, Specification of Manifest]
UDP	supported	User Datagram Protocol is used as a time sensitive transport protocol to exchange service-oriented communication messages between two endpoints. For this purpose, a UDP port mapping can be configured.	"UdpProps" in [9, Specification of Manifest]

▽

△

Protocol	*AUTOSAR Context*	*Rationale*	*AUTOSAR Reference*
IPSec	supported	IPsec provides cryptographic protection for IP datagrams in IPv4 and IPv6 networks. IP related information specified for IPsec configuration can be used for authenticating the remote endpoint. Implicitly IKEv2 (Internet Key Exchange) protocol v2 is used to set up the security association between the IP endpoints by means of key generation and exchange.	[8, Specification of Communication Management] [9, Specification of Manifest]AP-TPS-ManifestSpecification

Table 13.2: Network and transport layer protocols relevant to AP

13.14.3 Application Layer Protocols

Protocol	*AUTOSAR Context*	*Rationale*	*AUTOSAR Reference*
SOME/IP	native	Scalable service-oriented middleware over IP provides a transport for service-oriented communication over an IP network. It is based on service definitions that list the functionality that the service provides. A service can consist of a combinations of zero up to multiple events, methods and fields.	[13, SOME/IP Protocol Specification]
SOME/IP SD	native	This protocol is the service discovery part for service-oriented communication over an IP network. The main task is communicating the availability of functional entities called services and controlling the send behavior of event messages. This allows sending only event messages to receivers requiring them (Publish/Subscribe).	[14, SOME/IP Service Discovery Protocol Specification]

▽

△

Protocol	AUTOSAR Context	Rationale	AUTOSAR Reference
S2S	native	The service-oriented communication can be exchanged between AP and CP instances. If information contained in signal-based messages needs to be shared as service or vice versa, a signal-to-service conversion is needed. The signal-based payload for such type of communication needs to be carried in a SOME/IP message.	[8, Specification of Communication Management subsection "Signal-Based Network binding"]
UDP NM	native	Main purpose of the network management protocol (based on UDP) is to coordinate one or more groups of machines to wake up and shutdown their communication stack synchronously. The NM algorithm is based on periodic NM messages, which are received by all nodes in an NM cluster. Reception of NM messages indicates that sending nodes want to keep NM cluster awake. If any node does not need communication anymore, it stops sending NM messages.	[15, Specification of Network Management Protocol]
DLT	native	The protocol allows sending diagnostic, log and trace information onto the communications network. The purpose is to collect debug information from applications or other SW components, add metadata to the debug information before transmitting it to the receiver. In addition, the DLT protocol allows filtering of debug information depending on the severity level.	[16, Log and Trace Protocol Specification]

▽

△

Protocol	AUTOSAR Context	Rationale	AUTOSAR Reference
SecOC	native	Authentication and integrity protection of sensitive data is necessary to protect correct and safe functionality of the vehicle systems - this ensures that received data comes from an authentic machine and has the correct value. The SecOC protocol described in this document verifies the authenticity and freshness of PDU-based communication between machines within the vehicle architecture. The approach requires both the sending machine and the receiving machine to implement a SecOC module.	[17, Specification of Secure Onboard Communication Protocol]
IDS	native	The main use case for the Intrusion Detection System Protocol is the propagation of qualified security events in a way that is independent from the machine or the used communication mechanism. IDS monitoring instances can be allocated to all nodes of the vehicle architecture that are security relevant. This decision is typically based on a security analysis of the vehicle E/E architecture.	[18, Specification of Intrusion Detection System Protocol]
E2E	native	For safe communication integrity of messages can be ensured by the end-to-end protection offered by E2E protocol. End-to-end protection assumes that safety-related data exchange shall be protected at runtime against the effects of faults within the communication link. End-to-end protection detects faults and reports a status in addition to the data to the application. The application is responsible for handling such faults.	[19, E2E Protocol Specification]

Table 13.3: Application layer protocols native to AP

Protocol	*AUTOSAR Context*	*Rationale*	*AUTOSAR Reference*
DHCPv4	external standard	The protocol is used to enable an IPv4 endpoint to obtain IPv4 network configuration (like IP address, subnet mask, etc.) dynamically from a serving instance in the vehicle network.	no reference
DHCPv6	supported	The protocol is used to enable an IPv6 endpoint to obtain IPv6 network configuration (like IP address, subnet mask, etc.) dynamically from a serving instance in the vehicle network.	"Dhcpv6Props" in [9, Specification of Manifest]
IPC	external standard	Interprocess communication may be beneficial for the exchange of data between two processes, which are located in the same machine. A significant advantage of IPC is the lightweight type of communication, i.e. less resources are requested. There are different mechanisms, such as shared memory, message queues, pipes or sockets.	[10, Explanation of Adaptive Platform Design]
DNS	supported	The domain name service can be used by an Adaptive Application, for example to resolve the host name of a communication peer to the corresponding IP address. If supported by an Adaptive Machine, a DNS server needs to be configured, which can be requested for the name resolving.	"dnsServerAddress" in [9, Specification of Manifest]
mDNS	external standard	Multicast DNS protocol is used for the same purpose like DNS, i.e. name resolving between communication peers. mDNS is a special variant where no server instance is needed for the resolution. This works especially in small networks with a limited number of peers. The name resolution request is sent as a multicast message addressed to all peers to provide the requested information. Within AUTOSAR relevance is considered in context of SOVD, as discovery of the SOVD gateway by the backend connectivity could be accomplished by using mDNS.	[20, Explanation of Service-Oriented Vehicle Diagnostics]

▽

△

Protocol	*AUTOSAR Context*	*Rationale*	*AUTOSAR Reference*
DDS	supported	DDS is a middleware protocol and API standard for data-centric connectivity standardized by the Object Management Group. It can be considered as an alternative to SOME/IP to implement service-oriented communication.	[21, Requirements on Data Distribution Service]
DoIP	supported	The Diagnostics over Internet Protocol (Communication protocol of automotive electronics according to ISO-13400-2[5]) is an option to transport diagnostic messages (UDS) over the network.	[22, Specification of Diagnostic Manager subsection "DoIP"]
TLS / DTLS	supported	A transport layer security (TLS) channel works on top of TCP protocol and may provide authenticity, integrity and confidentiality. It may be used in combination with SOME/IP or DDS network binding to provide the mentioned security services for these protocols. In contrast datagram transport layer security channel provides equivalent security services like TLS for applications based on UDP transport.	[8, Specification of Communication Management] [9, Specification of Manifest]AP-TPS-ManifestSpecification
OAuth 2.0	external standard	This stands for Open Authorization and is an IETF based standard to authorize access to web resources. In AUTOSAR it may be used for authorization access to SOVD (service-oriented vehicle diagnostics) resources.	[20, Explanation of Service-Oriented Vehicle Diagnostics]
HTTP / HTTPS	external standard	Hypertext transport protocol (secured) is an IETF standard to exchange information between a web server and client. For SOVD-based communication, the diagnostic manager requires one or more socket connections including port and IP address to be able to establish an HTTP resp. HTTPS connection with an external entity.	[22, Specification of Diagnostic Manager subsection "SOVD Transport Layer"]

▽

△

Protocol	AUTOSAR Context	Rationale	AUTOSAR Reference
VISS	external standard	The protocol may be used by an external entity to request and receive specific vehicle signal information from a vehicle network via Automotive API. At the gateway connecting the external and the vehicle internal network, this protocol needs to be mapped to an internal communication protocol supported by the Adaptive Machine(s) being responsible to satisfy the service request.	[23, Specification of Automotive API Gateway]

Table 13.4: Application layer protocols not native to AP

13.14.4 Communication APIs

In addition to communication protocols which may be executed on an Adaptive Machine, this subsection provides an overview about communication APIs which might be used by an Adaptive Application.

API	Rationale	AUTOSAR Reference
`ara::com`	This API provides calls for service-oriented communication according to the publish / subscribe pattern. Network bindings are provided for SOME/IP protocol, DDS protocol and signal-to service mapping for communication with endpoints, which use exchange of signals.	[8, Specification of Communication Management]
`ara::rds`	This API is considered for the exchange of raw data, between e.g. certain sensor devices and applications, which need to process such data.	[24, Specification of Raw Data Stream]
ara::adi::sensorif	Data communication between sensors and data fusion unit e.g. for automated driving functions, which might be on an AP instance. This is part of ISO 23150 standard.	[25, Specification of Sensor Interfaces]
SOVD as part of `ara::diag`	The service-oriented vehicle diagnostic API extends the possibilities for diagnostics communication according to the related ASAM standard. A client of this API may be triggered by a diagnostic request, which is external to the vehicle and could be re-directed to different AP and CP instances for satisfying the request.	[22, Specification of Diagnostic Manager] Chapter "SOVD *Transport Layer*"

▽

△

API	Rationale	AUTOSAR Reference
ara::aag	Automotive API that is realized by the Automotive API Gateway provides access for external endpoints to request services from Adaptive Applications. A mapping is requested at the border of the vehicle to map communication mechanisms between the external network and the AP communication means (ara::com)	[23, Specification of Automotive API Gateway]

Table 13.5: Communication APIs relevant to AP

13.15 Service-Oriented Communication

The interaction in Service Oriented Architecture is between service providers and service consumers in which one provides some data and functionality and the other consumes it. Before the service consumer can use the data the service provider offers, the two parties need to come to a contract agreement. The contract agreement in AUTOSAR is specified in the form of a Service Interface (meta-class ServiceInterface). The Service Interface defines in detail which methods, fields, and events the service provider offers. The service consumer can rely that these methods, fields, and events that are defined in the Service Interface will be supplied by the service provider. Service Interfaces are modeled in AUTOSAR using ARXML during application design (see [9] for details).

In the application design the services are provided to other components by application software components (meta-class AdaptiveApplicationSwComponentType). Ports (meta-class PortPrototype) are used to describe the interconnection between Software Components. Each Port of a Software Component is typed by an interface (e.g., a Service Interface). In other words each Port definition is of a specific type and can only connect Software Components defining Ports of the same type. Each Software Component is able to define multiple Ports. A Port that is typed by a Service Interface defines a Service Instance. A PPort (Provided Port, meta-class PPortPrototype) provides a Service Instance and an RPort (Required Port, meta-class RPortPrototype) consumes a Service Instance. This means that the same Service Interface may be provided by different PPorts (of the same or of different Software Components) as a different Service Instance. An RPort indicates that a Software Component needs to use a Service Interface and that it searches for a provided Service Instance (using Service Discovery or static configuration). Please note that there may be many consumers for the same provided Service Instance in a system.

The modeling of Software Components on application design level in the AUTOSAR model (ARXML) is standardized by AUTOSAR. On the other hand a C++ Software Component Framework is not standardized by AUTOSAR and the implementation of a modeled component can be freely chosen by the implementer.

During the life cycle of a Service Interface, the content of the service may change (because consumers have change requests or errors need to be fixed). AUTOSAR supports the versioning of a Service Interface to indicate whether the service was changed in a backward compatible way or in an incompatible way. If the minor version number of a Service Interface is increased it means that the Service Interface was changed in a compatible way (e.g. by adding new functionality to the service), so that existing consumers are able to use the Service Interface without any changes. If the major version of a Service Interface increases it means that the service was changed in a non backward compatible way so that existing consumers may need to be adapted to the updated Service Interface.

A Mentioned Class Tables

For the sake of completeness, this chapter contains a set of class tables representing meta-classes mentioned in the context of this document. Please refer to [9] for additional details.

Class	**AdaptiveApplicationSwComponentType**			
Note	This meta-class represents the ability to support the formal modeling of application software on the AUTOSAR adaptive platform. Consequently, it shall only be used on the AUTOSAR adaptive platform. **Tags:** atp.recommendedPackage=AdaptiveApplicationSwComponentTypes This Class is only used by the AUTOSAR Adaptive Platform.			
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *SwComponentType*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
internalBehavior	AdaptiveSwcInternalBehavior	0..1	aggr	This aggregation represents the internal behavior of the AdaptiveApplicationSwComponentType for the AUTOSAR adaptive platform. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=internalBehavior.shortName, internalBehavior.variationPoint.shortLabel vh.latestBindingTime=preCompileTime

Table A.1: AdaptiveApplicationSwComponentType

Class	**DiagnosticContributionSet**			
Note	This meta-class represents a root node of a diagnostic extract. It bundles a given set of diagnostic model elements. The granularity of the DiagonsticContributionSet is arbitrary in order to support the aspect of decentralized configuration, i.e. different contributors can come up with an own DiagnosticContributionSet. **Tags:** atp.recommendedPackage=DiagnosticContributionSets			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
commonProperties	DiagnosticCommonProps	0..1	aggr	This attribute represents a collection of diagnostic properties that are shared among the entire DiagnosticContributionSet. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=commonProperties
element	DiagnosticCommonElement	*	ref	This represents a DiagnosticCommonElement considered in the context of the DiagnosticContributionSet **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=element.diagnosticCommonElement, element.variationPoint.shortLabel vh.latestBindingTime=postBuild
serviceTable	DiagnosticServiceTable	*	ref	This represents the collection of DiagnosticServiceTables to be considered in the scope of this DiagnosticContributionSet. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=serviceTable.diagnosticServiceTable, serviceTable.variationPoint.shortLabel vh.latestBindingTime=postBuild

Table A.2: DiagnosticContributionSet

Class	**FirewallRule**			
Note	Firewall Rule that defines the control information in individual packets. **Tags:** atp.Status=candidate atp.recommendedPackage=FirewallRules			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *Referrable*, *UploadableDesignElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	*Type*	*Mult.*	*Kind*	*Note*
bucketSize	PositiveInteger	0..1	attr	This attribute defines the capacity of the queue for rate limitation (leaky-bucket Algorithm). **Tags:** atp.Status=candidate
dataLinkLayer Rule	DataLinkLayerRule	0..1	aggr	Configuration of rules on the Data Link Layer **Tags:** atp.Status=candidate
ddsRule	DdsRule	0..1	aggr	Configuration of firewall rules for DDS. **Tags:** atp.Status=candidate
doIpRule	DoIpRule	0..1	aggr	Configuration of firewall rules for DoIP messages **Tags:** atp.Status=candidate
networkLayer Rule	NetworkLayerRule	0..1	aggr	Configuration of rules on the Network Layer **Tags:** atp.Status=candidate
payloadByte PatternRule	PayloadBytePattern Rule	*	aggr	Configuration of generic firewall rules **Tags:** atp.Status=candidate
refillAmount	PositiveInteger	0..1	attr	This attribute defines the output rate that describes how many packets leave the queue per second (leaky-bucket Algorithm). **Tags:** atp.Status=candidate
someipRule	SomeipProtocolRule	0..1	aggr	Configuration of firewall rules for SOME/IP messages **Tags:** atp.Status=candidate
someipSdRule	SomeipSdRule	0..1	aggr	Configuration of firewall rules for SOME/IP Service Discovery messages **Tags:** atp.Status=candidate
transportLayer Rule	TransportLayerRule	0..1	aggr	Configuration of rules on the Transport Layer **Tags:** atp.Status=candidate

Table A.3: FirewallRule

Class	**ModeDeclaration**			
Note	Declaration of one Mode. The name and semantics of a specific mode is not defined in the meta-model.			
Base	*ARObject*, *AtpClassifier*, *AtpFeature*, *AtpStructureElement*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Aggregated by	*AtpClassifier*.atpFeature, ModeDeclarationGroup.modeDeclaration			
Attribute	*Type*	*Mult.*	*Kind*	*Note*
value	PositiveInteger	0..1	attr	The RTE shall take the value of this attribute for generating the source code representation of this Mode Declaration.

Table A.4: ModeDeclaration

Class	**ModeDeclarationGroup**
Note	A collection of Mode Declarations. Also, the initial mode is explicitly identified. **Tags:** atp.recommendedPackage=ModeDeclarationGroups
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *UploadableDesignElement*, *UploadablePackageElement*

▽

△

Class	**ModeDeclarationGroup**			
Aggregated by	ARPackage.element			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
initialMode	ModeDeclaration	0..1	ref	The initial mode of the ModeDeclarationGroup. This mode is active before any mode switches occurred.
mode Declaration	ModeDeclaration	*	aggr	The ModeDeclarations collected in this ModeDeclaration Group. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=modeDeclaration.shortName, mode Declaration.variationPoint.shortLabel vh.latestBindingTime=blueprintDerivationTime

Table A.5: ModeDeclarationGroup

Class	**PPortPrototype**			
Note	Component port providing a certain port interface.			
Base	*ARObject*, *AbstractProvidedPortPrototype*, *AtpBlueprintable*, *AtpFeature*, *AtpPrototype*, *Identifiable*, *MultilanguageReferrable*, *PortPrototype*, *Referrable*			
Aggregated by	*AtpClassifier*.atpFeature, *SwComponentType*.port			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
provided Interface	PortInterface	0..1	tref	The interface that this port provides. **Stereotypes:** isOfType

Table A.6: PPortPrototype

Class	**PersistencyDeploymentElementToCryptoKeySlotMapping**			
Note	This meta-class represents the ability to define a mapping between the PersistencyDeploymentElement and a CryptoKeySlot. **Tags:** atp.recommendedPackage=FCInteractions This Class is only used by the AUTOSAR Adaptive Platform.			
Base	*ARElement*, *ARObject*, *CollectableElement*, *FunctionalClusterInteractsWithFunctionalClusterMapping*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *UploadableDeployment Element*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
cryptoAlgorithm String	String	0..1	attr	This attribute defines the cryptographic algorithm used for hashing, encryption, decryption, signature/MAC verification, or MAC generation.
cryptoKeySlot	CryptoKeySlot	0..1	ref	This reference represents the mapped CryptoKeySlot.
keySlotUsage	CryptoKeySlotUsage Enum	0..1	attr	This attribute defines the role of the keySlot assignment.
persistency Deployment Element	PersistencyDeployment Element	0..1	ref	This reference represents the mapped Persistency Deployment.
verificationHash	String	0..1	attr	This attribute defines the hash of the storage used in case of verification.

Table A.7: PersistencyDeploymentElementToCryptoKeySlotMapping

Class	PersistencyDeploymentToCryptoKeySlotMapping			
Note	This meta-class represents the ability to define a mapping between the PersistencyDeployment and a CryptoKeySlot. **Tags:** atp.recommendedPackage=FCInteractions This Class is only used by the AUTOSAR Adaptive Platform.			
Base	*ARElement*, *ARObject*, *CollectableElement*, *FunctionalClusterInteractsWithFunctionalClusterMapping*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *UploadableDeploymentElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
cryptoAlgorithmString	String	0..1	attr	This attribute defines the cryptographic algorithm used for hashing, encryption, decryption, signature/MAC verification, or MAC generation.
cryptoKeySlot	CryptoKeySlot	0..1	ref	This reference represents the mapped CryptoKeySlot.
keySlotUsage	CryptoKeySlotUsageEnum	0..1	attr	This attribute defines the role of the keySlot assignment.
persistencyDeployment	PersistencyDeployment	0..1	ref	This reference represents the mapped PersistencyDeployment.
verificationHash	String	0..1	attr	This attribute defines the hash of the storage used in case of verification.

Table A.8: PersistencyDeploymentToCryptoKeySlotMapping

Class	*PersistencyRedundancyHandling* (abstract)			
Note	This abstract base class represents a formal description of redundancy. This Class is only used by the AUTOSAR Adaptive Platform.			
Base	*ARObject*			
Subclasses	*PersistencyRedundancyChecksum*, PersistencyRedundancyMOutOfN			
Aggregated by	*PersistencyDeployment*.redundancyHandling, *PersistencyInterface*.redundancyHandling			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
scope	PersistencyRedundancyHandlingScopeEnum	0..1	attr	This attribute controls the scope in which the redundancy handling is applied.

Table A.9: PersistencyRedundancyHandling

Class	*PortPrototype* (abstract)			
Note	Base class for the ports of an AUTOSAR software component. The aggregation of PortPrototypes is subject to variability with the purpose to support the conditional existence of ports.			
Base	*ARObject*, *AtpBlueprintable*, *AtpFeature*, *AtpPrototype*, *Identifiable*, *MultilanguageReferrable*, *Referrable*			
Subclasses	*AbstractProvidedPortPrototype*, *AbstractRequiredPortPrototype*			
Aggregated by	*AtpClassifier*.atpFeature, *SwComponentType*.port			
Attribute	**Type**	**Mult.**	**Kind**	**Note**
clientServerAnnotation	ClientServerAnnotation	*	aggr	Annotation of this `PortPrototype` with respect to client/server communication.
delegatedPortAnnotation	DelegatedPortAnnotation	0..1	aggr	Annotations on this delegated port.
ioHwAbstractionServerAnnotation	IoHwAbstractionServerAnnotation	*	aggr	Annotations on this IO Hardware Abstraction port.
modePortAnnotation	ModePortAnnotation	*	aggr	Annotations on this mode port.
nvDataPortAnnotation	NvDataPortAnnotation	*	aggr	Annotations on this non voilatile data port.

▽

△

Class	***PortPrototype*** (abstract)			
parameterPort Annotation	ParameterPort Annotation	*	aggr	Annotations on this parameter port.
portPrototype Props	PortPrototypeProps	0..1	aggr	This attribute allows for the definition of further qualification of the semantics of a PortPrototype. This Attribute is only used by the AUTOSAR Adaptive Platform.
senderReceiver Annotation	SenderReceiver Annotation	*	aggr	Collection of annotations of this ports sender/receiver communication. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=senderReceiverAnnotation
triggerPort Annotation	TriggerPortAnnotation	*	aggr	Annotations on this trigger port.

Table A.10: PortPrototype

Class	**RPortPrototype**			
Note	Component port requiring a certain port interface.			
Base	*ARObject*, *AbstractRequiredPortPrototype*, *AtpBlueprintable*, *AtpFeature*, *AtpPrototype*, *Identifiable*, *MultilanguageReferrable*, *PortPrototype*, *Referrable*			
Aggregated by	*AtpClassifier*.atpFeature, *SwComponentType*.port			
Attribute	*Type*	*Mult.*	*Kind*	*Note*
required Interface	PortInterface	0..1	tref	The interface that this port requires. **Stereotypes:** isOfType

Table A.11: RPortPrototype

Class	**ServiceInterface**			
Note	This represents the ability to define a PortInterface that consists of a heterogeneous collection of methods, events and fields. **Tags:** atp.recommendedPackage=ServiceInterfaces This Class is only used by the AUTOSAR Adaptive Platform.			
Base	*ARElement*, *ARObject*, *AtpBlueprint*, *AtpBlueprintable*, *AtpClassifier*, *AtpType*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *PortInterface*, *Referrable*			
Aggregated by	ARPackage.element			
Attribute	*Type*	*Mult.*	*Kind*	*Note*
event	VariableDataPrototype	*	aggr	This represents the collection of events defined in the context of a ServiceInterface. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=event.shortName, event.variationPoint.short Label vh.latestBindingTime=blueprintDerivationTime xml.sequenceOffset=30
field	Field	*	aggr	This represents the collection of fields defined in the context of a ServiceInterface. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=field.shortName, field.variationPoint.short Label vh.latestBindingTime=blueprintDerivationTime xml.sequenceOffset=40
majorVersion	PositiveInteger	0..1	attr	Major version of the service contract. **Tags:** xml.sequenceOffset=10

▽

△

Class	**ServiceInterface**			
method	ClientServerOperation	*	aggr	This represents the collection of methods defined in the context of a ServiceInterface. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=method.shortName, method.variationPoint.shortLabel vh.latestBindingTime=blueprintDerivationTime xml.sequenceOffset=50
minorVersion	PositiveInteger	0..1	attr	Minor version of the service contract. **Tags:** xml.sequenceOffset=20
trigger	Trigger	*	aggr	This represents the collection of triggers defined in the context of a ServiceInterface. **Stereotypes:** atpSplitable; atpVariation **Tags:** atp.Splitkey=trigger.shortName, trigger.variationPoint.shortLabel vh.latestBindingTime=blueprintDerivationTime xml.sequenceOffset=60

Table A.12: ServiceInterface

Class	**SoftwareCluster**			
Note	This meta-class represents the ability to define an uploadable software-package, i.e. the SoftwareCluster shall contain all software and configuration for a given purpose. **Tags:** atp.recommendedPackage=SoftwareClusters This Class is only used by the AUTOSAR Adaptive Platform.			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *PackageableElement*, *Referrable*, *UploadableDeploymentElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
artifactChecksum	ArtifactChecksum	*	aggr	This aggregation carries the checksums for artifacts contained in the enclosing SoftwareCluster. Please note that the value of these checksums is only applicable at the time of configuration. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=artifactChecksum.shortName
artifactLocator	ArtifactLocator	*	aggr	This aggregation represents the artifact locations that are relevant in the context of the enclosing SoftwareCluster
claimedFunctionGroup	ModeDeclarationGroupPrototype	*	ref	Each SoftwareCluster can reserve the usage of a given functionGroup such that no other SoftwareCluster is allowed to use it
containedARElement	ARElement	*	ref	This reference represents the collection of model elements that cannot derive from UploadablePackageElement and that contribute to the completeness of the definition of the SoftwareCluster. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=containedARElement
containedPackageElement	UploadablePackageElement	*	ref	This reference identifies model elements that are required to complete the manifest content. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=containedPackageElement
containedProcess	Process	*	ref	This reference represent the processes contained in the enclosing SoftwareCluster.

▽

△

Class	SoftwareCluster			
design	SoftwareClusterDesign	*	ref	This reference represents the identification of all Software ClusterDesigns applicable for the enclosing Software Cluster. **Stereotypes:** atpUriDef
diagnostic Deployment Props	SoftwareCluster DiagnosticDeployment Props	0..1	ref	This reference identifies the applicable SoftwareCluster DiagnosticDeploymentProps that are applicable for the referencing SoftwareCluster.
license	Documentation	*	ref	This attribute allows for the inclusion of the full text of a license of the enclosing SoftwareCluster. In many cases open source licenses require the inclusion of the full license text to any software that is released under the respective license.
module Instantiation	AdaptiveModule Instantiation	*	ref	This reference identifies AdaptiveModuleInstantiations that need to be included with the SoftwareCluster in order to establish infrastructure required for the installation of the SoftwareCluster. **Stereotypes:** atpSplitable **Tags:** atp.Splitkey=moduleInstantiation
releaseNotes	Documentation	0..1	ref	This attribute allows for the explanations of changes since the previous version. The list of changes might require the creation of multiple paragraphs of test.
typeApproval	String	0..1	attr	This attribute carries the homologation information that may be specific for a given country.
vendorId	PositiveInteger	0..1	attr	Vendor ID of this Implementation according to the AUTOSAR vendor list.
vendor Signature	CryptoService Certificate	0..1	ref	This reference identifies the certificate that represents the vendor's signature.
version	StrongRevisionLabel String	0..1	attr	This attribute can be used to describe a version information for the enclosing SoftwareCluster.

Table A.13: SoftwareCluster

Class	**StartupConfig**			
Note	This meta-class represents a reusable startup configuration for processes.. **Tags:** atp.recommendedPackage=StartupConfigs This Class is only used by the AUTOSAR Adaptive Platform.			
Base	*ARElement*, *ARObject*, *CollectableElement*, *Identifiable*, *MultilanguageReferrable*, *Packageable Element*, *Referrable*, *UploadableDeploymentElement*, *UploadablePackageElement*			
Aggregated by	ARPackage.element			
Attribute	***Type***	***Mult.***	***Kind***	***Note***
environment Variable	TagWithOptionalValue	*	aggr	This aggregation represents the collection of environment variables that shall be added to the respective Process's environment prior to launch.
permissionTo CreateChild Process	Boolean	0..1	attr	This attribute defines if Process is permitted to create child Processes. When setting this parameter to true two things should be kept in mind: 1) safety and security implication of this configuration, 2) the fact that Process will assume management responsibilities for child Processes (i.e. it will be responsible for terminating Processes that it creates).
process Argument (ordered)	ProcessArgument	*	aggr	This aggregation represents the collection of command-line arguments applicable to the enclosing StartupConfig.

▽

△

Class	**StartupConfig**			
process ExecutionError	ApApplicationError	0..1	ref	this reference is used to identify the applicable execution error. **Tags:** atp.Status=draft
scheduling Policy	String	0..1	attr	This attribute represents the ability to define the scheduling policy for the initial thread of the application.
scheduling Priority	Integer	0..1	attr	This is the scheduling priority requested by the application itself.
termination Behavior	TerminationBehavior Enum	0..1	attr	This attribute defines the termination behavior of the Process.
timeout	EnterExitTimeout	0..1	aggr	This aggregation can be used to specify the timeouts for launching and terminating the process depending on the StartupConfig.

Table A.14: StartupConfig

Class	**StateDependentStartupConfig**			
Note	This meta-class defines the startup configuration for the process depending on a collection of machine states. This Class is only used by the AUTOSAR Adaptive Platform.			
Base	*ARObject*			
Aggregated by	Process.stateDependentStartupConfig			
Attribute	*Type*	*Mult.*	*Kind*	*Note*
execution Dependency	ExecutionDependency	*	aggr	This attribute defines that all processes that are referenced via the ExecutionDependency shall be launched and shall reach a certain ProcessState before the referencing process is started.
functionGroup State	ModeDeclaration	*	iref	This represent the applicable functionGroupMode. **InstanceRef implemented by:** FunctionGroupStateIn FunctionGroupSetInstanceRef
resource Consumption	ResourceConsumption	0..1	aggr	This aggregation provides the ability to define resource consumption boundaries on a per-process-startup-config basis.
resourceGroup	ResourceGroup	0..1	ref	Reference to an applicable resource group.
startupConfig	StartupConfig	0..1	ref	Reference to a reusable startup configuration with startup parameters.

Table A.15: StateDependentStartupConfig