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	R22-11

Document Change History

Date	Release	Changed by	Description
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

Disclaimer

This work (specification and/or software implementation) and the material contained in it, as released by AUTOSAR, is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the work.

The material contained in this work is protected by copyright and other types of intellectual property rights. The commercial exploitation of the material contained in this work requires a license to such intellectual property rights.

This work may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only. For any other purpose, no part of the work may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The work has been developed for automotive applications only. It has neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

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. Section 14 lists and rates risks associated with the architecture of the AUTOSAR Adaptive Platform and technical debt.

2 Definition of Terms and Acronyms

2.1 Acronyms and Abbreviations

Abbreviation / Acronym	Description
DoIP	Diagnostics over Internet Protocol
POSIX	Portable Operating System Interface
SecOC	AUTOSAR Secure Onboard Communication
TLS	Transport Layer Security
UML	Unified Modeling Language

2.2 Definition of Terms

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

Term	Description
Functional Cluster	A logical group of functionality within the AUTOSAR Adaptive Platform. `Functional Cluster`s are the second level of abstraction in the building block view (cf. Chapter 9). They are also subject of the individual specification documents that make up the AUTOSAR Adaptive Platform standard.
Function Group	A set of modeled `Process`es. See Section 13.2 for details.
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.

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_Main] and [5, 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.

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_Main] and [5, 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) [6] 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.

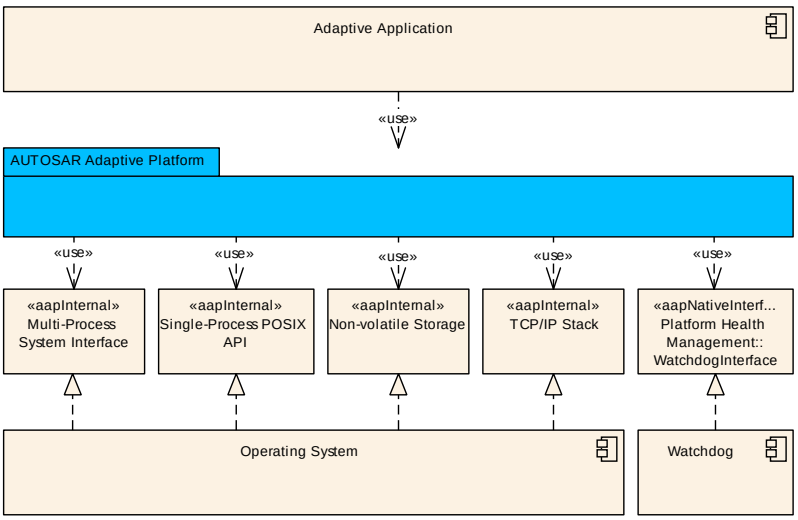

Figure 7.1: Overview of AUTOSAR Adaptive Platform and its context

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

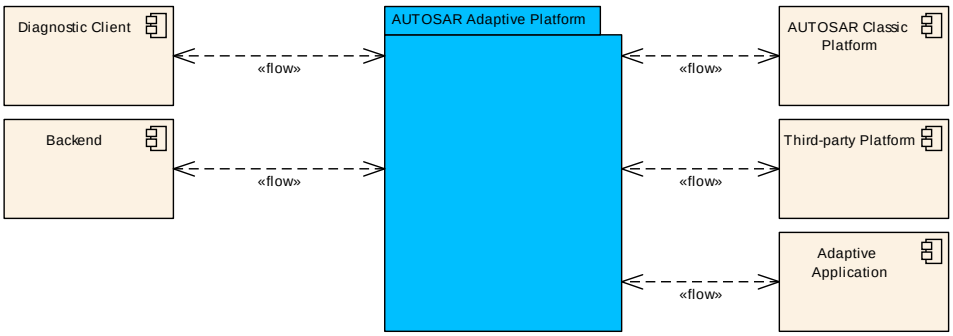

Figure 7.2: Overview of the AUTOSAR Adaptive Platform and external systems

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 AUTOSAR Adaptive Application

There may be many Adaptive Applications deployed in a vehicle on different Machines. An Adaptive Application that does not run on the current instance of the AUTOSAR Adaptive Platform is therefore an external system to the AUTOSAR Adaptive Platform. Such Adaptive Applications may exchange data such as sensor or status information by means of ara::com service interfaces.

7.3.2 AUTOSAR Classic Platform

The AUTOSAR Classic Platform is the main platform for deeply-embedded applications such as sensor/actor systems. Adaptive Applications may require access for example to sensor data provided by an AUTOSAR Classic Platform ECU and vice versa.

7.3.3 Third-party Platform

Besides the both platforms (AUTOSAR Adaptive Platform and AUTOSAR Classic Platform) provided by AUTOSAR, there may be ECUs in a vehicle and other systems that are built on different platforms that need to communicate with an Adaptive Application via AUTOSAR Adaptive Platform.

7.3.4 Diagnostic Client

A Diagnostic Client uses the diagnostic services provided by the AUTOSAR Adaptive Platform.

7.3.5 Backend

A Backend system provides Software Packages for download and controls the update process via Update and Configuration Management.

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 Clusters. The Functional Clusters of the AUTOSAR Adaptive Platform are defined to group a specific coherent technical functionality. Functional Clusters 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 Clusters interdependencies based on interfaces from other Functional Clusters they use. However, note that these interdependencies are recommendations rather than strict specifications because they would constrain implementations.

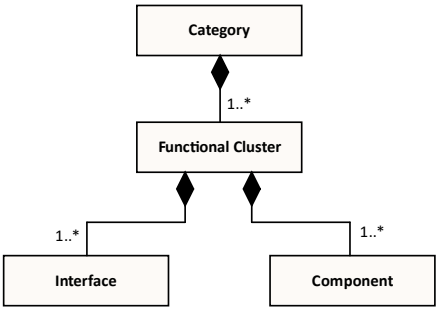

Figure 8.1: Type model of building blocks

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	Metaclass	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.
`aapNativeInterface`	Interface	An interface defined in the respective programming language of the stack implementation.

Stereotype	Metaclass	Semantics
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.
aapCallbackMethod	Operation	A method that acts as a callback.
aapServiceField	Attribute	A field specified as part of an ara::com service interface.
aapServiceEvent	Attribute	An event specified as part of an ara::com service interface.
aapProvidedPort	Usage, Realization	Denotes that the underlying relationship is configured in the Manifest using a provided port.
aapRequiredPort	Usage, Realization	Denotes that the underlying relationship is configured in the Manifest using a required port.

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 Applications) 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. 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 [7] 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`) [7] 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) [7], [8] 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) [7], [8] 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) [7], [8] 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.

8.6 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 Cluster`s in the AUTOSAR Adaptive Platform is currently not standardized. Some aspects, for example, access management, are also not yet fully incorporated and standardized in all `Functional Cluster`s.

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.

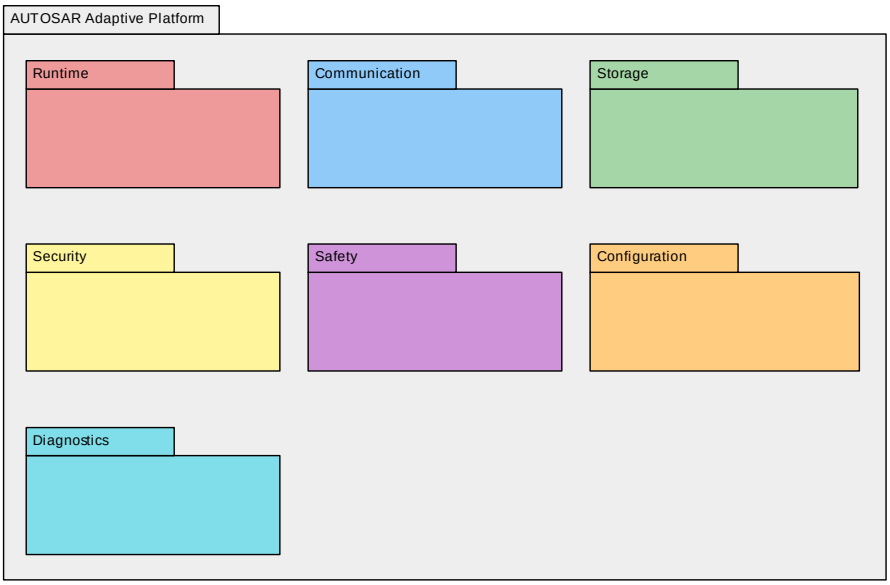

Figure 9.1: Overview of the AUTOSAR Adaptive Platform and its building block categories

9.1.1 Description pattern

The description of all building blocks (`FunctionalCluster`s) in this section uses the same pattern. Each `FunctionalCluster` is described in a separate section of the document. Such a section starts with an overview of the responsibilities of the `FunctionalCluster` followed by a sub-section called "Defined interfaces". The sub-section "Defined interfaces" lists all architectural interfaces specified in the namespace of the `FunctionalCluster`. Each interface is detailed in a separate table.

The subsection "Provided interfaces" then lists all interfaces provided by the FunctionalCluster (i.e. it is fully implemented 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.

9.2 Runtime

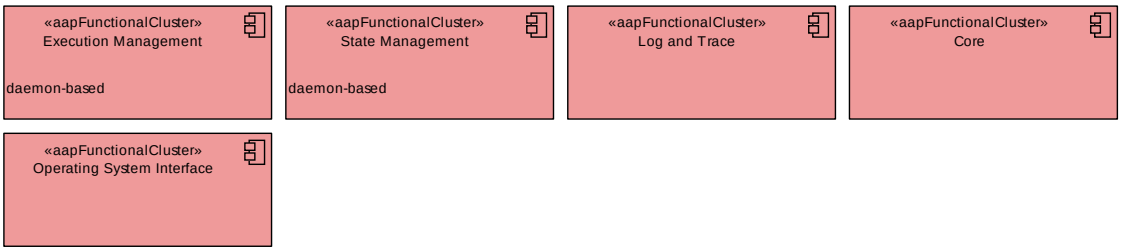

Figure 9.2: Functional Clusters in category Runtime

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), interfaces for the deterministic execution (see Section 9.2.1.1.2), and interfaces for `State Management` (see Section 9.2.1.1.3).

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).

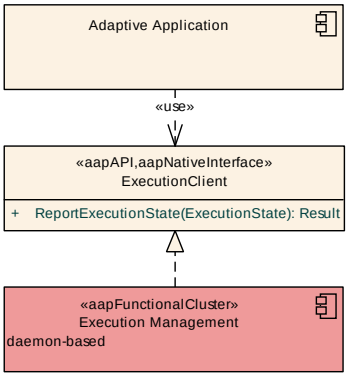

Figure 9.3: Interfaces for state reporting

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:	ReportExecutionState	Report the internal state of a `Process` to `Execution Management`.

9.2.1.1.2 Interfaces for deterministic execution

The DeterministicClient API (cf. Figure 9.4) provides operations to perform deterministic execution of tasks.

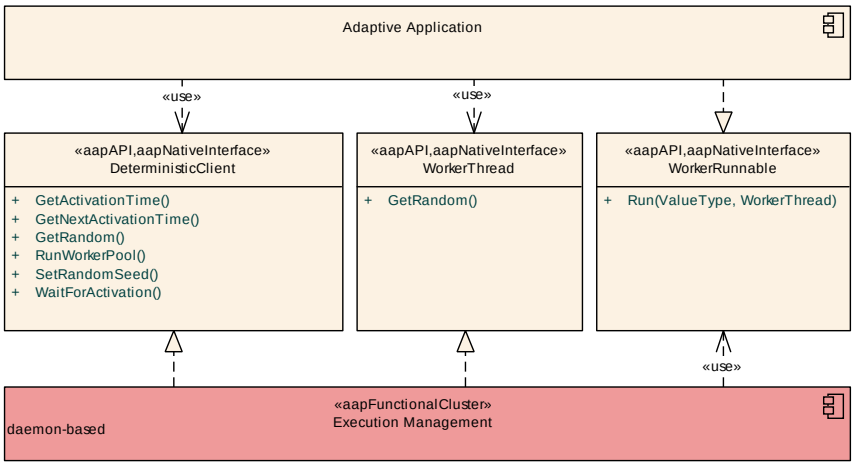

Figure 9.4: Interfaces for deterministic execution

Name:	DeterministicClient	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface provides the functionality for an application to run a cyclic deterministic execution. Each modelled `Process` which needs support for cyclic deterministic execution has to instantiate this interface.	
Operations:	GetActivationTime	Get the timestamp of the activation point.
	GetNextActivationTime	Get the timestamp of the next activation point.
	GetRandom	Returns a deterministic sequence of random numbers.
	RunWorkerPool	Runs tasks in a deterministic worker pool.
	SetRandomSeed	Seed the random number generator used for redundantly executed deterministic clients.
	WaitForActivation	Blocks and returns with a process control value when the next activation is triggered by the runtime.

Name:	WorkerRunnable	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface is used to implement worker runnables for the DeterministicClient.	
Operations:	Run	Runs the deterministic client worker runnable.

Name:	WorkerThread	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface is used to implement worker threads for the DeterministicClient.	
Operations:	GetRandom	Returns a deterministic pseudo-random number which is unique for each container element in the worker pool.

9.2.1.1.3 Interfaces for State Management

The StateClient API (cf. Figure 9.5) provides operations to control FunctionGroups and their FunctionGroupStates.

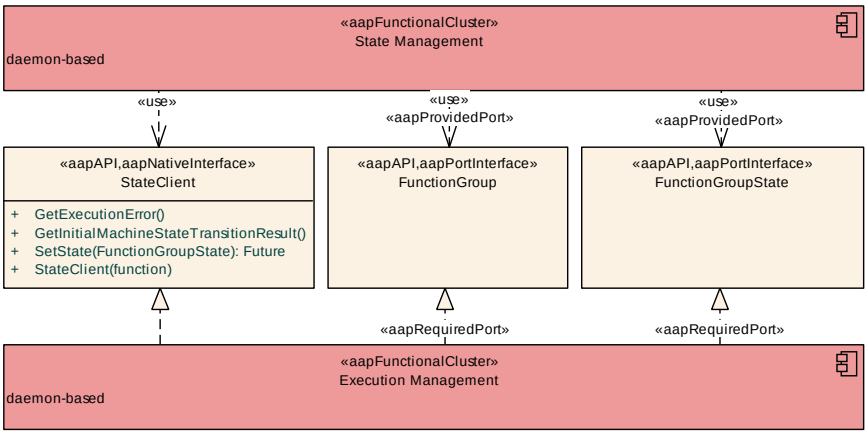

Figure 9.5: Interfaces for State Management

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:	GetExecutionError	Returns the execution error which changed the given FunctionGroup to an undefined FunctionGroupState.
	GetInitialMachineStateTransitionResult	Retrieve the result of `Machine State` initial transition to `Startup` state.
	SetState	Request a FunctionGroupState transition for a single FunctionGroup.
	StateClient	Constructor of this interface. It requires a callback to be invoked if a FunctionGroup changes its state unexpectedly to an undefined FunctionGroupState, i.e. without previous request by SetState(). The affected FunctionGroup is provided as an argument to the callback.

Name:	FunctionGroupState
Technology:	Port interface
Generated:	No
Meta-model interface type:	`ModeDeclaration`
Usage:	Public API
Description:	Represents a `Function Group State` defined in the `Manifest`.

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

9.2.1.2 Provided interfaces

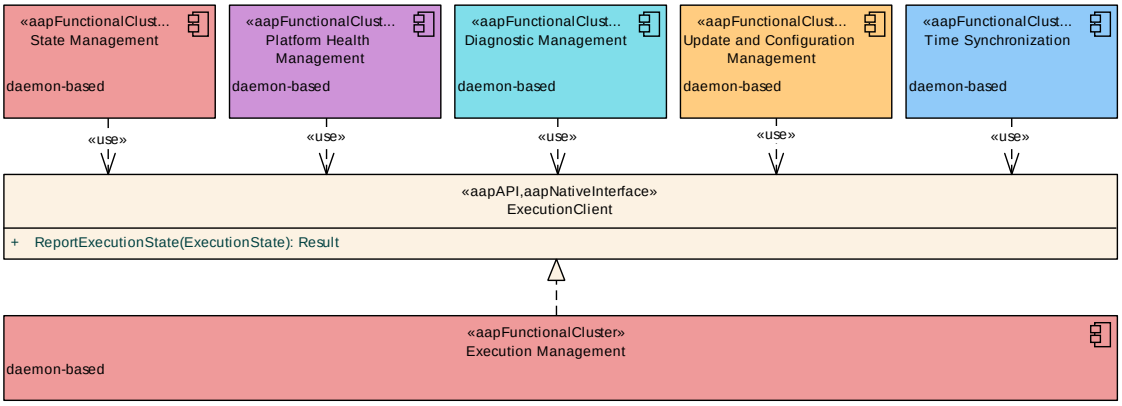

Figure 9.6: Users of the ExecutionClient interface

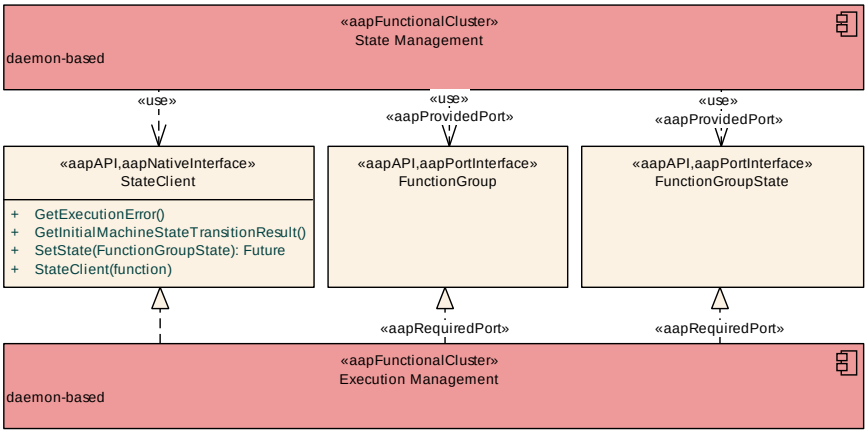

Figure 9.7: Users of the State Management interfaces

Interface	***Requiring functional clusters***
Execution Management::ExecutionClient	Diagnostic Management
	Platform Health Management
	State Management
	Time Synchronization

△

Interface	*Requiring functional clusters*
	Update and Configuration Management
Execution Management::FunctionGroup	State Management
Execution Management::FunctionGroupState	State Management
Execution Management::StateClient	State Management

Table 9.1: Interfaces provided by Execution Management to other Functional Clusters

9.2.1.3 Required interfaces

Figure 9.8: Interfaces required by Execution Management

Interface	*Purpose*
Multi-Process System Interface	`Execution Management` shall use this interface to start, configure and control os-level processes.
Adaptive Intrusion Detection System Manager::EventReporter	`Execution Management` shall use this interface to report security events (e.g., verification failure of an executable).
Execution Management::WorkerRunnable	`Execution Management` shall use this interface to execute WorkerRunnables using its DeterministicClient implementation.
Log and Trace::Logger	`Execution Management` shall use this interface to log standardized messages.
Persistency::KeyValueStorage	`Execution Management` should use this interface to read/write persistent data.
Platform Health Management::SupervisedEntity	`Execution Management` shall use this interface to enable supervision of its process(es) by `Platform Health Management`.

▽

△

Interface	*Purpose*
Registry::ManifestAccessor	`Execution Management` shall use this interface to read the configuration of its DeterministcClient and information about `Function Group`s and `Process`es from the `Manifest`s.
Time Synchronization::SynchronizedTimeBase Consumer	The DeterministicClient implementation in `Execution Management` should use this interface to synchronize execution of DeterministicClients.

Table 9.2: 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 State`s of all `Function Group`s running on the AUTOSAR Runtime for Adaptive Applications. `State Management` delegates to `Execution Management` to switch the individual `Function Group`s to the respective `Function Group State`s. `State Management` is a unique component in the AUTOSAR Adaptive Platform because it is not part of an AUTOSAR Adaptive Platform stack. The logic of `State Management` currently needs to 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 for triggering state changes (see Section 9.2.2.1.1), interfaces for diagnostic reset (see Section 9.2.2.1.2), interfaces for requesting a `Power Mode` (see Section 9.2.2.1.3), and interfaces for interaction with `Update and Configuration Management` (see Section 9.2.2.1.4).

9.2.2.1.1 Interfaces for triggering state changes

`State Management` provides several interface blueprints to get and set its internal state (cf. Figure 9.9).

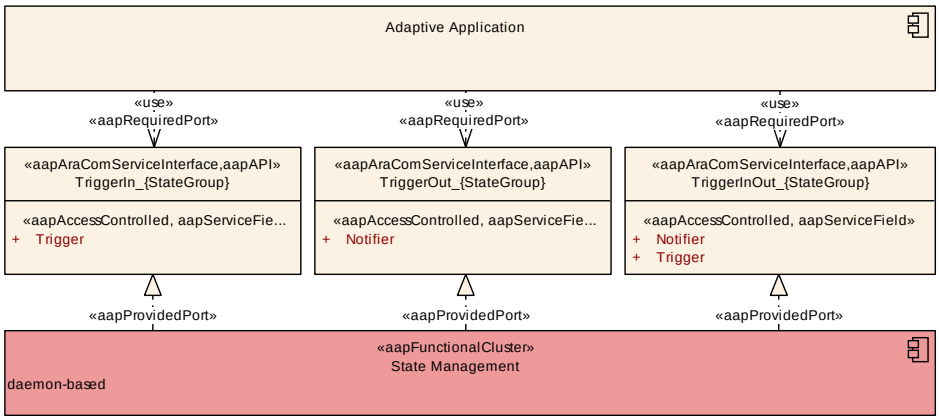

Figure 9.9: Interfaces for triggering state changes

Name:	TriggerIn_{StateGroup}	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface is to be used by `Adaptive Application`s to trigger `State Management` to change its internal state.	
Fields:	Trigger	A value to be evaluated by `State Management` in a project-specific way.

Name:	TriggerOut_{StateGroup}	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface is to be used by `Adaptive Application`s to be informed when `State Management` has changed its internal state.	
Fields:	Notifier	To be set by `State Management` in a project-specific way to inform `Adaptive Application`s about changes within `State Management`.

Name:	TriggerInOut_{StateGroup}	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface is to be used by `Adaptive Application`s to trigger `State Management` to change its internal state and to get information when it is carried out.	
Fields:	Notifier	To be set by `State Management` in a project-specific way to inform `Adaptive Application`s about changes within `State Management`.
	Trigger	A value to be evaluated by `State Management` in a project-specific way.

9.2.2.1.2 Interfaces for requesting a diagnostic reset

The DiagnosticReset interface propagates a diagnostic reset request (DiagnosticReset::message()) to all affected `Process`es. These `Process`es then shall answer the diagnostic reset request by calling DiagnosticReset::event().

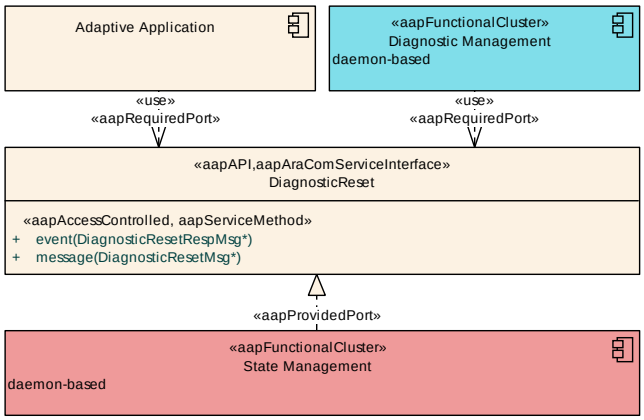

Figure 9.10: Interface for handling a diagnostic reset

Name:	DiagnosticReset	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface provides functionality to handle diagnostic reset requests.	
Operations:	event	All `Process`es which got a diagnostic reset request shall call this method to provide an answer to `State Management`.
	message	Sends a diagnostic reset message to all affected `Processes`.

9.2.2.1.3 Interfaces for requesting a Power Mode

The PowerMode interface propagates a diagnostic `Power Mode` request (PowerMode::message()) to all running `Process`es. These `Process`es then shall answer the `Power Mode` request by calling PowerMode::event().

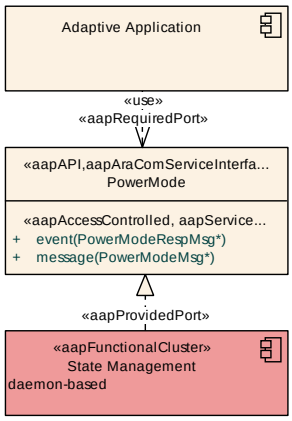

Figure 9.11: Interface for handling a Power Mode request

Name:	PowerMode	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface provides functionality to handle `Power Mode` requests.	
Operations:	event	All `Processes` which have received a `Power Mode` request shall call this method to provide an answer to `State Management`.
	message	Sends a `Power Mode` request to all running `Processes`.

9.2.2.1.4 Interfaces for interaction with Update and Configuration Management

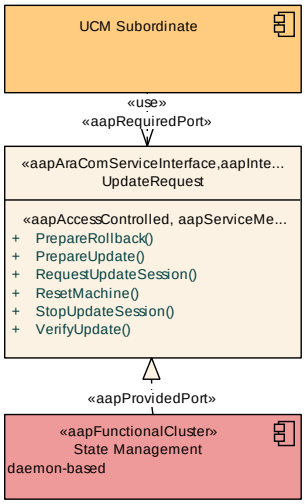

Figure 9.12: Interface for software update handling

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 Group`s after an update.

9.2.2.2 Provided interfaces

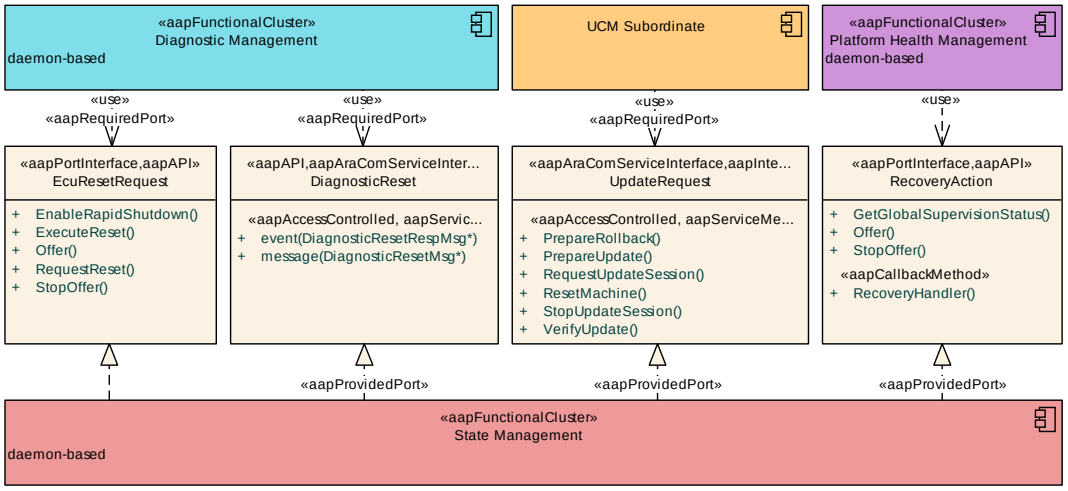

Figure 9.13: Users of the State Management interfaces

Interface	*Requiring functional clusters*
Diagnostic Management::Communication Control	Diagnostic Management
Diagnostic Management::EcuResetRequest	Diagnostic Management
Platform Health Management::RecoveryAction	Platform Health Management
State Management::DiagnosticReset	Diagnostic Management
State Management::UpdateRequest	Update and Configuration Management

Table 9.3: Interfaces provided by State Management to other Functional Clusters

9.2.2.3 Required interfaces

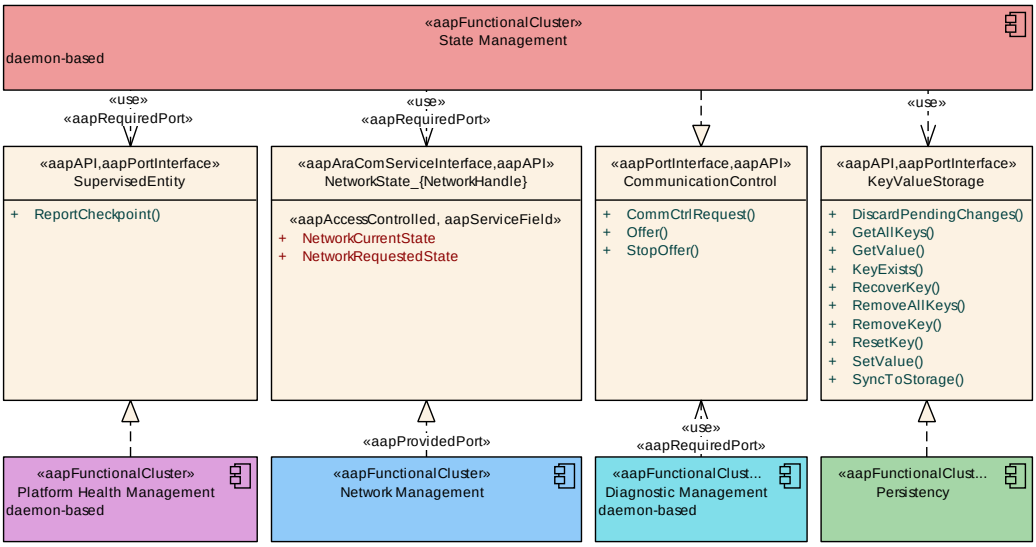

Figure 9.14: Interfaces required by State Management

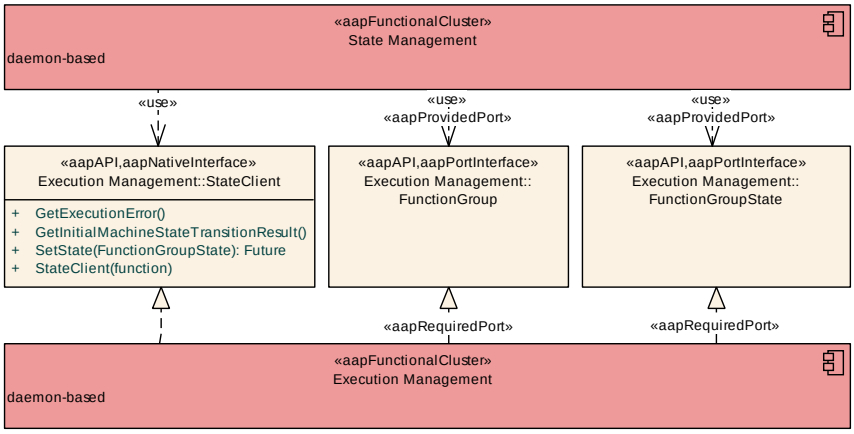

Figure 9.15: Interfaces of Execution Management required by State Management

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 request FunctionGroupState transitions and to check their status.
Execution Management::FunctionGroupState	This interface shall be used to request FunctionGroupState transitions and to check their status.
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::NetworkState_{Network Handle}	This interface shall be used to retrieve information about the network status of a `NetworkHandle`.
Persistency::KeyValueStorageOperations	This interface should be used to persist information (e.g. update session).
Persistency::KeyValueStorage	This interface should be used to persist information (e.g. update session).
Platform Health Management::SupervisedEntity	`State Management` shall use this interface to enable supervision of its process(es) by `Platform Health Management`.

Table 9.4: 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 ().

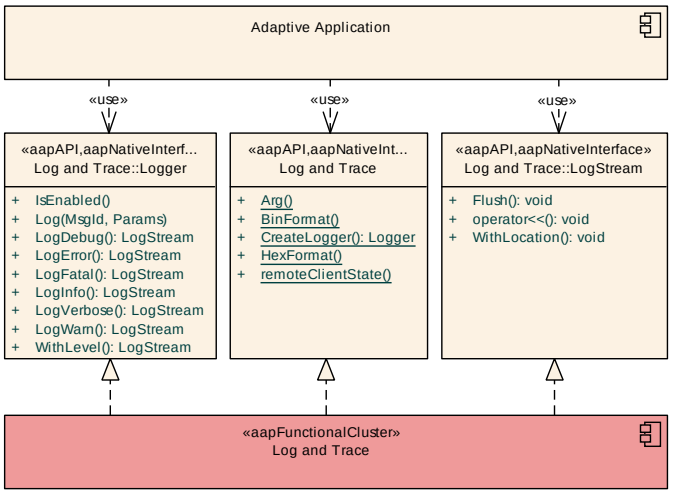

Figure 9.16: Interfaces of Log and Trace

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

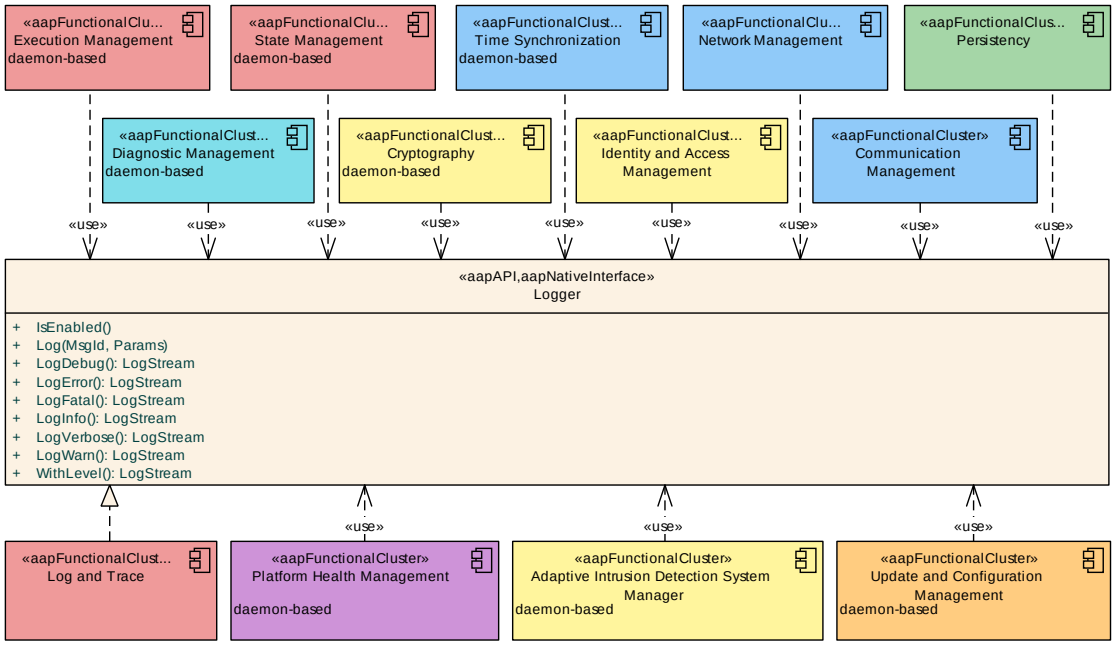

Figure 9.17: Users of the Log and Trace interfaces

Interface	*Requiring functional clusters*
Log and Trace::Logger	Adaptive Intrusion Detection System Manager
	Communication Management
	Cryptography
	Diagnostic Management
	Execution Management
	Identity and Access Management
	Network Management
	Persistency
	Platform Health Management
	State Management
	Time Synchronization
	Update and Configuration Management

Table 9.5: Interfaces provided by Log and Trace to other Functional Clusters

9.2.3.3 Required interfaces

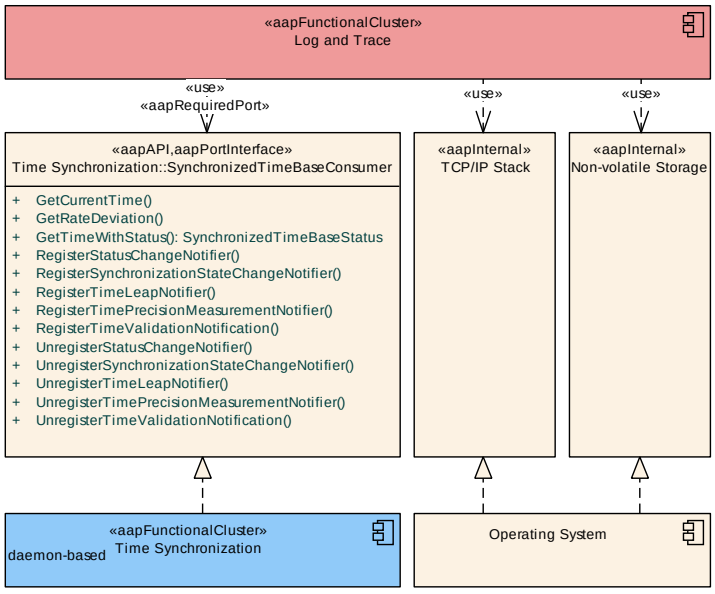

Figure 9.18: Interfaces required by LogAndTrace

Interface	*Purpose*
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.
TCP/IP Stack	`Log and Trace` shall use this interface to write log messages to an IP-based network stream.
Registry::ManifestAccessor	`Log and Trace` shall use this interface to read information about modeled messages from the `Manifest`s.
Time Synchronization::SynchronizedTimeBase Consumer	`Log and Trace` shall use this interface to determine the timestamps that are associated with log messages.

Table 9.6: 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

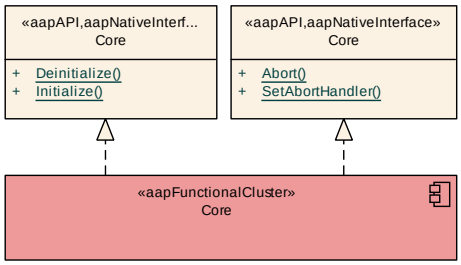

Figure 9.19: Interfaces of Core

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.

9.2.4.2 Provided interfaces

`Core` currently provides no interfaces to other `Functional Cluster`s.

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 functionality for implementing multi-threaded real-time embedded applications and corresponds to 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

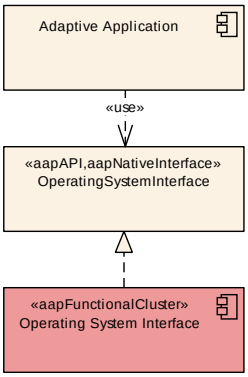

Figure 9.20: Interfaces defined by Operating System Interface

Name:	OperatingSystemInterface
Technology:	Native interface
Usage:	Public API
Description:	This interface represents the POSIX PSE51 profile API. The API is not detailed in this document.

9.2.5.2 Provided interfaces

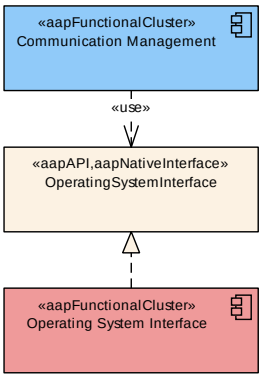

Figure 9.21: Users of the OperatingSystemInterface interfaces

Interface	*Requiring functional clusters*
Operating System Interface::OperatingSystem Interface	Communication Management

Table 9.7: Interfaces provided by Operating System Interface to other Functional Clusters

9.2.5.3 Required interfaces

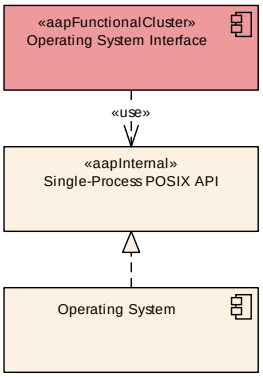

Figure 9.22: Interfaces required by Operating System Interface

Interface	*Purpose*
Single-Process POSIX API	`Operating System Interface` uses this interface to provide the functionality of the POSIX PSE51 profile, e.g. `Thread`s. Usually the whole POSIX PSE51 profile API will already be provided by the underlying operating system. In this case, the `Operating System Interface` will not have an implementation in the Adaptive Platform stack.

Table 9.8: Interfaces required by Operating System Interface

9.3 Communication

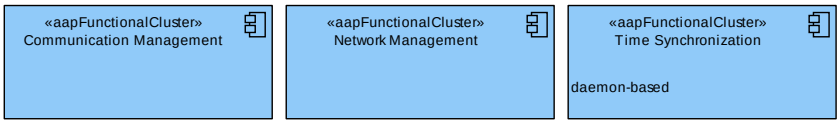

Figure 9.23: Functional Clusters in category Communication

9.3.1 Communication Management

Name:	Communication Management
Short Name:	com
Category:	Communication
Daemon-based:	No

△

Responsibilities:	`Communication Management` is responsible for all levels of service-oriented and raw 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 raw data streams (see Section 9.3.1.1.1), 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`, e.g. a `Proxy` and a `Skeleton` interface. However, these generated interfaces are not covered in this document.

9.3.1.1.1 Interfaces for raw data streams

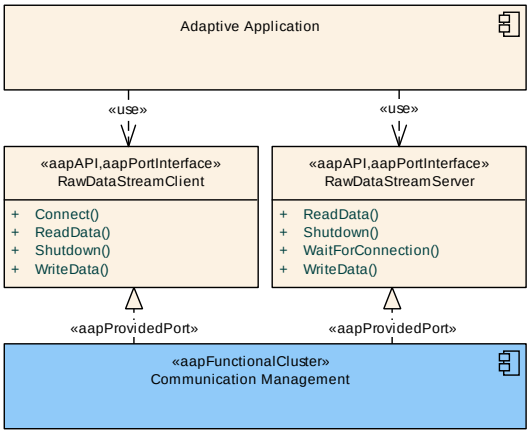

Figure 9.24: Interfaces for raw data streams

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 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.
	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 network connection.	
Operations:	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 RawDataStreamServer 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.

9.3.1.1.2 Interfaces for SecOC

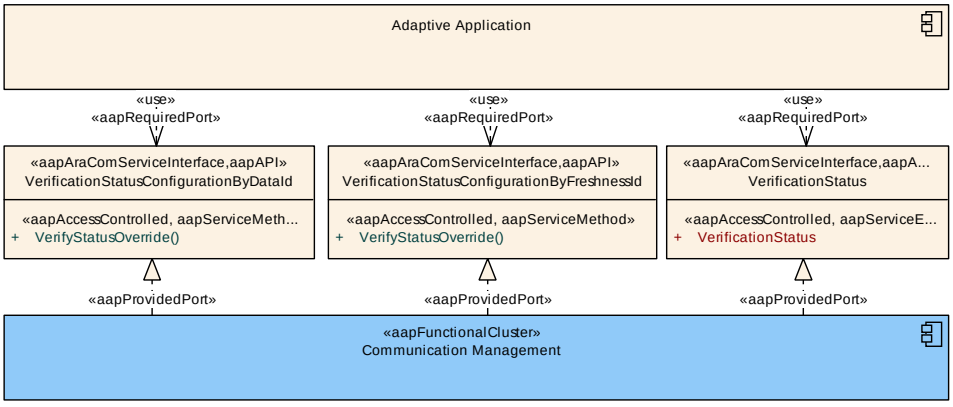

Figure 9.25: Interfaces for SecOC

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

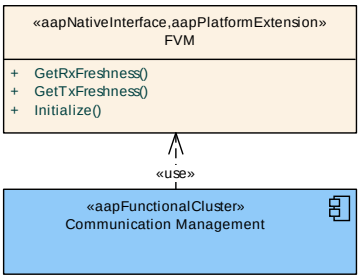

Figure 9.26: Interfaces for freshness value management

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 Clusters.

9.3.1.3 Required interfaces

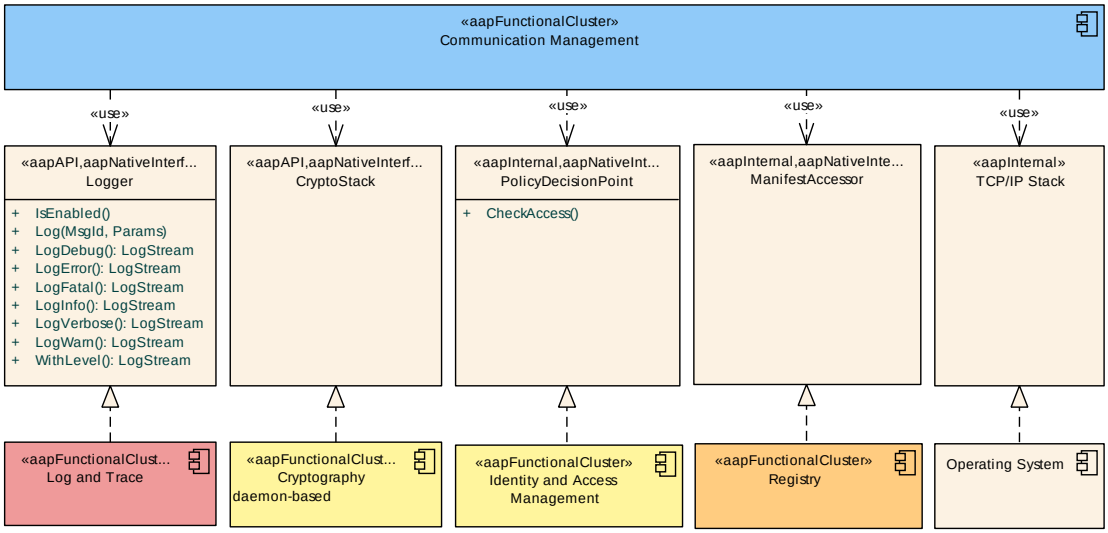

Figure 9.27: Interfaces required by Communication Management

Interface	***Purpose***
TCP/IP Stack	Communication Management shall use this interface to establish and control TCP/IP-based network connections.
Adaptive Intrusion Detection System Manager::EventReporter	Communication Management shall use this interface to report security events.
Communication Management::FVM	Communication Management shall use this interface to get freshness values.
Cryptography::CryptoStack	Communication Management shall use this interface to establish encrypted connections and generate / verify checksums (MAC).
Identity and Access Management::Policy DecisionPoint	Communication Management shall use this interface to check access to ara::com service methods, fields, and events.

▽

△

Interface	*Purpose*
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 `Thread`s used by the implementation.
Registry::ManifestAccessor	`Communication Management` shall use this interface to read service information from the `Manifest`s.

Table 9.9: Interfaces required by Communication Management

9.3.2 Network Management

Name:	Network Management
Short Name:	nm
Category:	Communication
Daemon-based:	No
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.

9.3.2.1 Defined interfaces

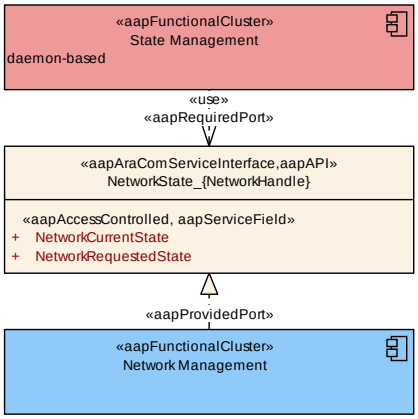

Figure 9.28: Interfaces for time base providers

Name:	NetworkState_{NetworkHandle}
Technology:	ara::com service interface
Usage:	Public API
Description:	This interface provides information about network status per `NetworkHandle`. This interface is intended to be used by `State Management` only.

▽

△

Fields:	NetworkCurrentState	Indicates if a PNC / VLAN / Physical Network is currently active or not.
	NetworkRequestedState	Request a PNC / VLAN / Physical Network to get active or to release it.

9.3.2.2 Provided interfaces

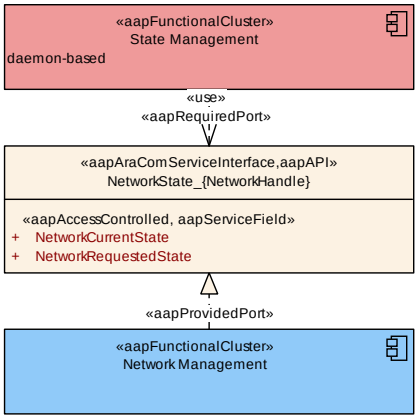

Figure 9.29: Users of Network Management interfaces

Interface	***Requiring functional clusters***
Network Management::NetworkState_{Network Handle}	State Management

Table 9.10: Interfaces provided by Network Management to other Functional Clusters

9.3.2.3 Required interfaces

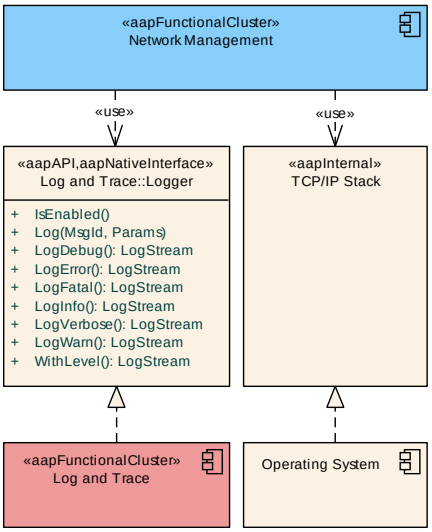

Figure 9.30: Interfaces required by Network Management

Interface	*Purpose*
TCP/IP Stack	`Network Management` should use this interface to set and to determine the status of IP-based networks.
Log and Trace::Logger	`Network Management` shall use this interface to log standardized messages.
Registry::ManifestAccessor	`Network Management` shall use this interface to read information about `NmNetworkHandle`s from the `Manifest`s.

Table 9.11: Interfaces required by Network Management

9.3.3 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 `Machine`s 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.

9.3.3.1 Defined interfaces

The interfaces of `Time Synchronization` are categorized into interfaces for providing time information (see Section 9.3.3.1.1) and interfaces for consuming time information (see Section 9.3.3.1.2).

9.3.3.1.1 Interfaces for time base providers

`Time Synchronization` defines the SynchronizedTimeBaseProvider and OffsetTimeBaseProvider interfaces to provide time information for a synchronized time base.

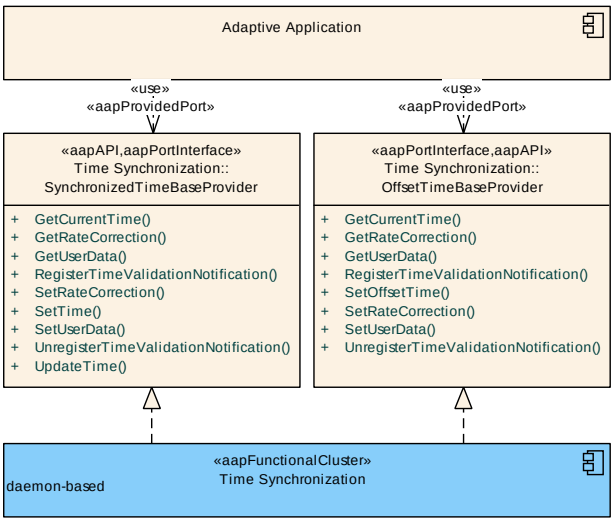

Figure 9.31: Interfaces for time base providers

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:	OffsetTimeBaseProvider	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`SynchronizedTimeBaseProviderInterface`	
Usage:	Public API	
Description:	Provides access to the offset time base. It allows to get the current time point, the rate deviation, the current status and the received user data.	
Operations:	GetCurrentTime	Get 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.
	SetOffsetTime	Set a new offset time value for the clock. Setting a new time also triggers transmission on the bus.
	SetRateCorrection	Set the rate correction that will be applied to time values.
	SetUserData	Set the user data of the time base.
	UnregisterTimeValidationNotification	Un-register the notifier to receive time sync parameters.

9.3.3.1.2 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.

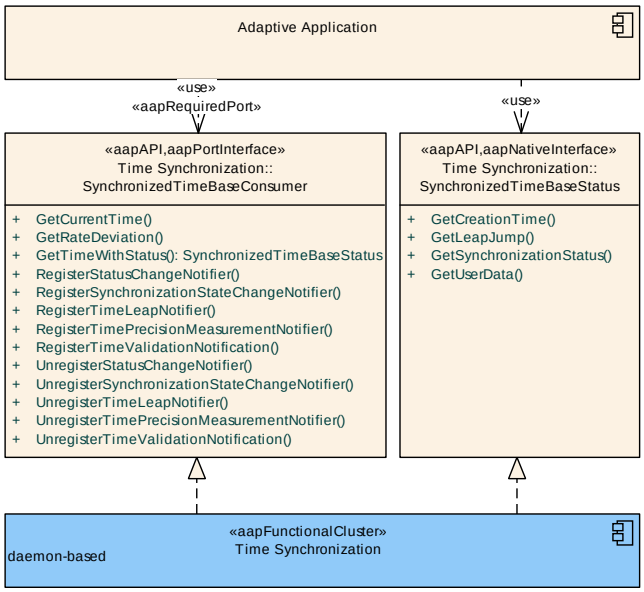

Figure 9.32: Interfaces for time base consumers

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:	GetCurrentTime	Obtain the current time (regardless of the current sync status).
	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 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 a time point including its states.	
Operations:	GetCreationTime	Obtain 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.

9.3.3.2 Provided interfaces

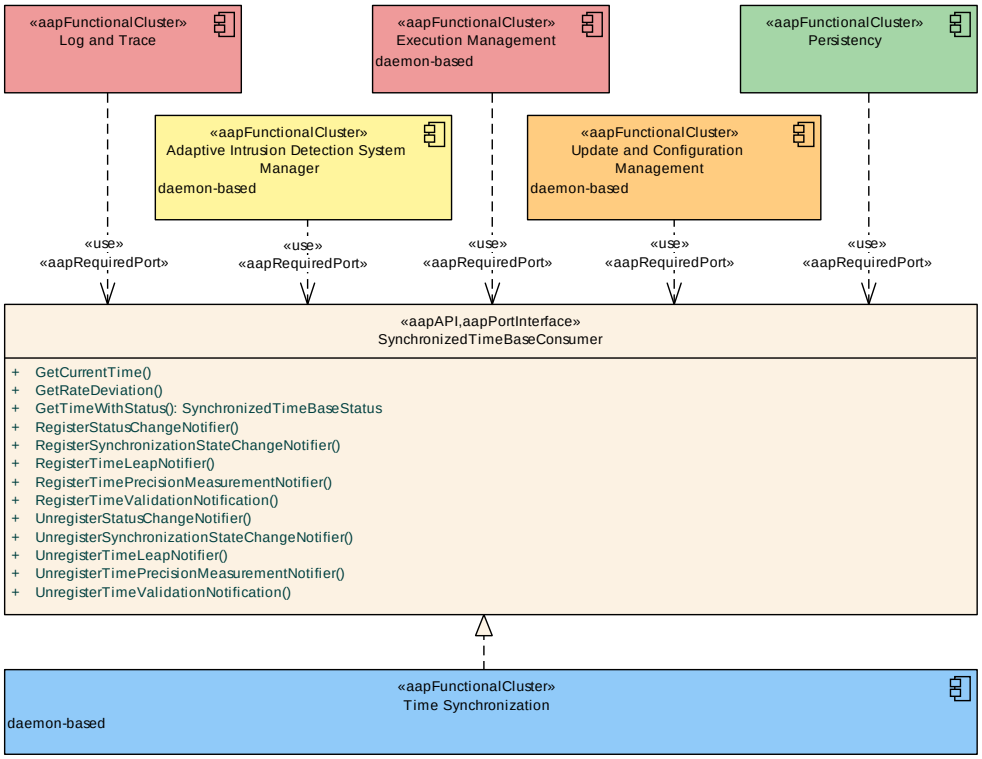

Figure 9.33: Users of Time Synchronization interfaces

Interface	*Requiring functional clusters*
Time Synchronization::SynchronizedTimeBase Consumer	Adaptive Intrusion Detection System Manager
	Execution Management
	Log and Trace
	Persistency
	Update and Configuration Management

Table 9.12: Interfaces provided by Time Synchronization to other Functional Clusters

9.3.3.3 Required interfaces

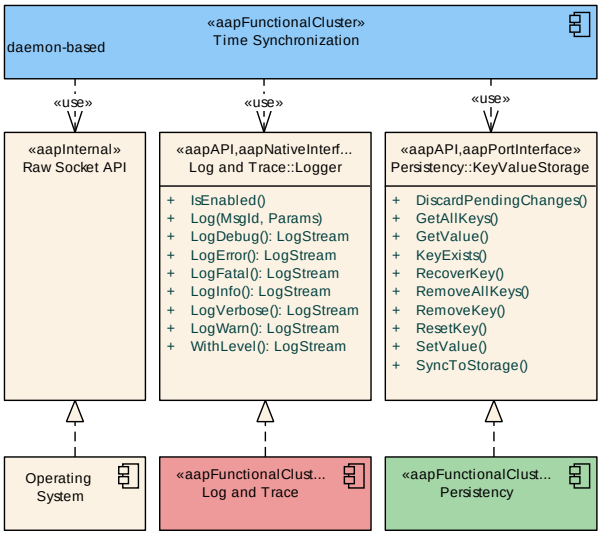

Figure 9.34: Interfaces required by Time Synchronization

Interface	*Purpose*
Raw Socket API	`Time Synchronization` should use this interface to send and receive raw ethernet packets as required by the time synchronization protocol.
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::KeyValueStorageOperations	`Time Synchronization` should use this interface to persist the last received timestamp to enable a faster startup.
Persistency::KeyValueStorage	`Time Synchronization` should use this interface to persist 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`
Registry::ManifestAccessor	`Time Synchronization` shall use this interface to read information about `TimeBaseResource`s as well as their providers and consumers from the `Manifest`s.

Table 9.13: Interfaces required by Time Synchronization

9.4 Storage

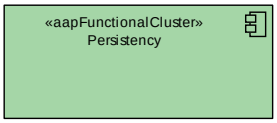

Figure 9.35: Functional Clusters in category Storage

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 `Process`es 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` 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 9.36). A FileStorage has to be opened using OpenFileStorage(). A FileStorage then provides access to several files using their name.

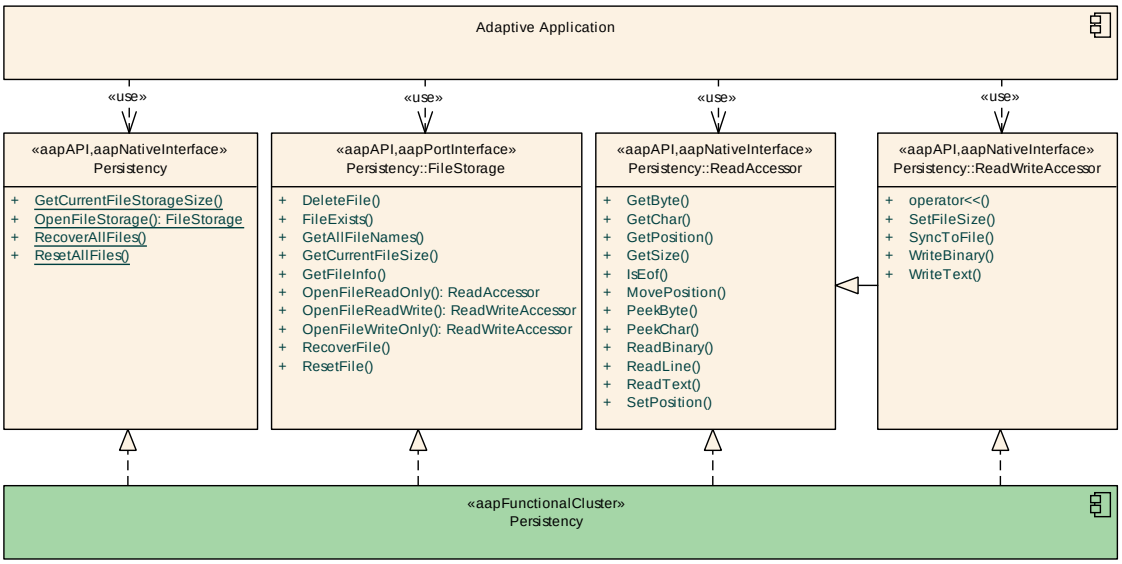

Figure 9.36: Interfaces for file storage

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.

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.
	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.
	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.
	SetPosition	Sets the current position relative to the beginning of 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.	
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.

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 9.37). 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.

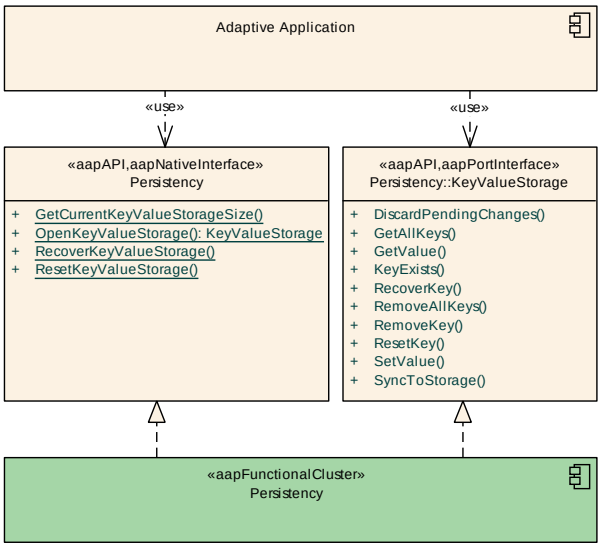

Figure 9.37: Interfaces for key-value-based data storage

Name:	KeyValueStorageOperations	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface provides functions to open and manage KeyValueStorages.	
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.
	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 9.38).

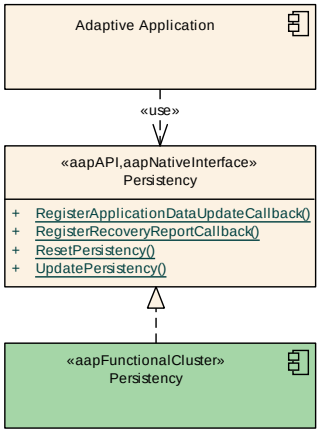

Figure 9.38: Interfaces for general persistency handling

Name:	PersistencyHandlingOperations	
Technology:	Native interface	
Usage:	Public API	
Description:	This interface provides operations manage persistent data.	
Operations:	RegisterApplicationDataUpdate Callback	Registers an application data update callback with `Persistency`.
	RegisterRecoveryReportCallback	Register a recovery reporting 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

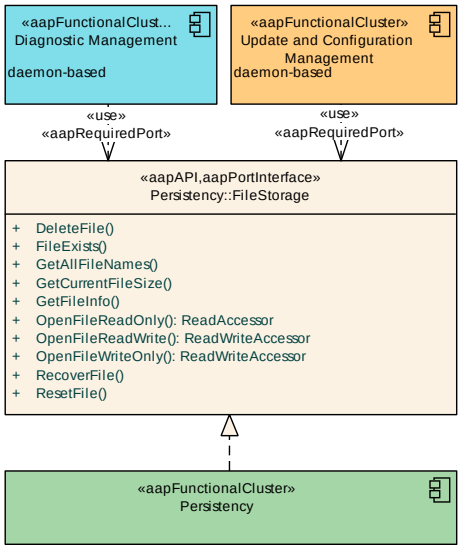

Figure 9.39: Users of the FileStorage interfaces

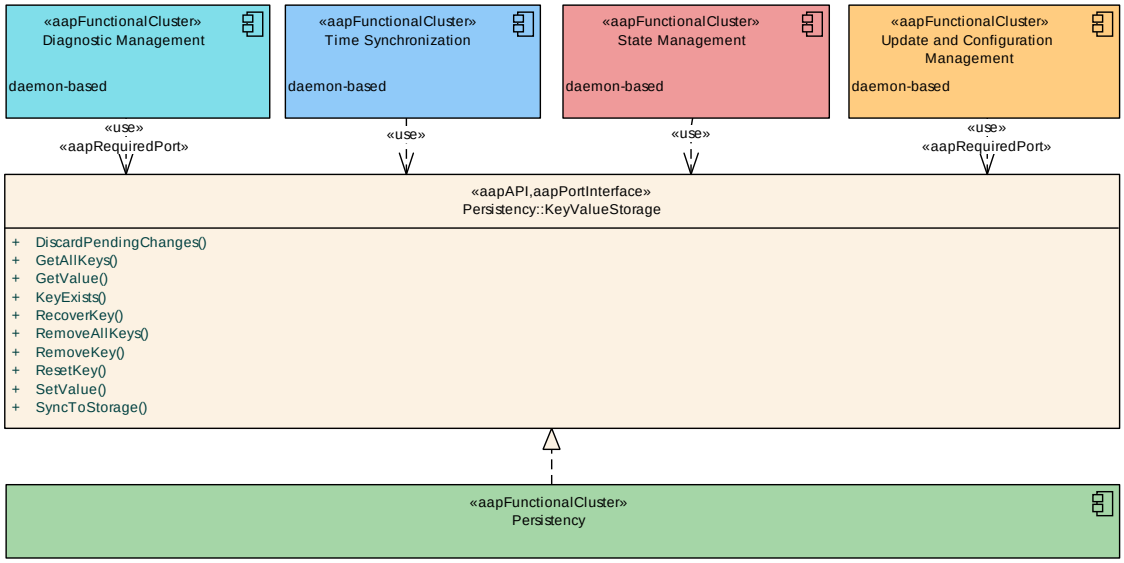

Figure 9.40: Users of the KeyValueStorage interfaces

Interface	*Requiring functional clusters*
Persistency::PersistencyHandlingOperations	Diagnostic Management
	Update and Configuration Management
Persistency::FileStorageOperations	Diagnostic Management
	Update and Configuration Management
Persistency::KeyValueStorageOperations	Diagnostic Management
	State Management
	Time Synchronization
	Update and Configuration Management

△

Interface	*Requiring functional clusters*
Persistency::FileStorage	Diagnostic Management
	Update and Configuration Management
Persistency::KeyValueStorage	Diagnostic Management
	Execution Management
	State Management
	Time Synchronization
	Update and Configuration Management
Persistency::ReadAccessor	Diagnostic Management
	Update and Configuration Management
Persistency::ReadWriteAccessor	Diagnostic Management
	Update and Configuration Management

Table 9.14: Interfaces provided by Persistency to other Functional Clusters

9.4.1.3 Required interfaces

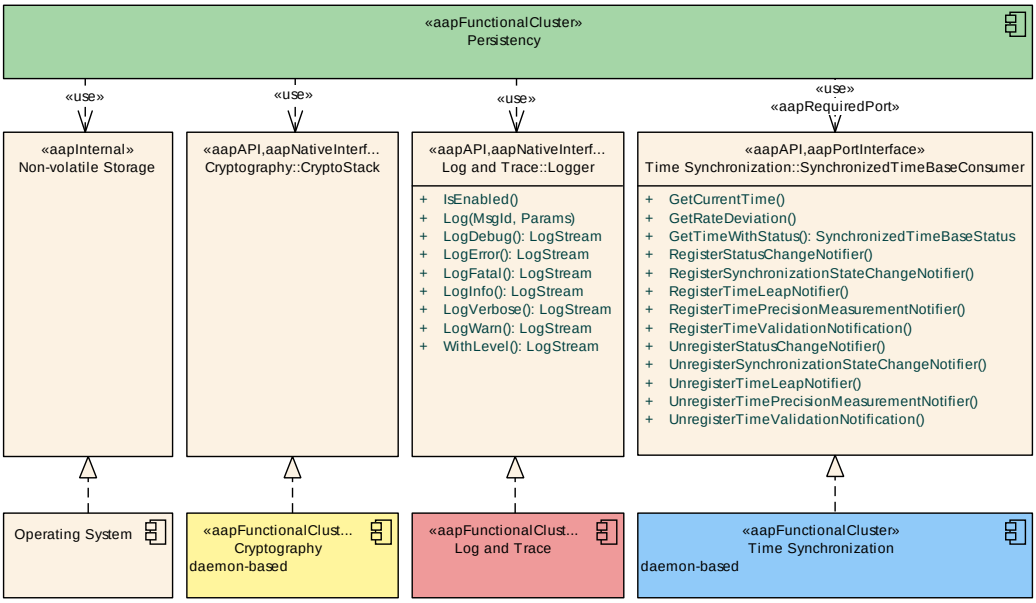

Figure 9.41: Interfaces required by Persistency

Interface	*Purpose*
Non-volatile Storage	`Persistency` uses this interface to access the non-volatile storage provided by the underlying operating system, for example, a file system.
Cryptography::CryptoStack	`Persistency` uses this interface to ensure confidentiality and integrity of the persisted data.
Log and Trace::Logger	`Persistency` shall use this interface to log standardized messages.

▽

△

Interface	*Purpose*
Registry::ManifestAccessor	`Persistency` shall use this interface to read its configuration information from the `Manifest`s.
Time Synchronization::SynchronizedTimeBase Consumer	`Persistency` should use this interface to determine timestamps included in the meta-information of files, e.g., modification timestamp.

Table 9.15: Interfaces required by Persistency

9.5 Security

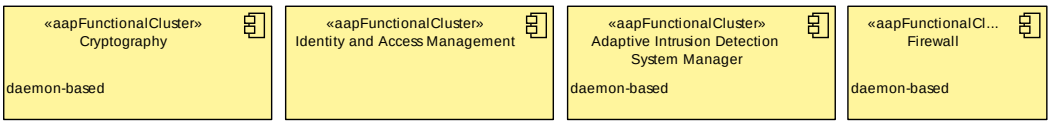

Figure 9.42: Functional Clusters in category Security

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 as reported by `Identity and Access Management`. 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.

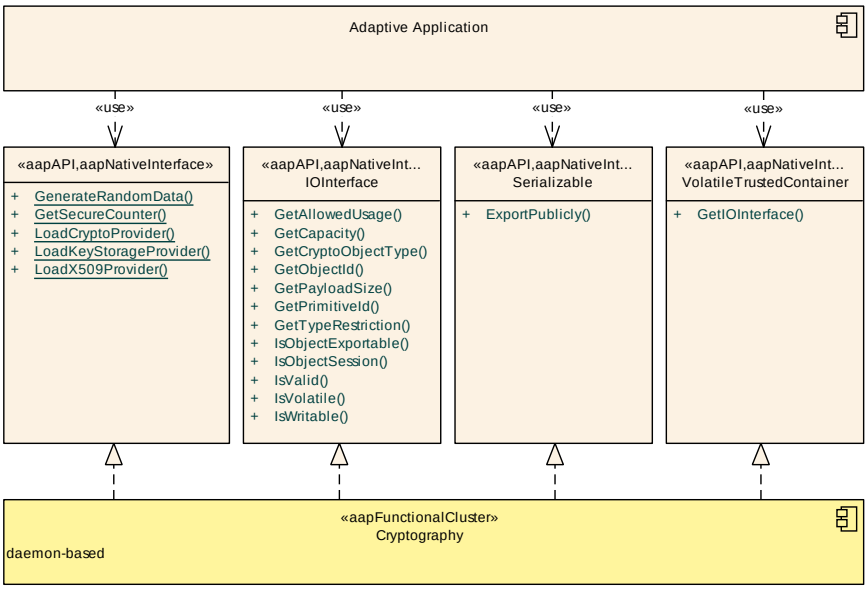

Figure 9.43: Common Interfaces of Cryptography

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.

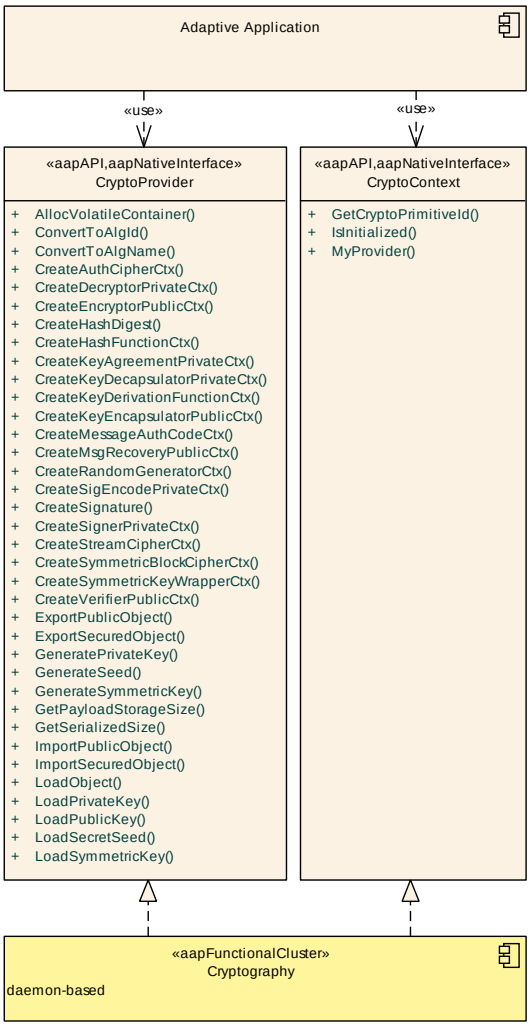

Figure 9.44: General Cryptography Interfaces

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

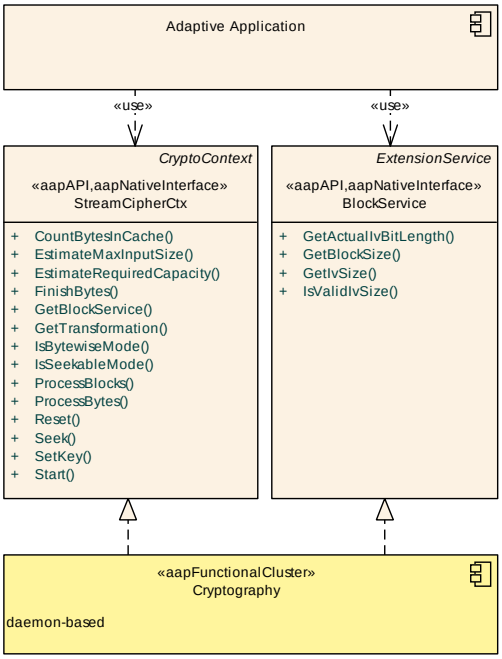

Figure 9.45: BlockService and CryptoContext Interfaces

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).

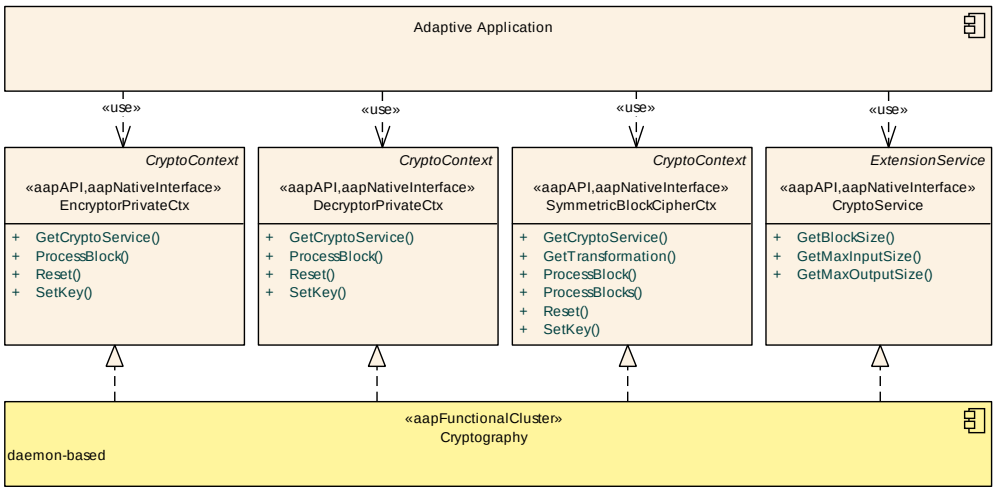

Figure 9.46: CryptoService and CryptoContext Interfaces

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.

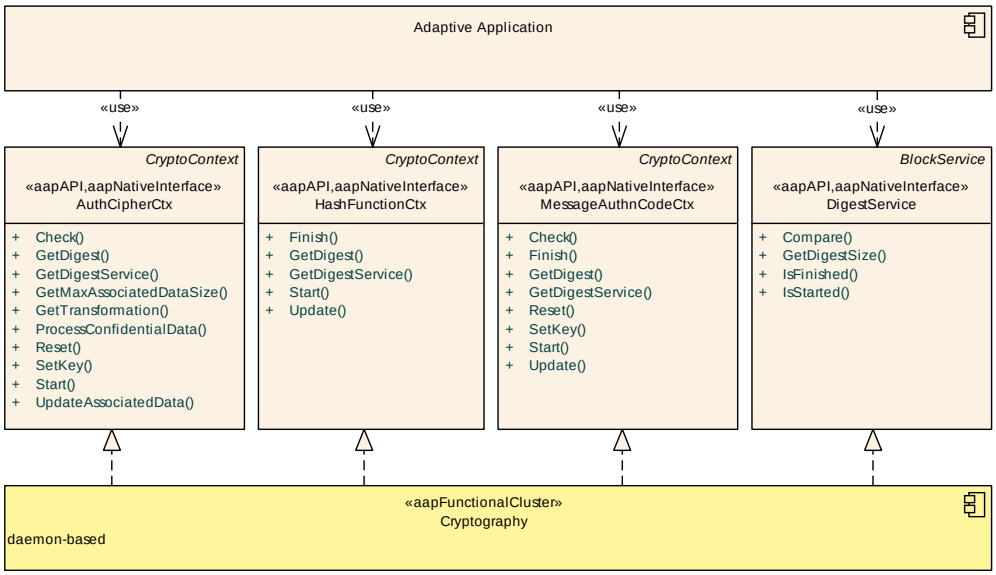

Figure 9.47: DigestService and CryptoContext Interfaces

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.

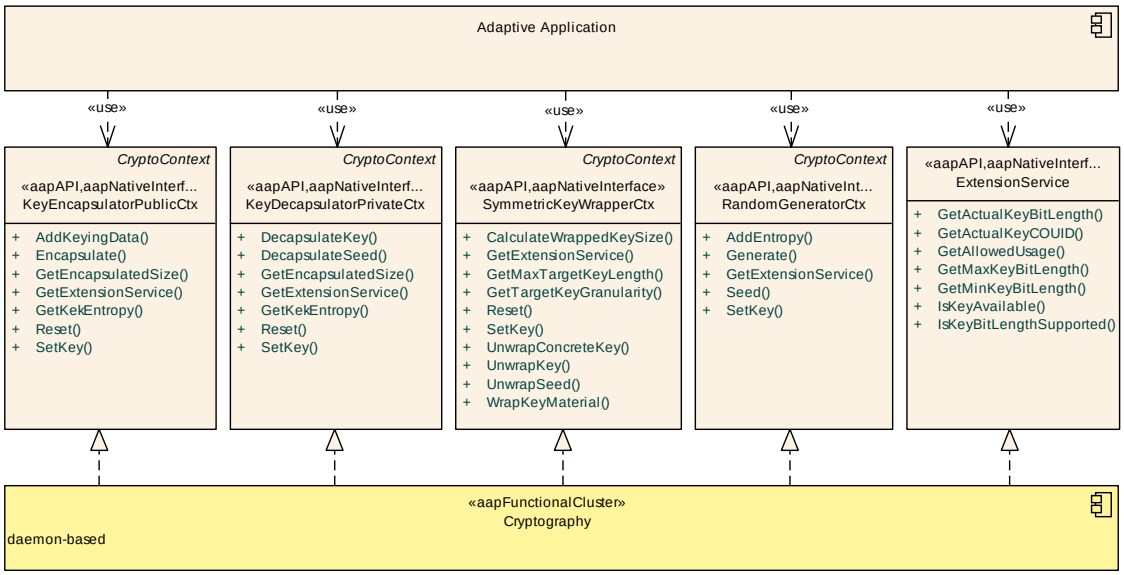

Figure 9.48: ExtensionService and CryptoContext Interfaces (1 of 2)

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.

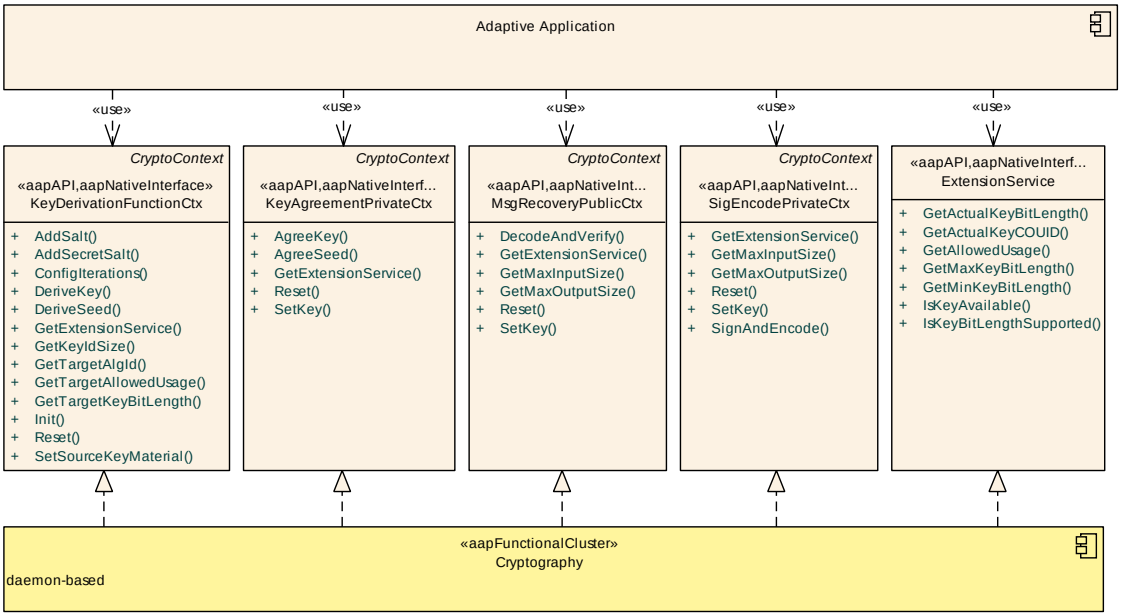

Figure 9.49: ExtensionService and CryptoContext Interfaces (2 of 2)

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.

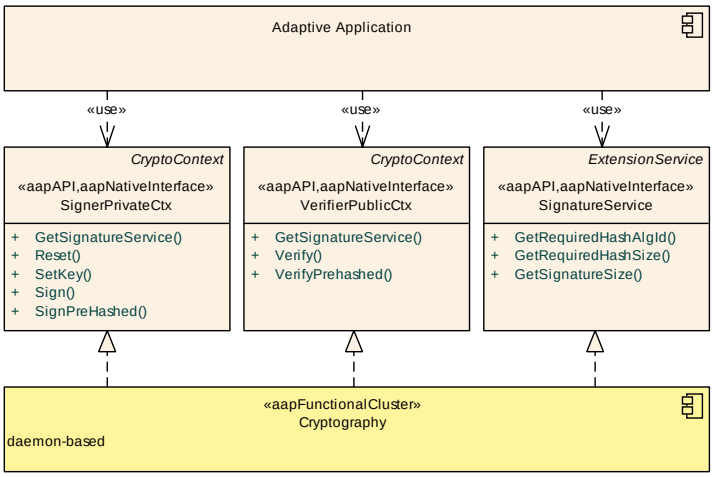

Figure 9.50: SignatureService and CryptoContext Interfaces

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

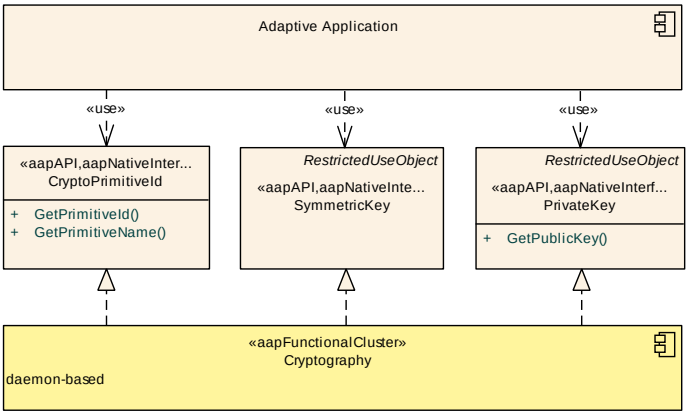

Figure 9.51: Cryptographic Object Interfaces

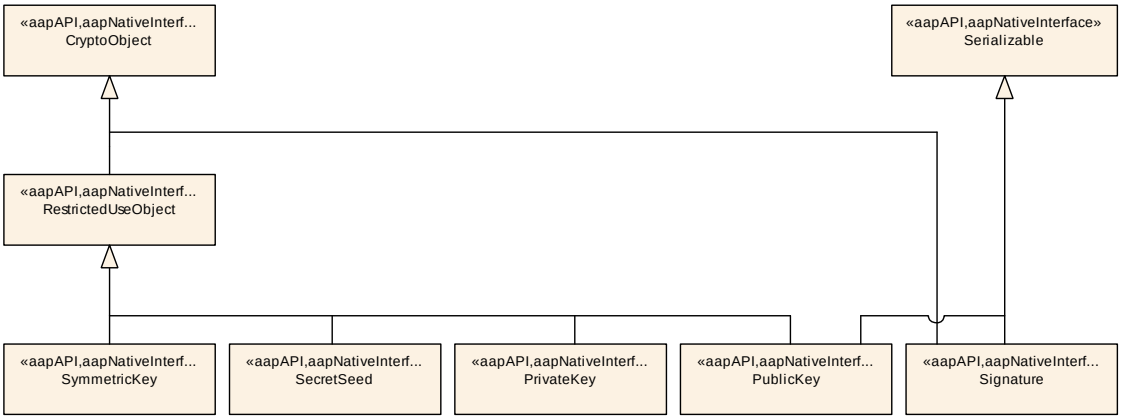

Figure 9.52: Taxonomy of Cryptographic Object Interfaces

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 `CryptoPrimitive`s 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 ^=	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

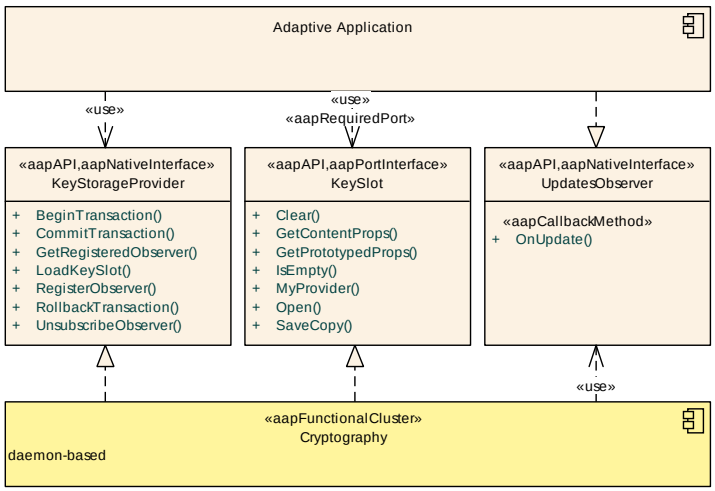

Figure 9.53: Cryptographic Key Handling Interfaces

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

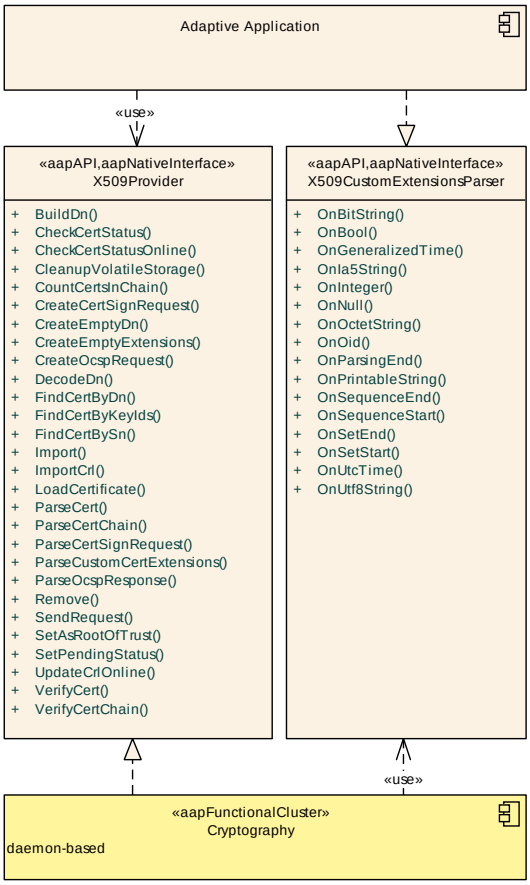

Figure 9.54: X.509 Certificate Handling Interfaces

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.

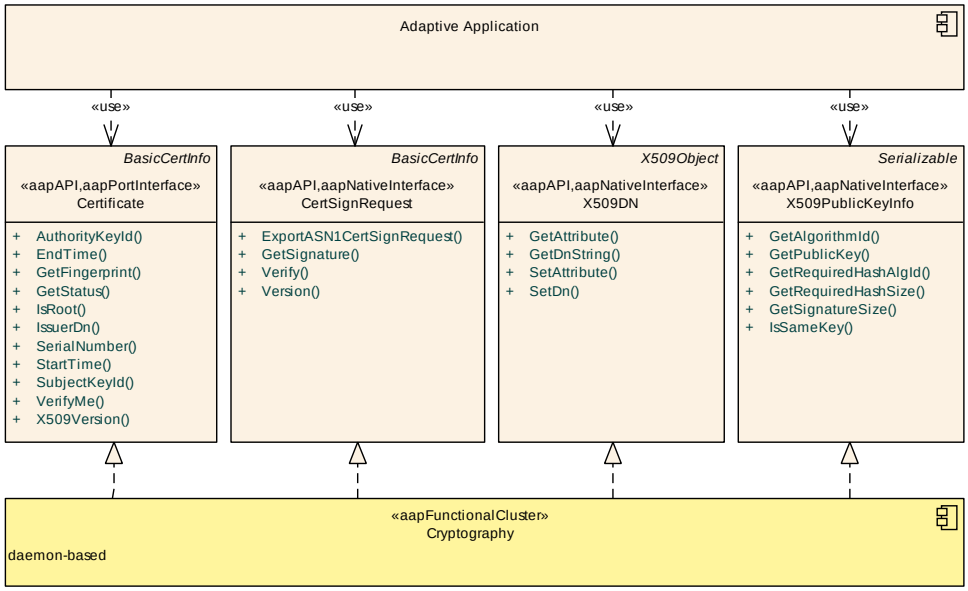

Figure 9.55: X.509 Certificate Object Interfaces

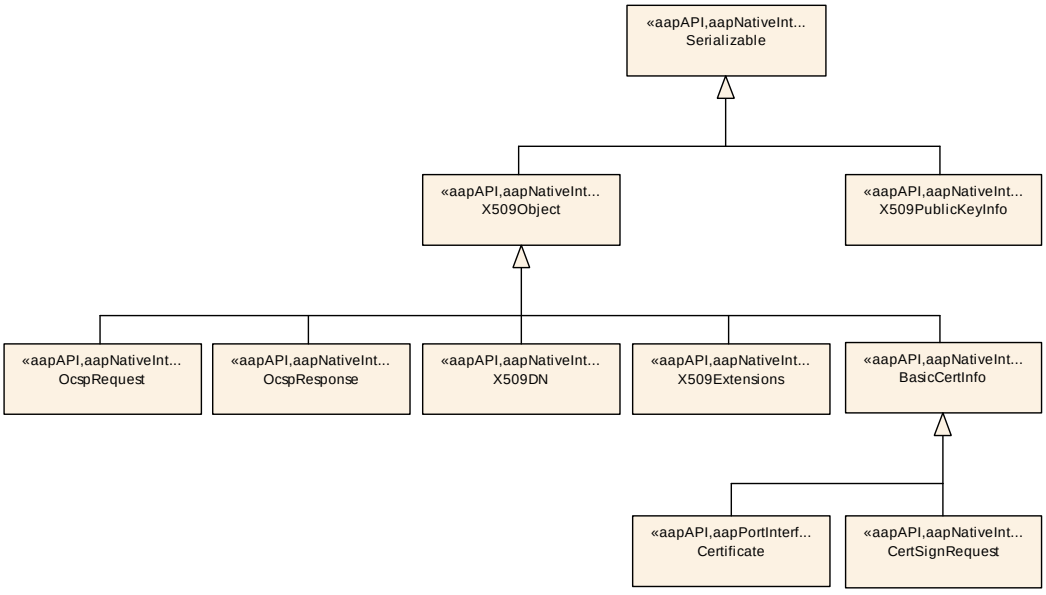

Figure 9.56: Taxonomy of X.509 Certificate Object Interfaces

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

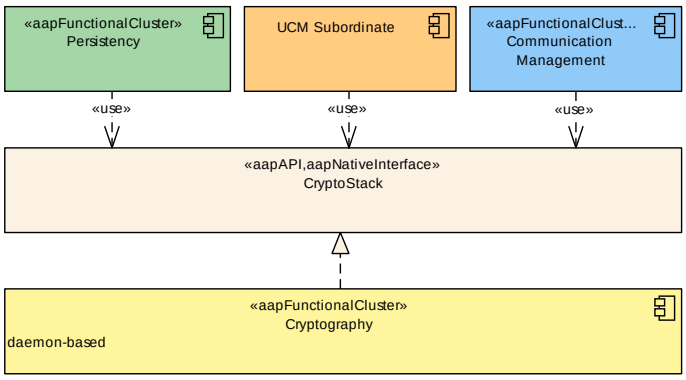

Figure 9.57: Users of the Cryptography interfaces

Interface	*Requiring functional clusters*
Cryptography::CryptoStack	Communication Management
	Persistency
	Update and Configuration Management

Table 9.16: Interfaces provided by Cryptography to other Functional Clusters

9.5.1.3 Required interfaces

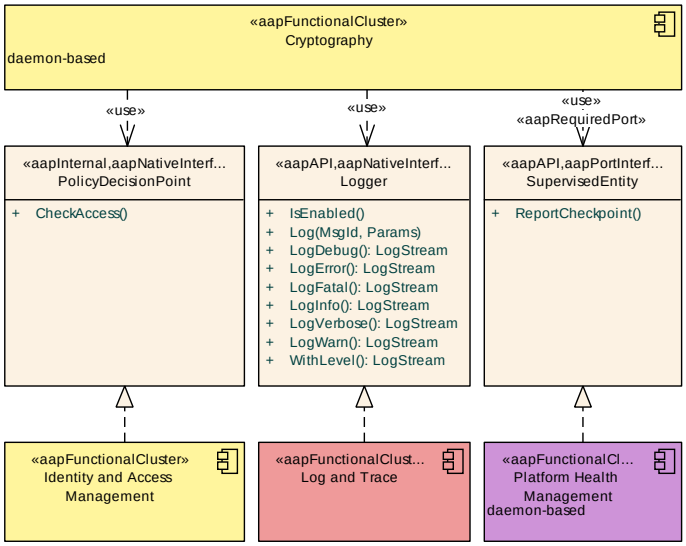

Figure 9.58: Interfaces required by Cryptography

Interface	*Purpose*
Adaptive Intrusion Detection System Manager::EventReporter	This interface should be used to e.g., report attempts to change root certificates.
Cryptography::UpdatesObserver	`Cryptography` uses this interface to notify when a key has been updated.
Cryptography::X509CustomExtensionsParser	`Cryptography` uses this interface for propagating parser events.
Identity and Access Management::PolicyDecisionPoint	`Cryptography` shall use this interface to check access to certificates.
Log and Trace::Logger	`Cryptography` shall use this interface to log standardized messages.
Platform Health Management::SupervisedEntity	`Cryptography` should use this interface for supervision of its daemon process(es).
Registry::ManifestAccessor	`Cryptography` shall use this interface to read its configuration information from the `Manifest`s.

Table 9.17: Interfaces required by Cryptography

9.5.2 Identity and Access Management

Name:	Identity and Access Management
Short Name:	iam
Category:	Security
Daemon-based:	No
Responsibilities:	`Identity and Access Management` checks access to resources of the AUTOSAR Adaptive Platform, for example, on `Service Interfaces` and `Functional Clusters`. `Identity and Access Management` hereby introduces access control for `Adaptive Applications` and protection against privilege escalation in case of attacks. In addition, `Identity and Access Management` enables integrators to verify access on resources requested by `Adaptive Applications` in advance during deployment.

9.5.2.1 Defined interfaces

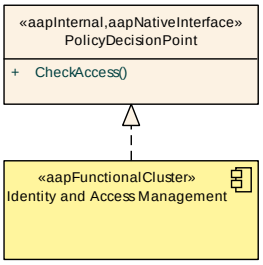

Figure 9.59: Interfaces of Identity and Access Management

Name:	PolicyDecisionPoint	
Technology:	Native interface	
Usage:	Internal	
Description:	This interface serves Policy Enforcement Points with authorization decisions.	
Operations:	CheckAccess	Evaluates an access request against the authorization policies (`Grant`) before issuing an access decision.

9.5.2.2 Provided interfaces

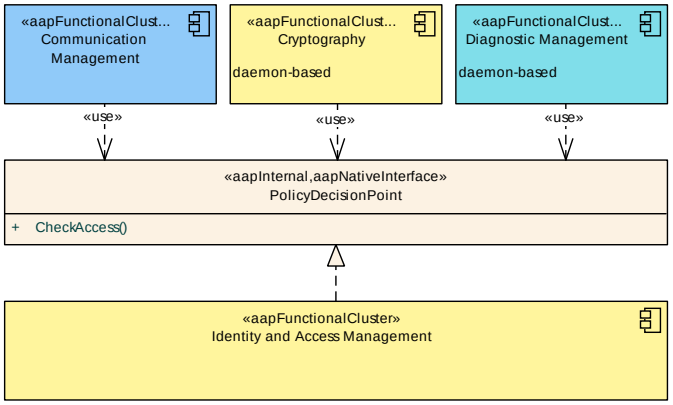

Figure 9.60: Users of the Identity and Access Management interfaces

Interface	*Requiring functional clusters*
Identity and Access Management::Policy DecisionPoint	Communication Management
	Cryptography
	Diagnostic Management

Table 9.18: Interfaces provided by Identity and Access Management to other Functional Clusters

9.5.2.3 Required interfaces

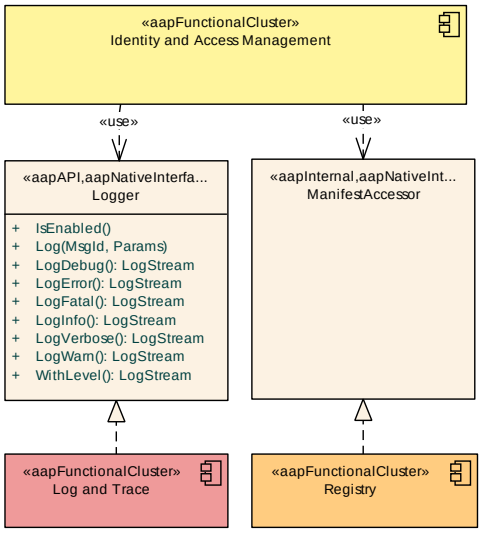

Figure 9.61: Interfaces required by Identity and Access Management

Interface	*Purpose*
Adaptive Intrusion Detection System Manager:: EventReporter	This interface should be used to e.g., report denied access.
Log and Trace::Logger	`Identity and Access Management` shall use this interface to log standardized messages.
Registry::ManifestAccessor	`Identity and Access Management` should use this interface to access `Grant` information modeled in the `Manifest`s.

Table 9.19: Interfaces required by Identity and Access Management

9.5.3 Adaptive Intrusion Detection System Manager

Name:	Adaptive Intrusion Detection System Manager
Short Name:	idsm
Category:	Security

△

Daemon-based:	Yes
Responsibilities:	`Adaptive Intrusion Detection System Manager` provides functionality to report security events.

9.5.3.1 Defined interfaces

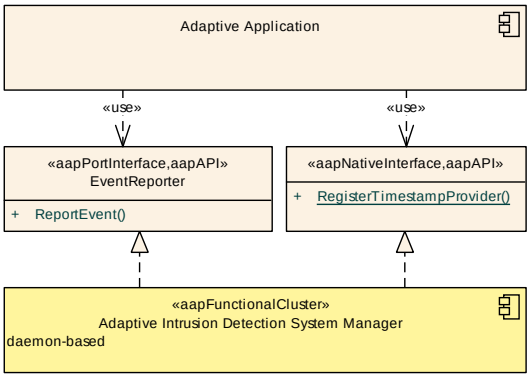

Figure 9.62: Interfaces of Adaptive Intrusion Detection System Manager

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 `Adaptive Intrusion Detection System Manager`.	
Operations:	ReportEvent	Create a new security event at the `Adaptive Intrusion Detection System Manager`.

Name:	TimestampProvider	
Technology:	Native interface	
Usage:	Public API	
Description:	Provides functionality to register a timestamp provider with the `Adaptive Intrusion Detection System Manager`.	
Operations:	RegisterTimestampProvider	Register a callback for providing timestamps to the `Adaptive Intrusion Detection System Manager`.

9.5.3.2 Provided interfaces

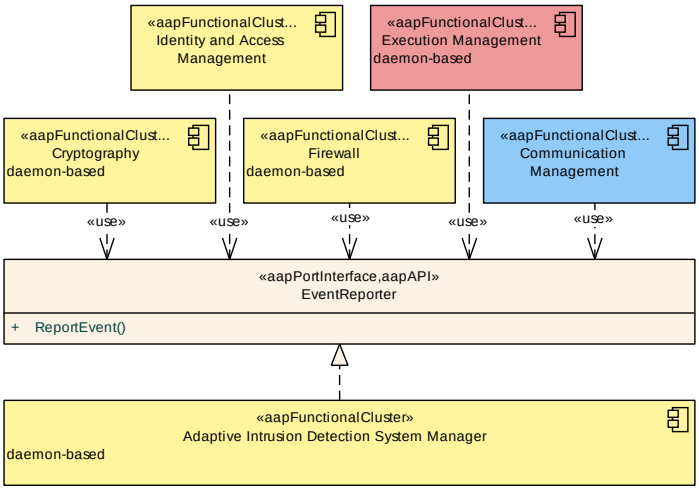

Figure 9.63: Users of the Adaptive Intrusion Detection System Manager interfaces

Interface	*Requiring functional clusters*
Adaptive Intrusion Detection System Manager:: EventReporter	Communication Management
	Cryptography
	Execution Management
	Firewall
	Identity and Access Management

Table 9.20: Interfaces provided by Adaptive Intrusion Detection System Manager to other Functional Clusters

9.5.3.3 Required interfaces

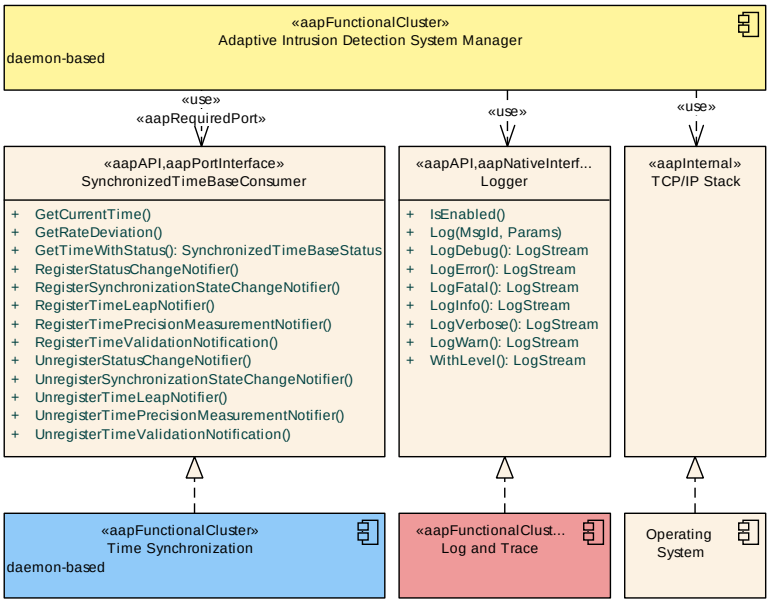

Figure 9.64: Interfaces required by Adaptive Intrusion Detection System Manager

Interface	*Purpose*
TCP/IP Stack	`Adaptive Intrusion Detection System Manager` shall use this interface to propagate qualified security events via the IDS protocol.
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 9.21: Interfaces required by Adaptive Intrusion Detection System Manager

9.5.4 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.

9.5.4.1 Defined interfaces

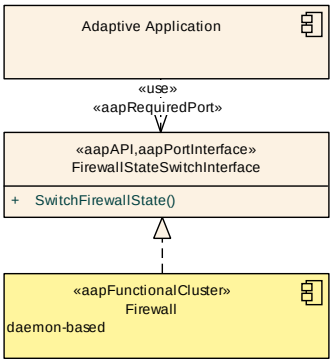

Figure 9.65: Interfaces of Firewall

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.

9.5.4.2 Provided interfaces

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

9.5.4.3 Required interfaces

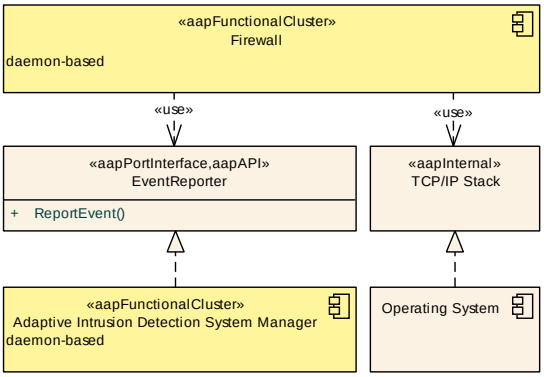

Figure 9.66: Interfaces required by Firewall

Interface	*Purpose*
TCP/IP Stack	The `Firewall` uses this interface to enable/disable firewall rules.
Adaptive Intrusion Detection System Manager:: EventReporter	The `Firewall` uses this interface to report security events.

Table 9.22: Interfaces required by Firewall

9.6 Safety

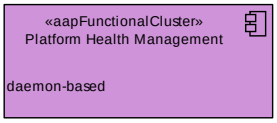

Figure 9.67: Functional Clusters in category Safety

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 9.68). `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).

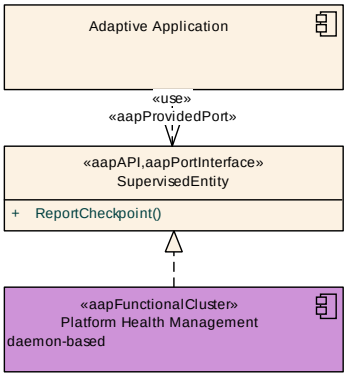

Figure 9.68: Interfaces for supervision

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 9.69).

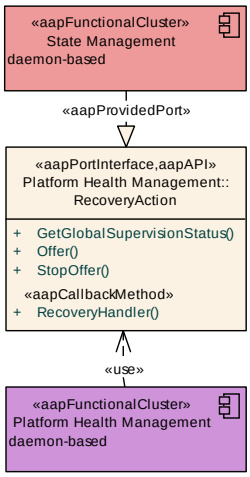

Figure 9.69: Interfaces for recovery

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:	GetGlobalSupervisionStatus	Returns the status of global supervision that the supervised entity belongs to.
	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 9.70).

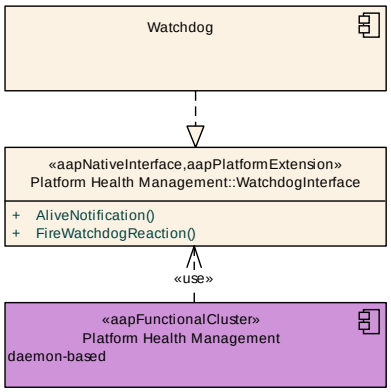

Figure 9.70: Interfaces for watchdog handling

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, WatchdogInterface shall initiate watchdog reaction.
	FireWatchdogReaction	Initiates an error reaction of the hardware watchdog.

9.6.1.2 Provided interfaces

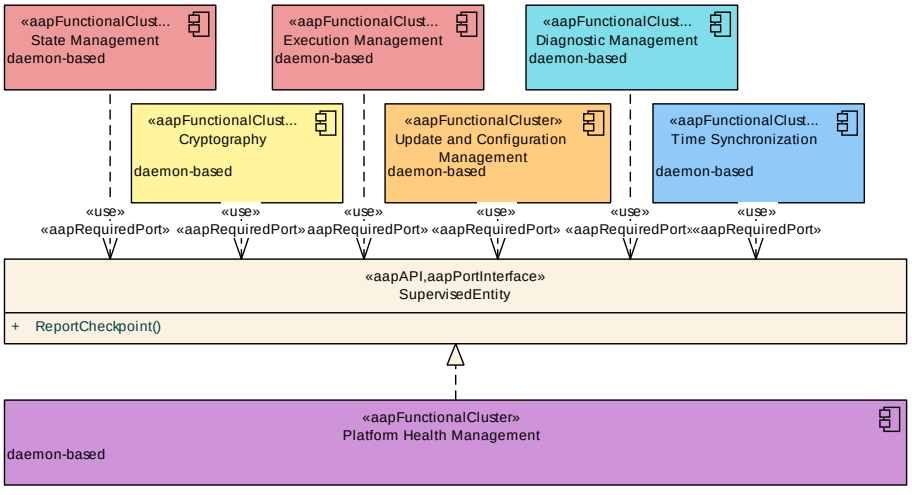

Figure 9.71: Users of the SupervisedEntity interface

Interface	*Requiring functional clusters*
Platform Health Management::SupervisedEntity	Cryptography
	Diagnostic Management
	Execution Management
	State Management
	Time Synchronization
	Update and Configuration Management

Table 9.23: Interfaces provided by Platform Health Management to other Functional Clusters

9.6.1.3 Required interfaces

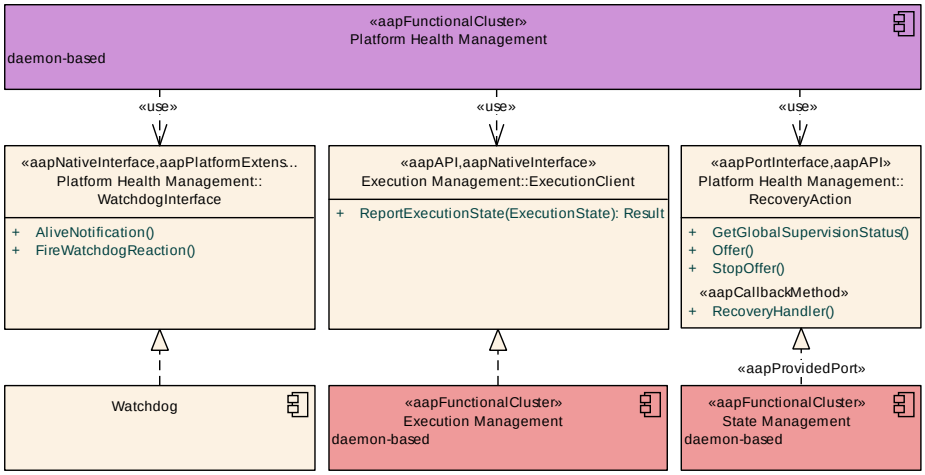

Figure 9.72: Interfaces required by Platform Health Management

Interface	*Purpose*
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::RecoveryAction	`Platform Health Management` uses this interface to trigger failure recovery.
Platform Health Management::Watchdog Interface	`Platform Health Management` uses this interface to control the hardware watchdog.
Registry::ManifestAccessor	`Platform Health Management` shall use this interface to read information about SupervisedEntities from the `Manifest`s.

Table 9.24: Interfaces required by Platform Health Management

9.7 Configuration

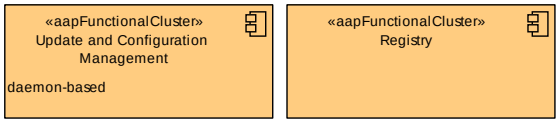

Figure 9.73: Functional Clusters in category Configuration

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. Hereby, `Update and Configuration Management` enables to update the software and its configuration flexibly through over-the-air updates (OTA). `Update and Configuration Management` is separated into two main components `UCM Master` and `UCM Subordinate`. `UCM Subordinate` controls the update process on the local Adaptive Platform. `UCM Master` controls an update of the software in the entire vehicle.

9.7.1.1 Defined interfaces

The interfaces of `Update and Configuration Management` are categorized into interfaces for `UCM Subordinate` (see Section 9.7.1.1.1), interfaces for `UCM Master` (see Section 9.7.1.1.2), and interfaces for the D-PDU API (see Section 9.7.1.1.3).

9.7.1.1.1 UCM Subordinate

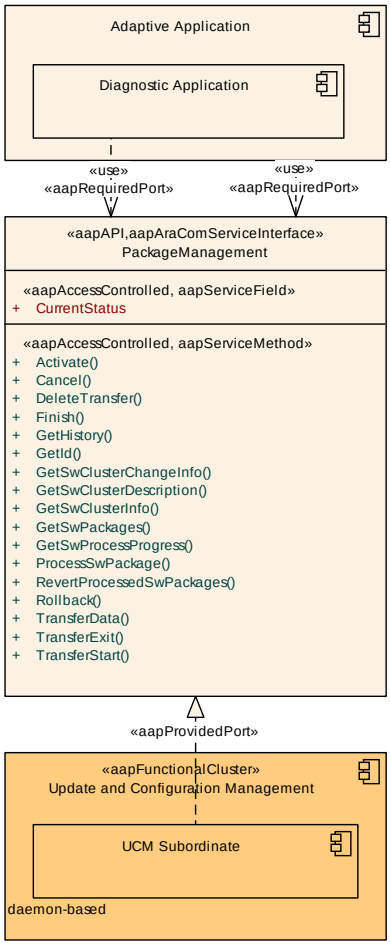

Figure 9.74: Interfaces of UCM Subordinate

Name:	PackageManagement	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface provides functionality for managing and transferring `Software Package`s to an `UCM Subordinate`.	
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 `UCM Subordinate`.
	GetId	Get the `UCM Subordinate` Instance Identifier.

△

	GetSwClusterChangeInfo	Get a list of pending changes to the set of `Software Cluster`s on the Adaptive Platform. The returned list includes all `Software Cluster`s that are to be added, updated or removed. The list of changes is extended in the course of processing `Software Package`s.
	GetSwClusterDescription	Get the general information of the `Software Cluster`s present in the platform.
	GetSwClusterInfo	Get a list of `Software Cluster`s that are in state `k Present`.
	GetSwPackages	Get the `Software Package`s that available in `UCM Subordinate`.
	GetSwProcessProgress	Get the progress of the currently processed `Software Package`.
	ProcessSwPackage	Process a previously transferred `Software Package`.
	RevertProcessedSwPackages	Revert the changes done by processing (by calling ProcessSwPackage()) of one or several `Software Package`s.
	Rollback	Rollback the system to the state before the packages were processed.
	TransferData	Block-wise transfer of a `Software Package` to `UCM Subordinate`.
	TransferExit	Finish the transfer of a `Software Package` to `UCM Subordinate`.
	TransferStart	Start the transfer of a `Software Package`.
Fields:	CurrentStatus	The current status of the `UCM Subordinate`.

9.7.1.1.2 UCM Master

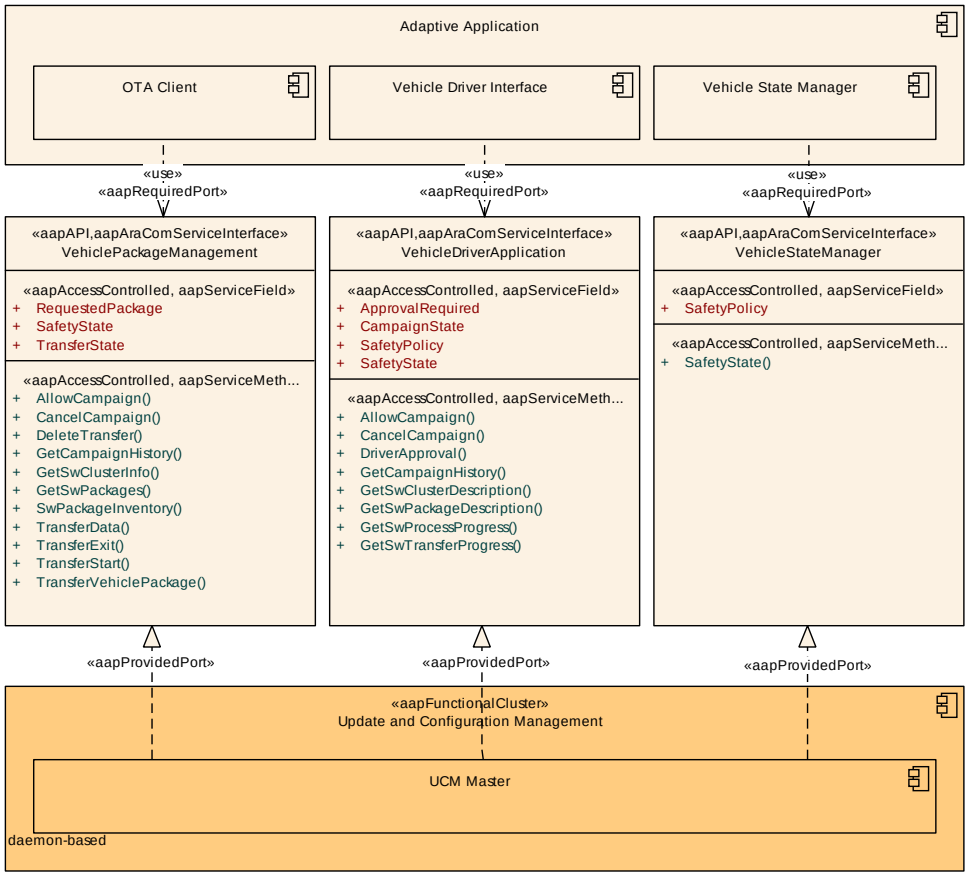

Figure 9.75: Interfaces of UCM Master

Name:	VehiclePackageManagement	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface provides functionality for managing and transferring `Vehicle Package`s and `Software Package`s to `UCM Master`.	
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 `UCM Master`.
	GetSwClusterInfo	Get a list of `SoftwareCluster`s that are in state `k Present`.
	GetSwPackages	Get the `Software Package`s that are part of current campaign handled by `UCM Master`.
	SwPackageInventory	Performs an inventory of all `Software Package`s.
	TransferData	Block-wise transfer of a `Software Package` or `Vehicle Package` to `UCM Master`.

△

	TransferExit	Finish the transfer of a `Software Package` or `Vehicle Package` to `UCM Master`.
	TransferStart	Start the transfer of a `Software Package` to `UCM Master`.
	TransferVehiclePackage	Start the transfer of a `Vehicle Package` to `UCM Master`.
Fields:	RequestedPackage	`Software Package` to be transferred to `UCM Master`.
	SafetyState	Vehicle state computed by the `Vehicle State Manager` Adaptive Application.
	TransferState	The current status of a campaign from an `OTA Client` perspective.

Name:	VehicleDriverApplication	
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.
	CancelCampaign	Aborts an ongoing campaign processing of a `Vehicle Package`.
	DriverApproval	Inform `UCM Master` of the driver's notification resolution (approve or reject).
	GetCampaignHistory	Retrieve all actions that have been performed by `UCM Master`.
	GetSwClusterDescription	Get the general information of the `Software Cluster`s present in the Adaptive Platform.
	GetSwPackageDescription	Get the general information of the `Software Package`s that are part of current campaign handled by `UCM Master`.
	GetSwProcessProgress	Get the progress of the current package processing.
	GetSwTransferProgress	Get the progress of the current package transfer.
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 current status of campaign.
	SafetyPolicy	Safety policy from the `Vehicle Package` to be computed by the `Vehicle State Manager` Adaptive Application.
	SafetyState	Vehicle state computed by the `Vehicle State Manager` Adaptive Application.

Name:	VehicleStateManager	
Technology:	ara::com service interface	
Usage:	Public API	
Description:	This interface provides functionality for a `Vehicle State Manager` Adaptive Application to inform `UCM Master` about the safety state and policy of the vehicle.	
Operations:	SafetyState	Called by the `Vehicle State Manager` Adaptive Application when safety state is changed.
Fields:	SafetyPolicy	Safety policy from the `Vehicle Package` to be computed by the `Vehicle State Manager` Adaptive Application.

9.7.1.1.3 D-PDU API

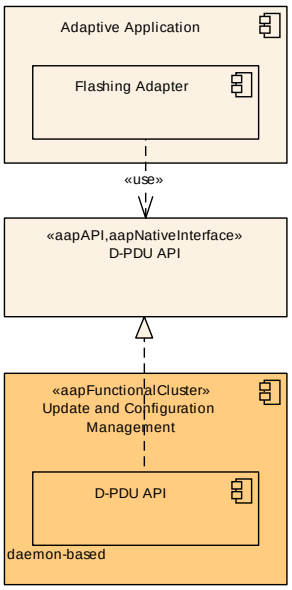

Figure 9.76: Interfaces of UCM Master

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

`Update and Configuration Management` does not provide any interfaces to other `Functional Cluster`s.

9.7.1.3 Required interfaces

Figures 9.77 and 9.78 show the interfaces that are required by `Update and Configuration Management`. These interface are thus required by both `UCM Subordinate` and `UCM Master`.

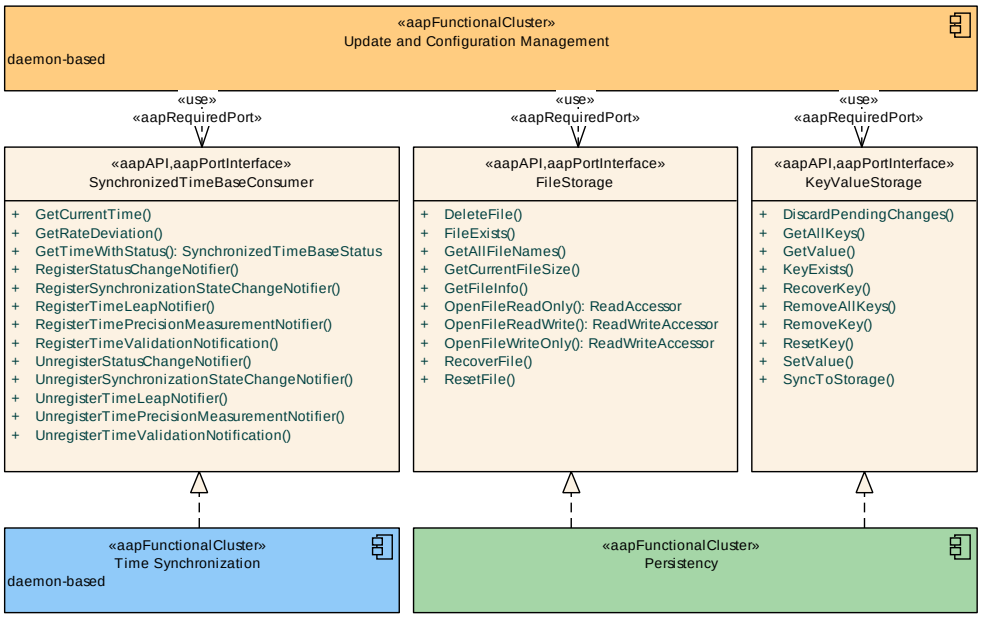

Figure 9.77: Interfaces required by Update and Configuration Management (part 1)

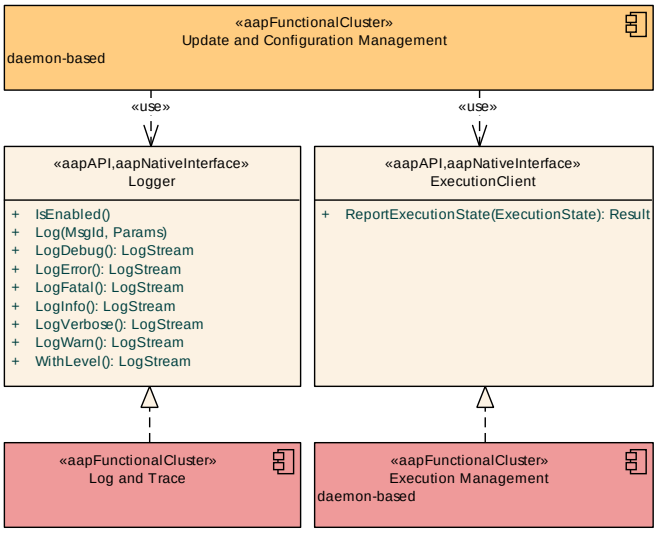

Figure 9.78: Interfaces required by Update and Configuration Management (part 2)

Figure 9.79 shows the interfaces that are only required by `UCM Subordinate`. Correspondingly, Figure 9.80 shows the interfaces that are only required by `UCM Master`.

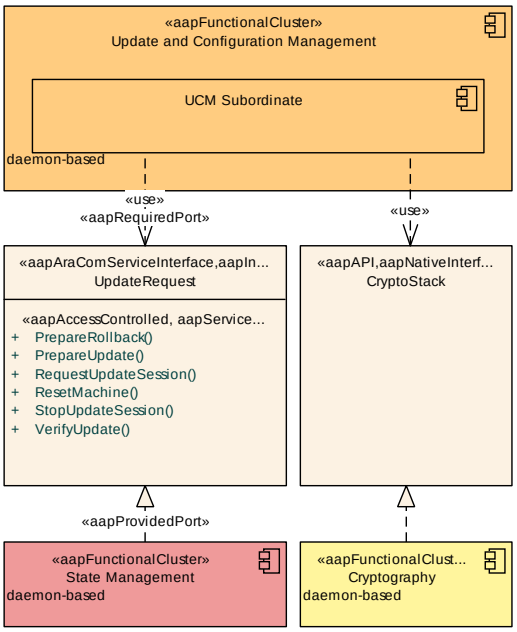

Figure 9.79: Interfaces required by UCM Subordinate

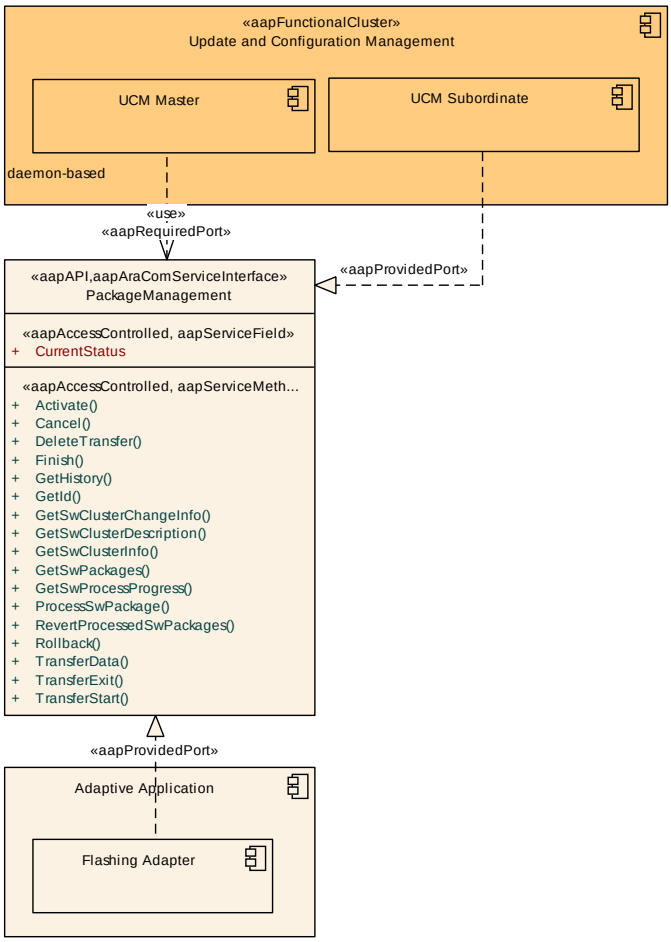

Figure 9.80: Interfaces required by UCM Master

Interface	*Purpose*
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	This interface should be used to store files, for example downloaded packages.
Persistency::KeyValueStorage	This interface should be used to store internal state of `Update and Configuration Management`.
Persistency::ReadAccessor	This interface should be used to store files, for example downloaded packages.
Persistency::ReadWriteAccessor	This interface should be used to store files, for example downloaded packages.
Platform Health Management::SupervisedEntity	This interface should be used to supervise the daemon process(es) of `Update and Configuration Management`.
Registry::ManifestAccessor	`Update and Configuration Management` shall use this interface to read information about its configuration from the `Manifest`s.
Time Synchronization::SynchronizedTimeBase Consumer	`Update and Configuration Management` shall use this interface to get latest timestamp.

Table 9.25: Interfaces required by Update and Configuration Management

9.7.2 Registry

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

9.7.2.1 Defined interfaces

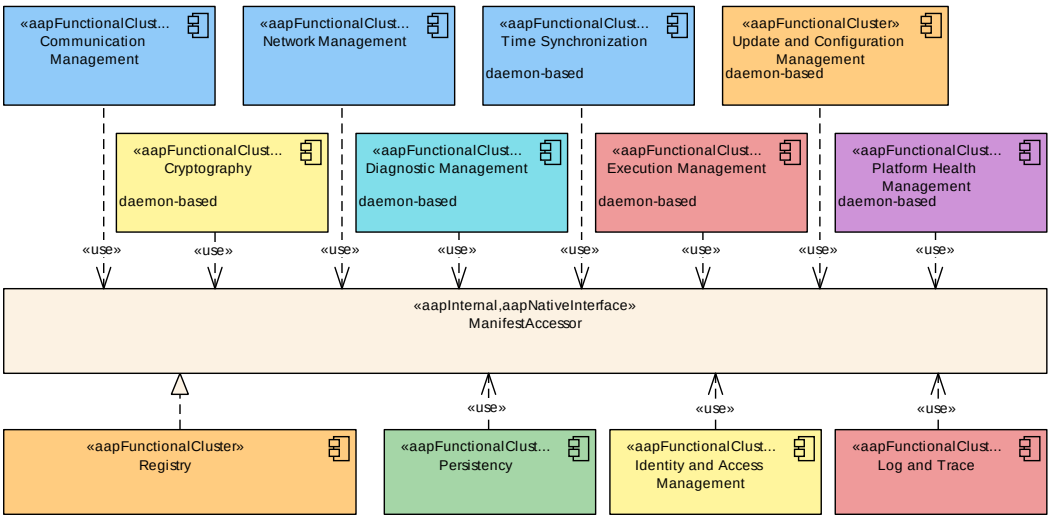

Figure 9.81: Interfaces of Registry

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.

9.7.2.2 Provided interfaces

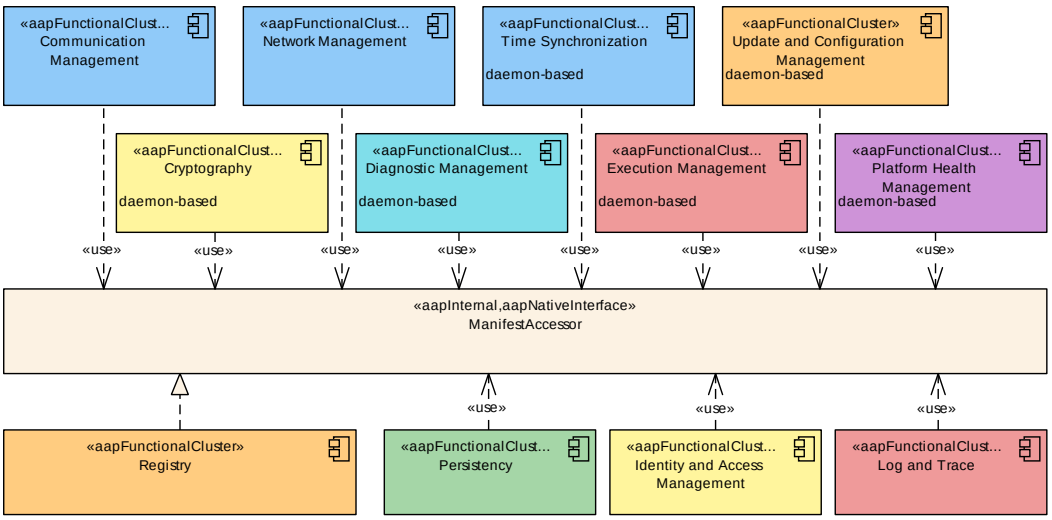

Figure 9.82: Users of the Registry interfaces

Interface	*Requiring functional clusters*
Registry::ManifestAccessor	Communication Management
	Cryptography
	Diagnostic Management
	Execution Management
	Identity and Access Management
	Log and Trace
	Network Management
	Persistency
	Platform Health Management
	Time Synchronization
	Update and Configuration Management

Table 9.26: Interfaces provided by Registry to other Functional Clusters

9.7.2.3 Required interfaces

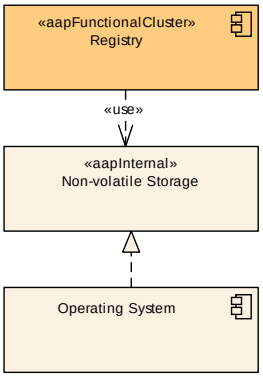

Figure 9.83: Interfaces required by Registry

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

Table 9.27: Interfaces required by Registry

9.8 Diagnostics

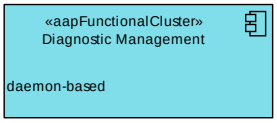

Figure 9.84: Functional Clusters in category Diagnostics

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 `Process`es 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 Client`s 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

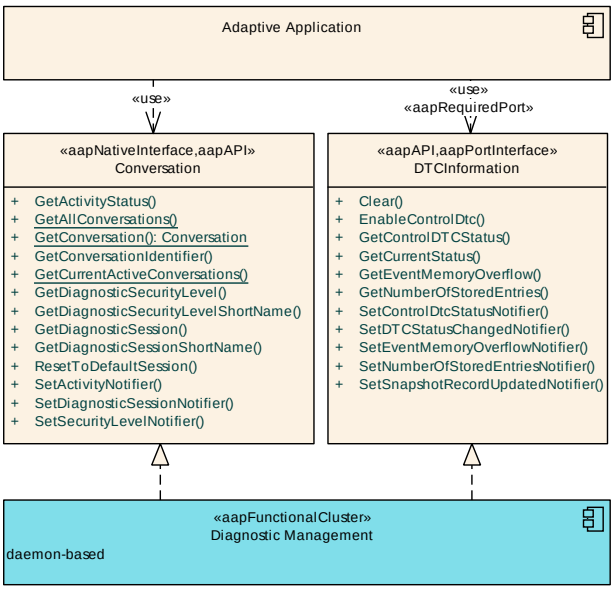

Figure 9.85: Common interfaces of Diagnostic Management (1 of 2)

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.

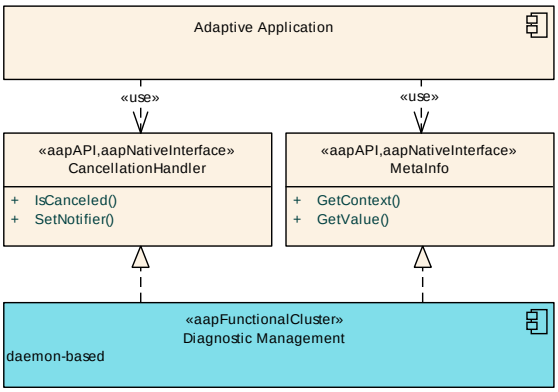

Figure 9.86: Common interfaces of Diagnostic Management (2 of 2)

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.

9.8.1.1.2 Interfaces for request handling

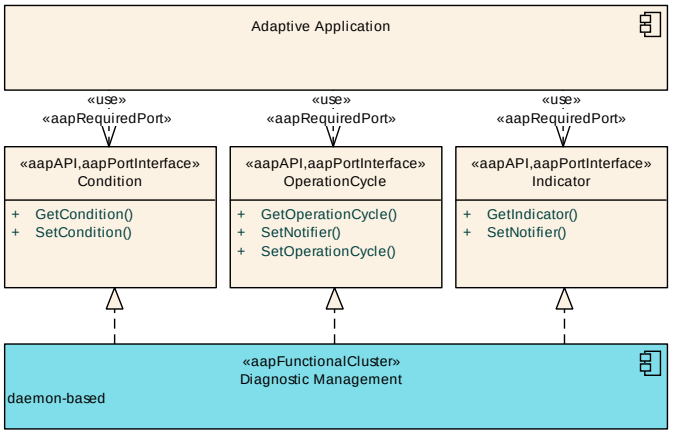

Figure 9.87: Interfaces for request handling

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.

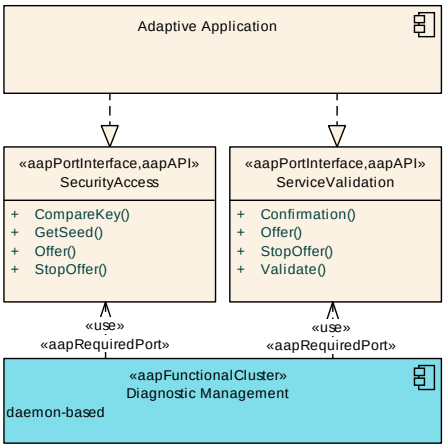

Figure 9.88: Interfaces for request handling

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.

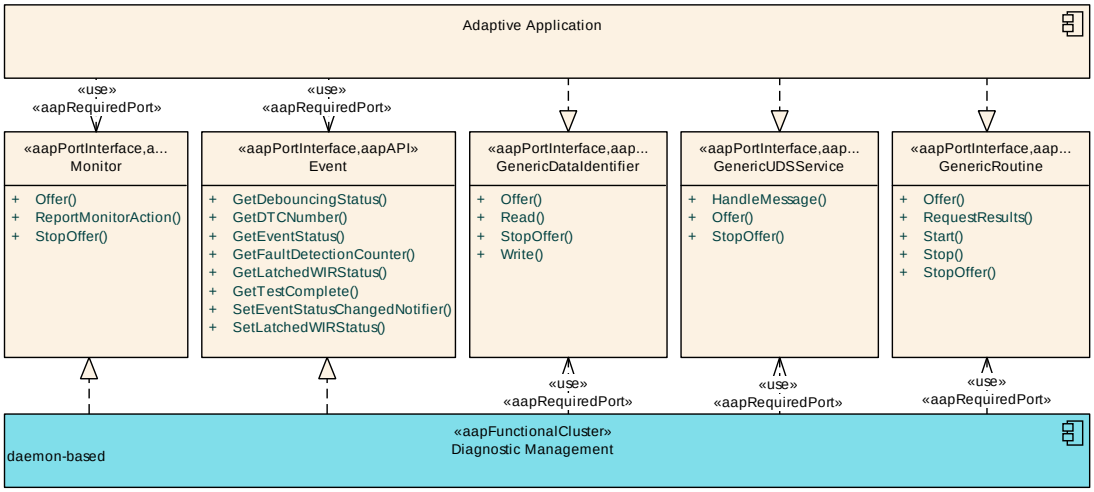

Figure 9.89: Interfaces for generic request handling

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 `Diagnostic RoutineIdentifier`.
	Start	Called for `RoutineControl` with `SubFunction Start` request for this `DiagnosticRoutine Identifier`.
	Stop	Called for `RoutineControl` with `SubFunction Stop` request for this `DiagnosticRoutineIdentifier`.
	StopOffer	Disable forwarding of request messages from `Diagnostic Management`.

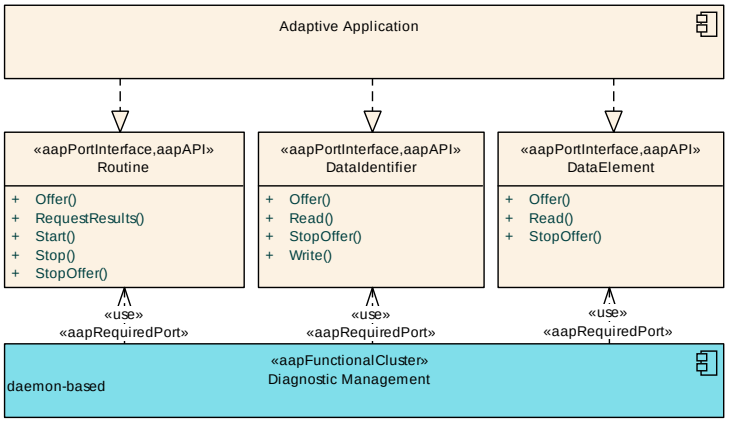

Figure 9.90: Generated interfaces for request handling

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 `Diagnostic RoutineIdentifier`.
	Start	Called for `RoutineControl` with `SubFunction Start` request for this `DiagnosticRoutine Identifier`.
	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 Upload and Download

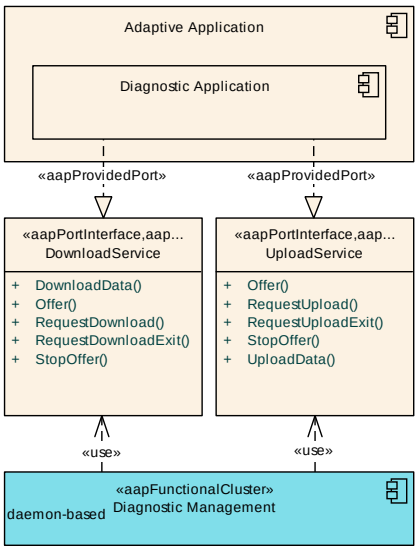

Figure 9.91: Interfaces for Upload and Download

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`.

9.8.1.1.4 Interfaces for State Management

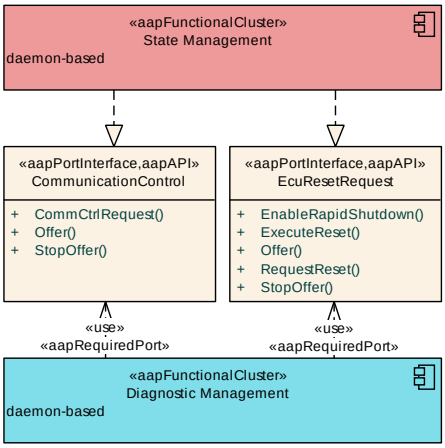

Figure 9.92: Interfaces for State Management

Name:	CommunicationControl	
Technology:	Port interface	
Generated:	No	
Meta-model interface type:	`DiagnosticComControl`	
Usage:	Public API	
Description:	This interface provides functionality for `CommunicationControl`.	
Operations:	CommCtrlRequest	Called for `CommunicationControl` (0x28) with any subfunction as subfunction value is part of argument list.
	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	StopOffer	Disable forwarding of request messages from `Diagnostic Management`.

Name:	EcuResetRequest	
Technology:	Port interface	
Generated:	Yes	
Meta-model interface type:	`DiagnosticEcuResetInterface`	
Usage:	Public API	
Description:	This interface provides functionality for `EcuReset` requests.	
Operations:	*EnableRapidShutdown*	Called for subfunction En-/DisableRapidShutdown.
	ExecuteReset	Execute the requested reset.

▽

△

	Offer	Enable forwarding of request messages from `Diagnostic Management`.
	RequestReset	Called for any EcuReset subfunction, except En-/DisableRapidShutdown.
	StopOffer	Disable forwarding of request messages from `Diagnostic Management`.

9.8.1.1.5 Interfaces for UDS Transportlayer API

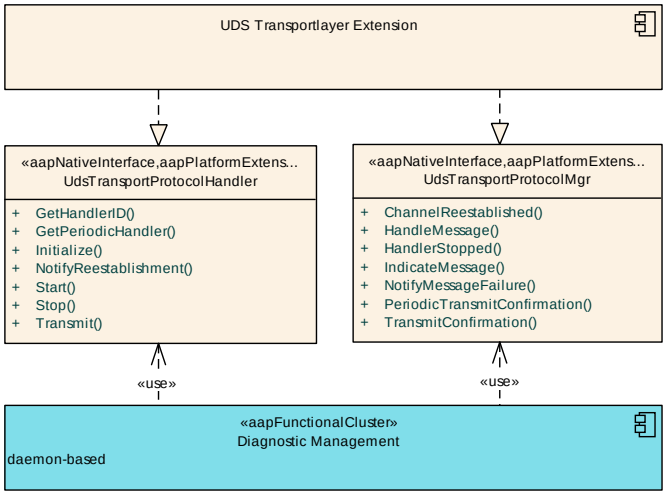

Figure 9.93: Interfaces for the UDS Transportlayer API (1 of 2)

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()

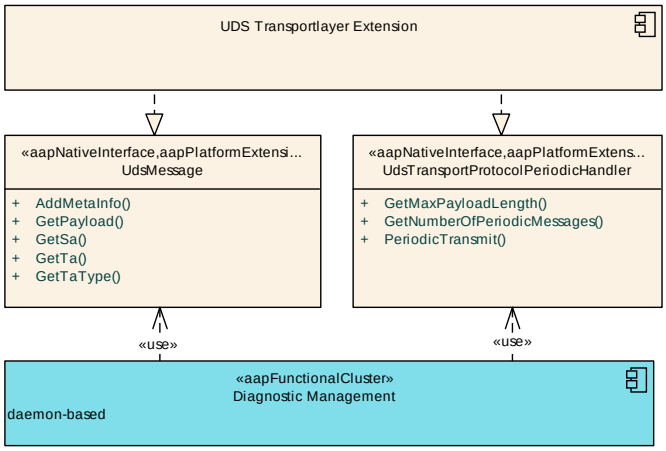

Figure 9.94: Interfaces for the UDS Transportlayer API (2 of 2)

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.

9.8.1.1.6 Interfaces for DoIP API

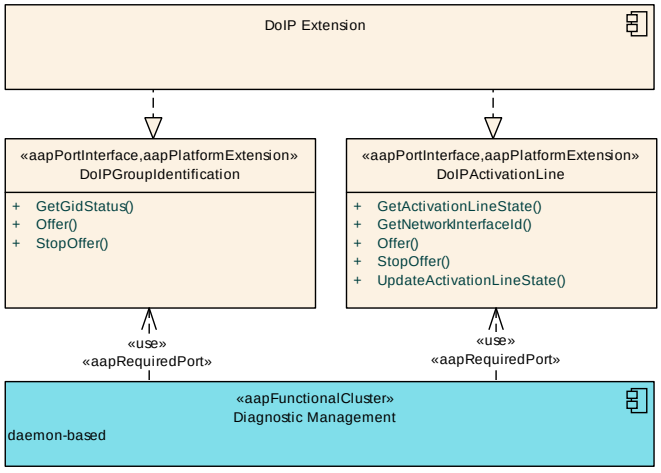

Figure 9.95: Interfaces for the DoIP API (1 of 2)

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:	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.

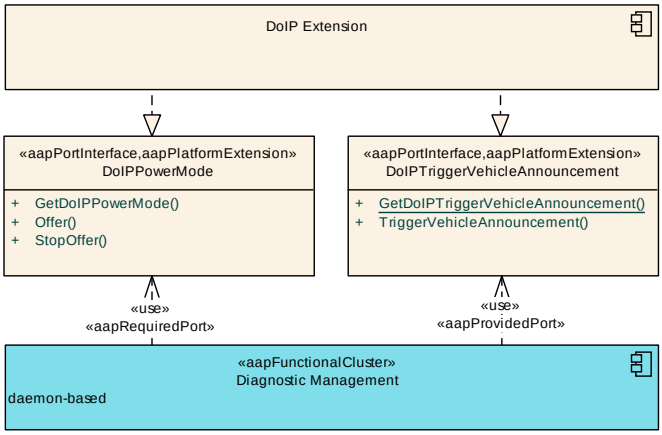

Figure 9.96: Interfaces for the DoIP API (2 of 2)

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.

9.8.1.2 Provided interfaces

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

9.8.1.3 Required interfaces

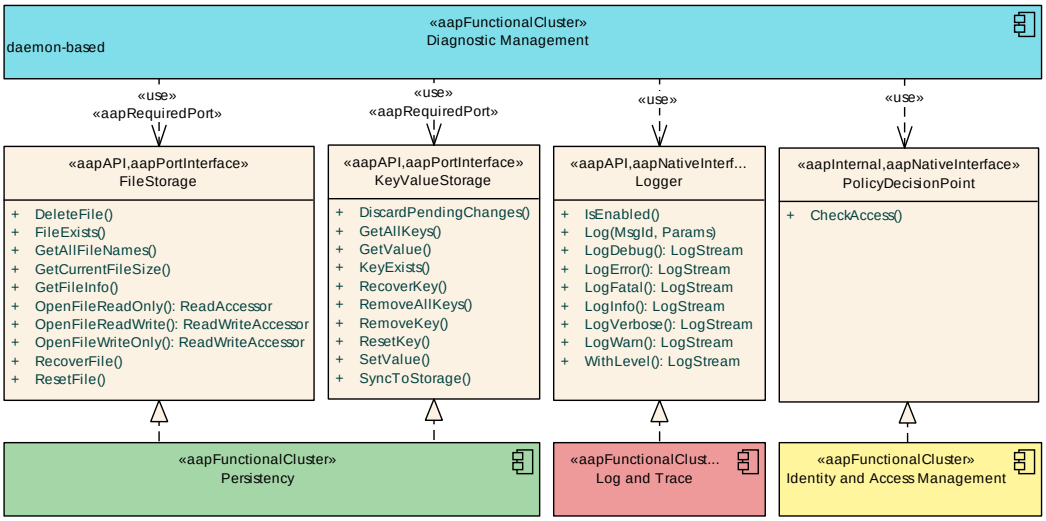

Figure 9.97: Interfaces required by Diagnostic Management

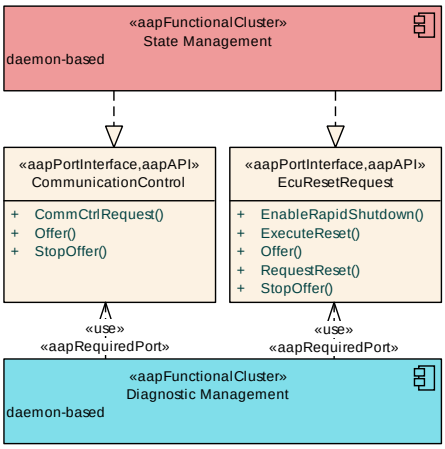

Figure 9.98: Interfaces required by Diagnostic Management from State Management

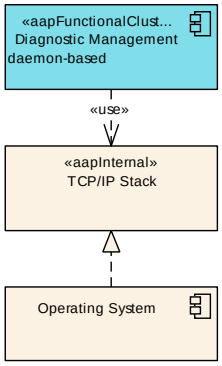

Figure 9.99: Interfaces required by Diagnostic Management from external components

Interface	*Purpose*
TCP/IP Stack	This interface is used for DoIP Transport Protocols.
Diagnostic Management::Communication Control	This interface should be used to realize UDS Service 0x28 - CommunicationControl.
Diagnostic Management::DataElement	This interface is used to handle read `DataElement` requests.
Diagnostic Management::DataIdentifier	This interface is used to handle `ReadDataByIdentifier` and `WriteDataByIdentifier` requests.
Diagnostic Management::DoIPActivationLine	This interface is used to control a DoIP Transport Layer implementation.
Diagnostic Management::DoIPGroup Identification	This interface is used to control a DoIP Transport Layer implementation.
Diagnostic Management::DoIPPowerMode	This interface is used to control a DoIP Transport Layer implementation.
Diagnostic Management::DoIPTriggerVehicle Announcement	This interface is used to control a DoIP Transport Layer implementation.
Diagnostic Management::DownloadService	This interface is used to handle download requests.
Diagnostic Management::EcuResetRequest	This interface is used to handle reset requests.
Diagnostic Management::GenericDataIdentifier	This interface is used to handle `ReadDataByIdentifier` and `WriteDataByIdentifier` requests.
Diagnostic Management::GenericRoutine	This interface is used to handle `RoutineControl` requests.
Diagnostic Management::GenericUDSService	This interface is used to handle UDS requests.
Diagnostic Management::Routine	This interface is used to handle `RoutineControl` requests.
Diagnostic Management::SecurityAccess	This interface is used to handle `SecurityAccess` requests.

△

Interface	*Purpose*
Diagnostic Management::ServiceValidation	This interface is used to handle `ServiceValidation` requests.
Diagnostic Management::UdsMessage	This interface is used to access an UDS Transport Layer implementation.
Diagnostic Management::UdsTransportProtocol Handler	This interface is used to access an UDS Transport Layer implementation.
Diagnostic Management::UdsTransportProtocol Mgr	This interface is used to access an UDS Transport Layer implementation.
Diagnostic Management::UdsTransportProtocol PeriodicHandler	This interface is used to access an UDS Transport Layer implementation.
Diagnostic Management::UploadService	This interface is used to handle upload requests.
Execution Management::ExecutionClient	This interface is used to report the status of the `Diagnostic Management` daemon process(es).
Identity and Access Management::Policy DecisionPoint	`Diagnostic Management` shall use this interface to check access from `Diagnostic Client`s.
Log and Trace::Logger	`Diagnostic Management` shall use this interface to log standardized messages.
Persistency::FileStorageOperations	This interface is used to read and write data to files.
Persistency::KeyValueStorageOperations	This interface should be used to persist key-value data.
Persistency::PersistencyHandlingOperations	This interface is used for general persistency handling.
Persistency::FileStorage	This interface is used to read and write data to files.
Persistency::KeyValueStorage	This interface should be used to persist key-value data.
Persistency::ReadAccessor	This interface is used to read and write data to files.
Persistency::ReadWriteAccessor	This interface is used to read and write data to files.
Platform Health Management::SupervisedEntity	This interface is used to supervise the `Diagnostic Management` daemon process(es).
Registry::ManifestAccessor	`Diagnostic Management` shall use this interface to read its configuration information from the `Manifest`s.
State Management::DiagnosticReset	This interface is used to handle diagnostic reset requests.

Table 9.28: 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.

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

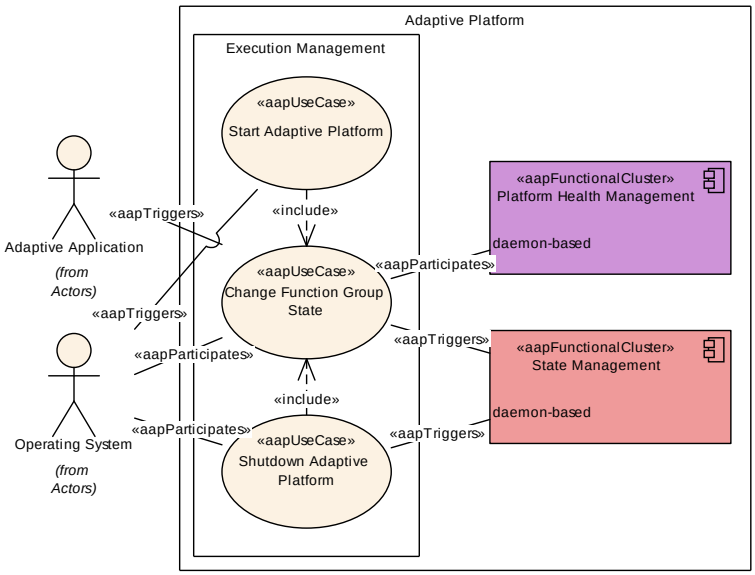

Figure 10.1: Use cases for Execution Management

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 Platform with Supervision of Applications`	This scenario shows 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 Platform with Supervision of Applications`	This scenario shows 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 scenario shows the regular change of a `Function Group State`.

Table 10.4: Use-Case Change Function Group State

10.1.2 State Management

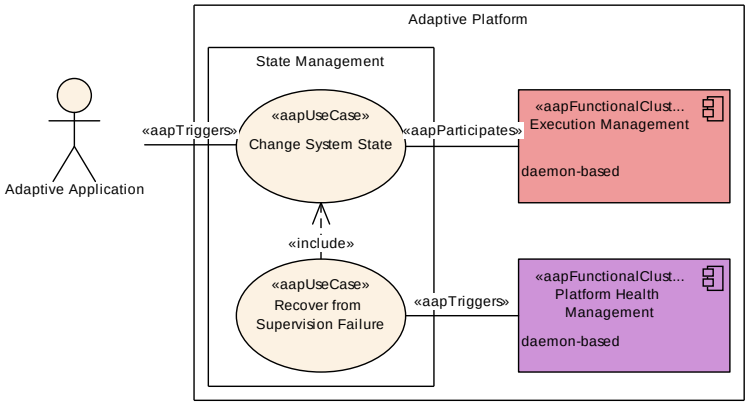

Figure 10.2: Use cases for State Management

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:	Adaptive Application	The `Adaptive Application` triggers this use case by changing the value of the field `Trigger` in a `TriggerIn_{StateGroup}` or `TriggerInOut_{StateGroup}` service interface.
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 `Process`es) has changed.	
Scenarios:	`Accept Trigger Input`	This scenario shows a state change reaction by `State Management` after a change to a `TriggerIn_{StateGroup}` or `TriggerInOut_{StateGroup}` service interface.
	`Reject Trigger Input`	This scenario demonstrates how a change to a `TriggerIn_{StateGroup}` or `TriggerInOut_{StateGroup}` service interface does not lead to a state change reaction by `State Management`.

Table 10.5: Use-Case Change System State

10.1.2.2 Perform Recovery

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.
Included use cases:	`Change System State`	This use case is included to perform recovery by requesting a transition to a new system state.

▽

△

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:	`Successful Recovery`	Default scenario that shows a successful recovery from a supervision failure by switching to a new system state.

Table 10.6: Use-Case Recover from Supervision Failure

10.2 Storage

10.2.1 Persistency

The use cases for `Persistency` are organized into three categories. Section 10.2.1.1 lists the use cases for reading persistent data. Section 10.2.1.2 lists the use cases for storing persistent data. Section 10.2.1.3 lists the use cases related to monitoring of persistent storage.

10.2.1.1 Reading Persistent Data

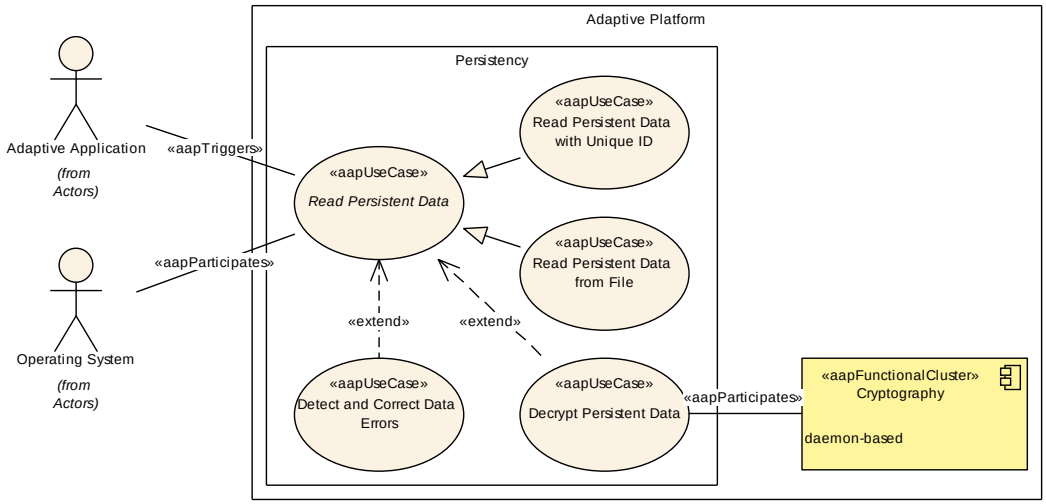

Figure 10.3: Use cases for Reading Persistent Data

10.2.1.1.1 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 Data from File Successfully`	This scenario shows how data is read from a file successfully.

Table 10.7: Use-Case Read Persistent Data from File

10.2.1.1.2 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 Data with Unique ID Successfully	This scenario shows how data is read for a unique ID successfully.

Table 10.8: Use-Case Read Persistent Data With Unique ID

10.2.1.1.3 Decrypt Persistent Data

Name:	Decrypt Persistent Data	
Description:	This use case describes decryption of persistent data.	
Supporting:	Yes	
Participating Actors:	Cryptography	Cryptography participates in this use case by decrypting data.
Preconditions:	None	
Invariants:	None	
Postconditions:	None	

Table 10.9: Use-Case Decrypt Persistent Data

10.2.1.1.4 Detect and Correct Data Errors

Name:	Detect and Correct Data Errors
Description:	This use case describes detection of errors in persistent data and their correction.
Supporting:	Yes
Preconditions:	None
Invariants:	None
Postconditions:	None

Table 10.10: Use-Case Detect and Correct Data Errors

10.2.1.2 Storing Persistent Data

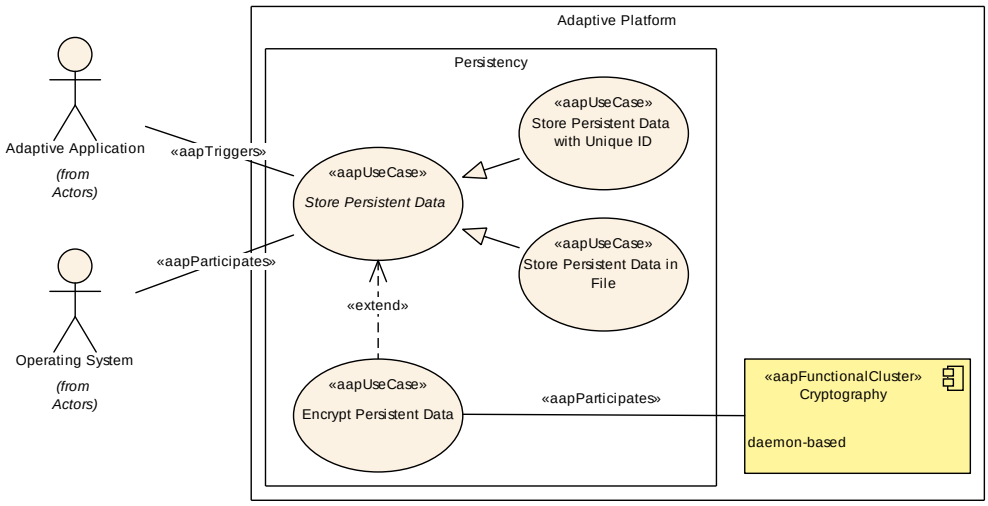

Figure 10.4: Use cases for Storing Persistent Data

10.2.1.2.1 Store Persistent Data in File

Name:	Store Persistent Data in File	
Description:	This use case describes storing persistent data to 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`).
Preconditions:	• The requested file exists.	
Invariants:	None	
Postconditions:	None	
Scenarios:	`Store Data to File Successfully`	This scenario shows how data is successfully stored to a file.

Table 10.11: Use-Case Store Persistent Data in File

10.2.1.2.2 Store Persistent Data with Unique ID

Name:	Store Persistent Data with Unique ID	
Description:	This use case describes storing persistent data associated with 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:	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`).
Preconditions:	• The requested unique ID (key) exists.	
Invariants:	None	
Postconditions:	None	
Scenarios:	`Store Data with Unique ID Sucessfully`	This scenario shows how data is successfully stored with a unique ID (key).

Table 10.12: Use-Case Store Persistent Data with Unique ID

10.2.1.2.3 Encrypt Persistent Data

Name:	Encrypt Persistent Data	
Description:	This use case describes the encryption of persistent data.	
Supporting:	Yes	
Participating Actors:	Cryptography	The `Cryptography` participates in this scenario by encrypting persistent data.
Preconditions:	None	
Invariants:	None	
Postconditions:	None	

Table 10.13: Use-Case Encrypt Persistent Data

10.2.1.3 Monitoring

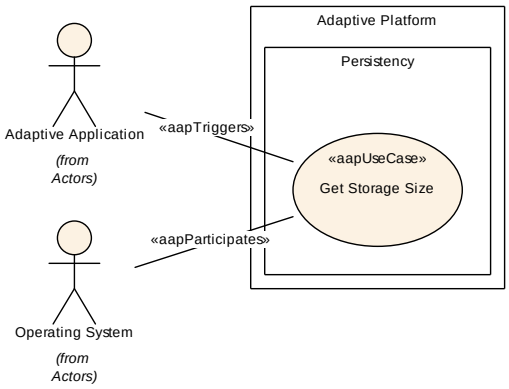

Figure 10.5: Use cases for Monitoring

10.2.1.3.1 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 10.14: Use-Case Get Storage Size

10.3 Security

10.3.1 Firewall

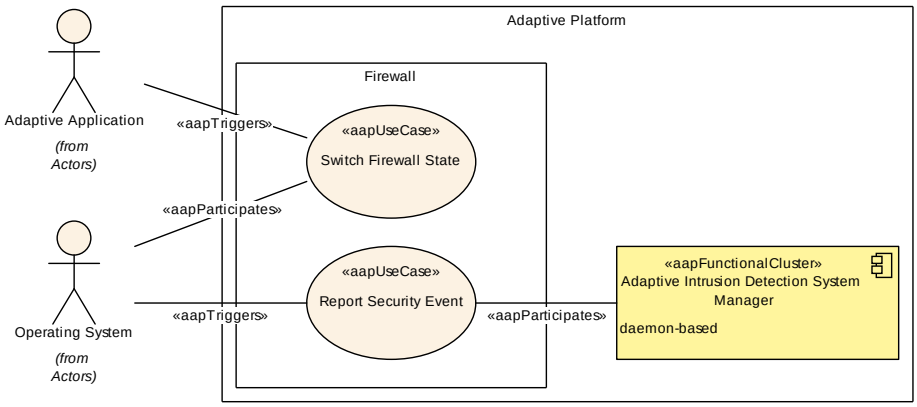

Figure 10.6: Use cases for Firewall

10.3.1.1 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 Successfully`	This scenario shows the successful switch of the firewall state.

Table 10.15: Use-Case Switch Firewall State

10.3.1.2 Report Security Event

Name:	Report Security Event	
Description:	This use case describes reporting a Security Event to `Adaptive 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:	Adaptive Intrusion Detection System Manager	`Adaptive 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 `Adaptive Intrusion Detection System Manager`.	
Scenarios:	`Report Security Event Successfully`	This scenario shows the successful reporting of an security event.

Table 10.16: Use-Case Report Security Event

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.

11.1 Runtime

11.1.1 Execution Management

11.1.1.1 Start Adaptive Platform

11.1.1.1.1 Scenario: Start Platform with Supervision of Applications

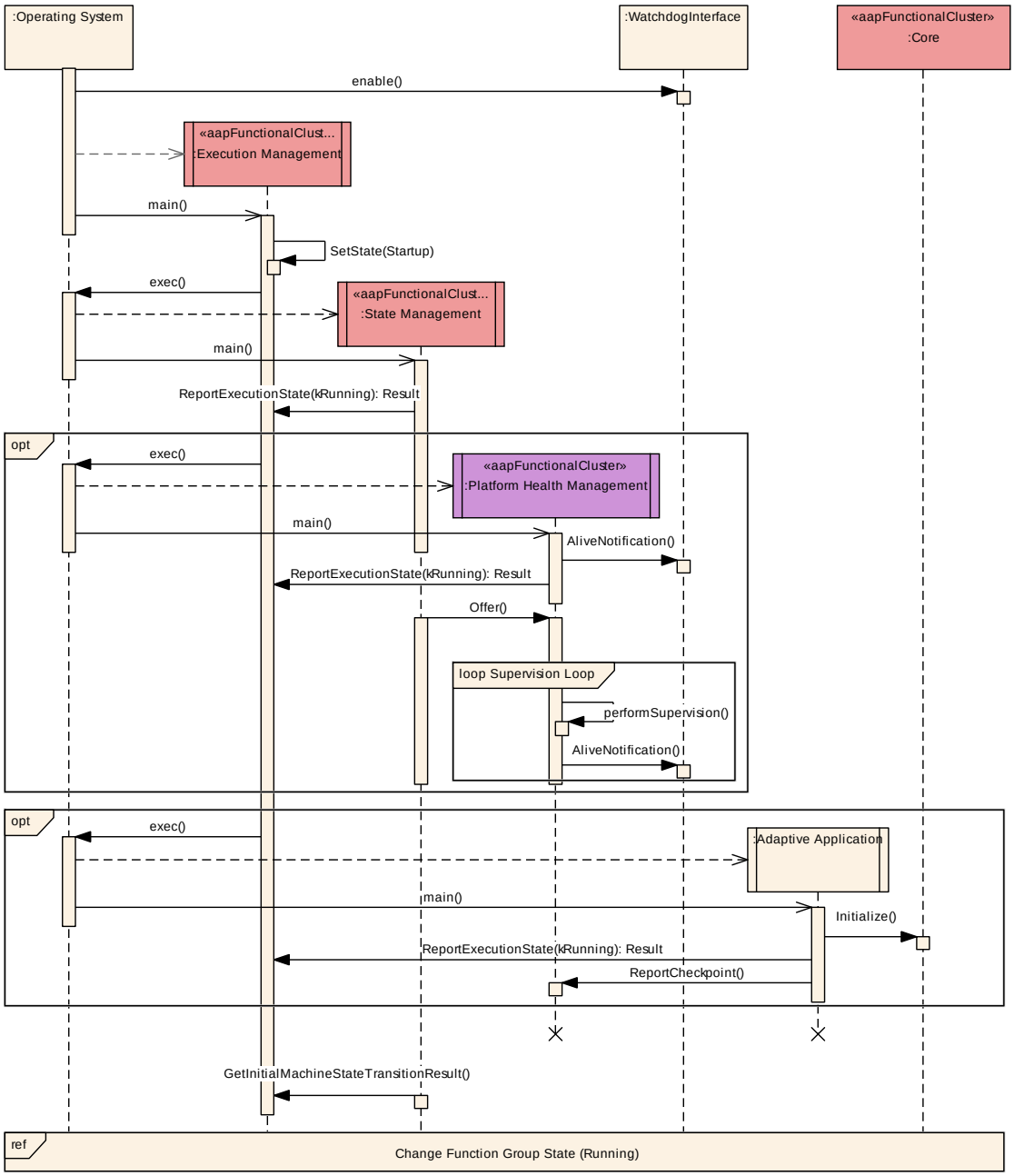

Figure 11.1: Scenario: Start Platform with Supervision of Applications

Figure 11.1 shows a scenario of 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 Platform with Supervision of Applications

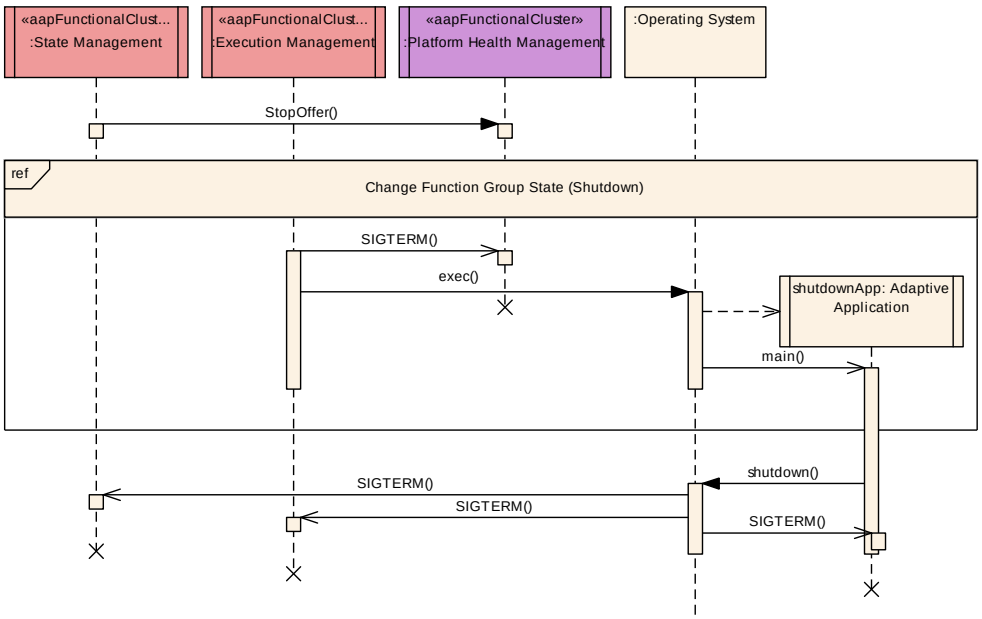

Figure 11.2: Scenario: Shutdown Platform with Supervision of Applications

Figure 11.2 shows a 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 State Management and an application that issues a shutdown request towards the Operating System (shutdownApp in Figure 11.2). Execution Management will therefore 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 (i.e. State Management, Execution Management) of the AUTOSAR Adaptive Platform 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

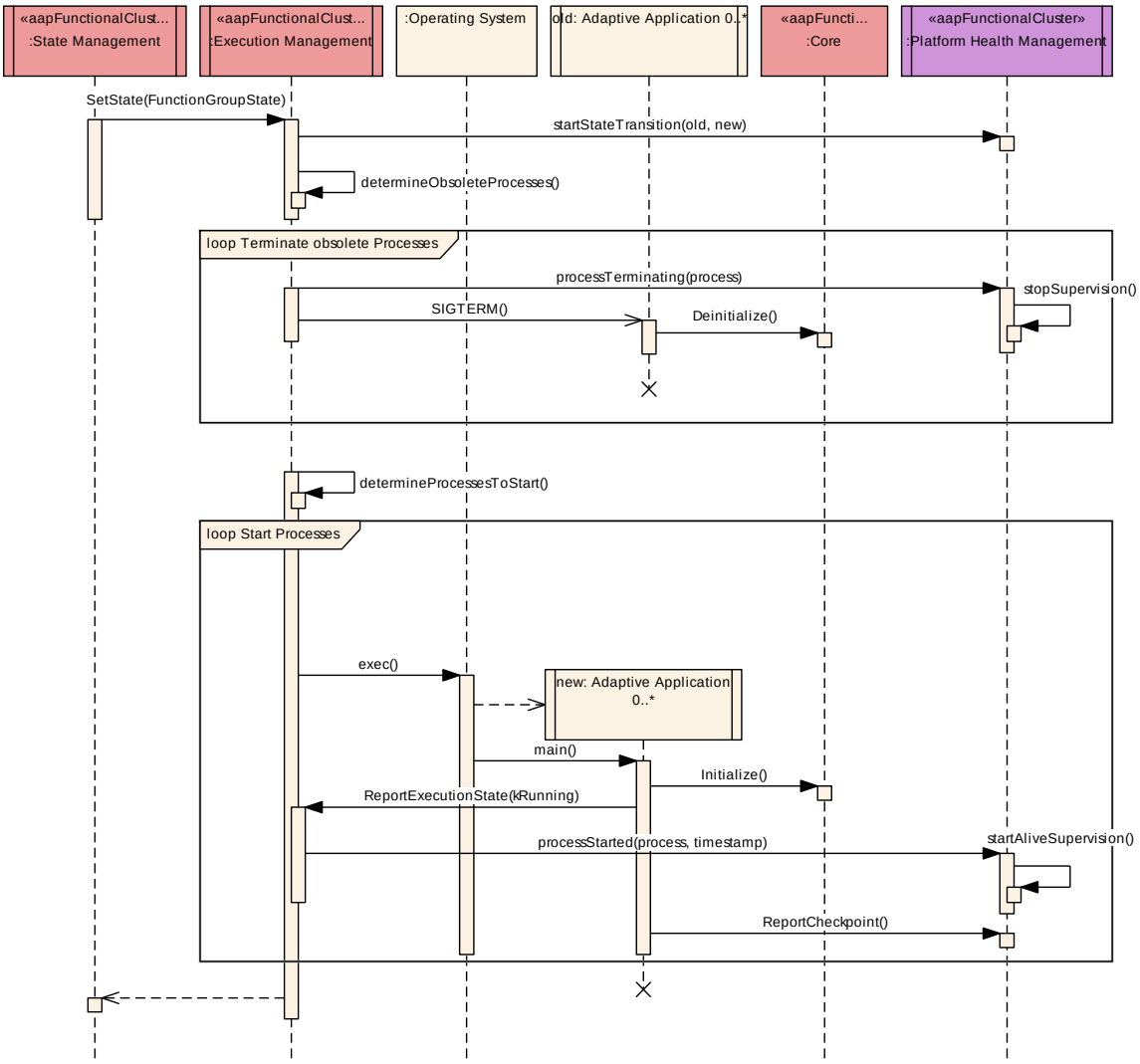

Figure 11.3: Scenario: Regular change of a Function Group State

Figure 11.3 shows a scenario for changing a `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: Accept Trigger Input

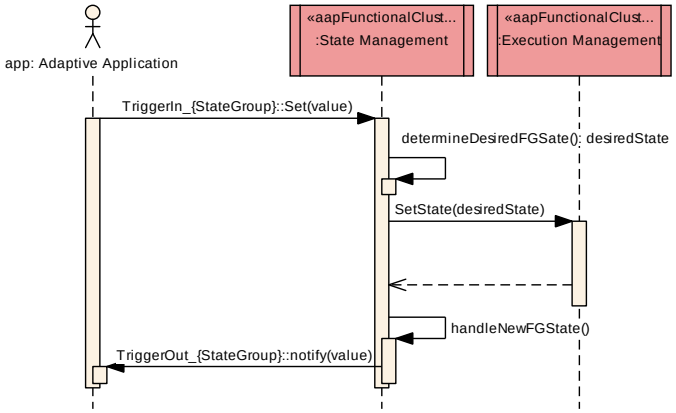

Figure 11.4: Scenario: Accept Trigger Input

Figure 11.4 shows the default scenario for changing the system state. An `Adaptive Application` changes a field in the `TriggerIn_{StateGroup}` service interface. Alternatively, the `TriggerInOut_{StateGroup}` may be used (not shown). Based on the new input data, `State Management` determines a new desired system state, that is a set of desired `Function Group State`s, and requests these `Function Group State`s from `Execution Management` by calling `SetState()`.

After the new system state has been reached, `State Management` updates the fields in the `TriggerOut_{StateGroup}` (and `TriggerInOut_{StateGroup}`, not shown) interfaces accordingly.

11.1.2.1.2 Scenario: Reject Trigger Input

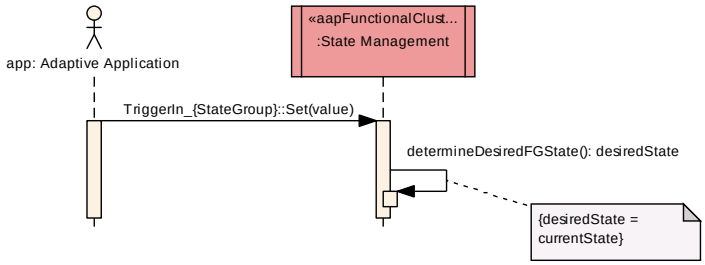

Figure 11.5: Scenario: Reject Trigger Input

Figure 11.5 shows an alternate scenario. An Adaptive Application changes a field in the TriggerIn_{StateGroup} (alternatively TriggerInOut_{StateGroup}) service interface. Despite the new input data, State Management determines that the current system state is still the desired system state. Therefore, no further action is taken by State Management.

11.1.2.2 Recover from Supervision Failure

11.1.2.2.1 Scenario: Successful Recovery

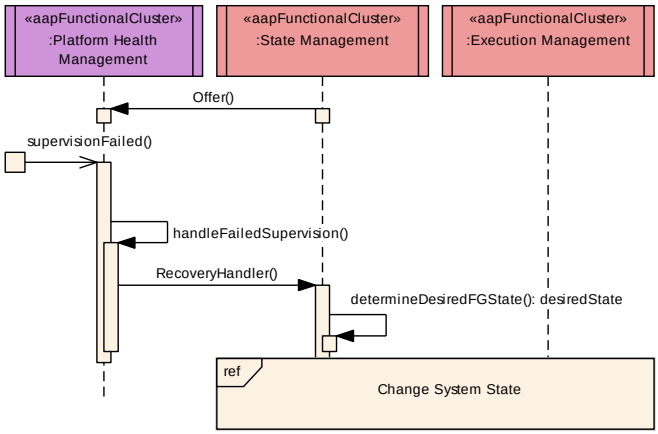

Figure 11.6: Scenario: Successful Recovery

Figure 11.6 shows the scenario for performing successful recovery 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.

11.2 Storage

11.2.1 Persistency

11.2.1.1 Read Persistent Data from File

11.2.1.1.1 Scenario: Read Data from File Successfully

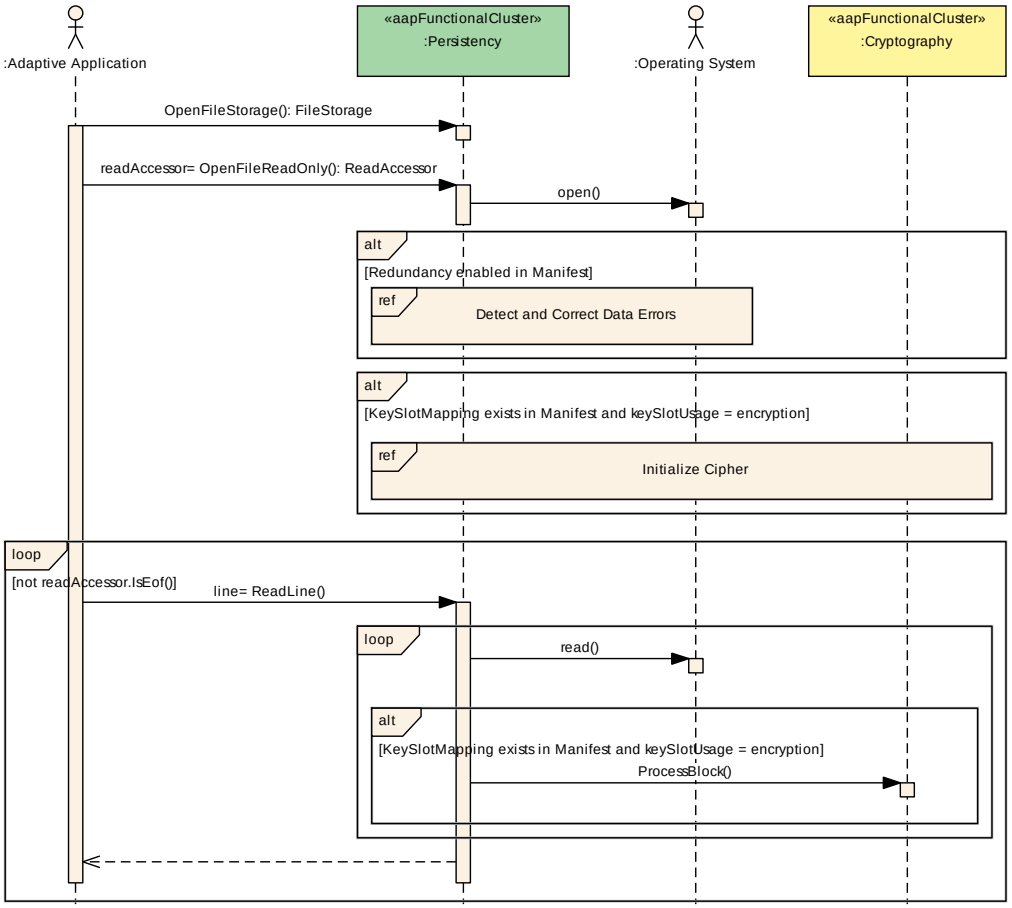

Figure 11.7: Scenario: Read Data from File Successfully

Figure 11.7 shows the scenario for successfully reading data from a file successfully. 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.

11.2.1.2 Read Persistent Data with Unique ID

11.2.1.2.1 Scenario: Read Data with Unique ID Successfully

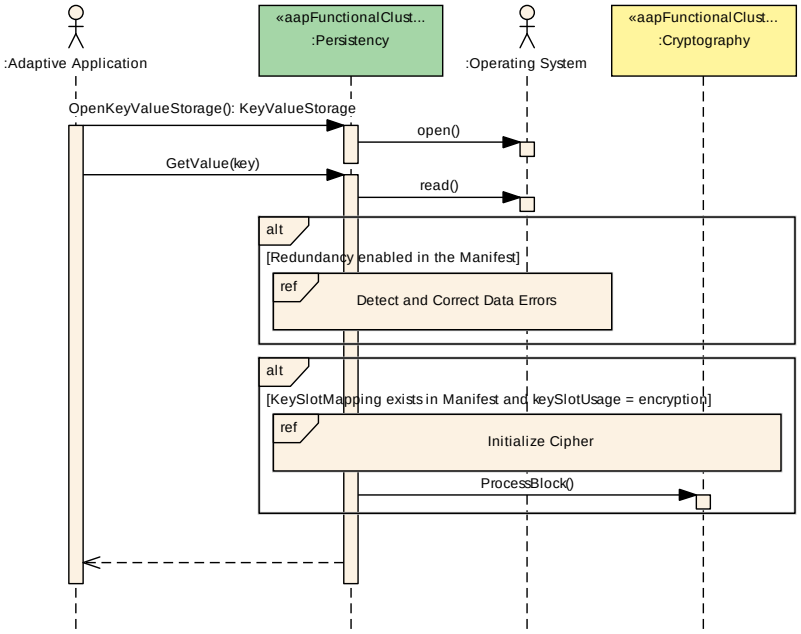

Figure 11.8: Scenario: Read Data with Unique ID Successfully

Figure 11.8 shows the scenario for successfully reading data by providing a unique identifier (key). 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.

11.2.1.3 Store Persistent Data to File

11.2.1.3.1 Scenario: Store Data to File Successfully

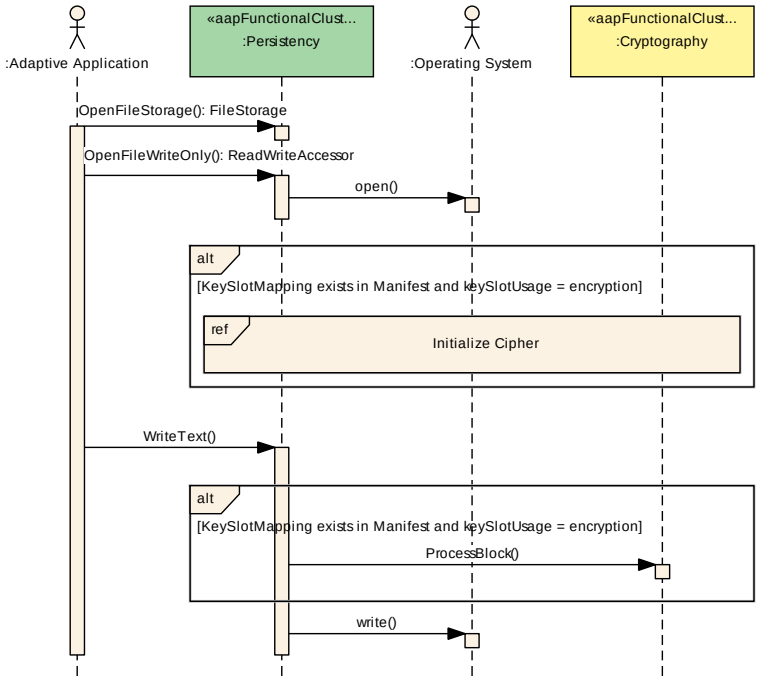

Figure 11.9: Scenario: Store Data to File Successfully

Figure 11.9 shows a scenario for storing data to a file successfully. 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 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.

11.2.1.4 Store Persistent Data with Unique ID

11.2.1.4.1 Scenario: Store Data with Unique ID Successfully

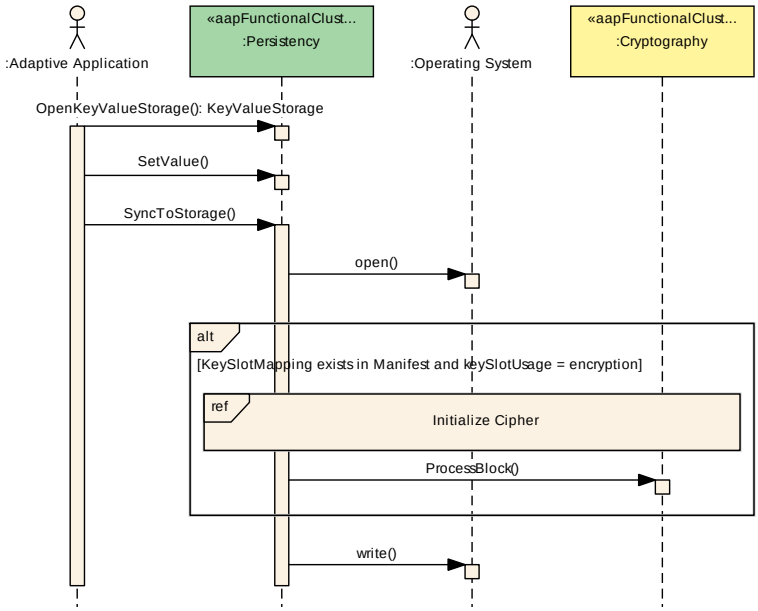

Figure 11.10: Scenario: Store Persistent Data with a Unique ID Successfully

Figure 11.10 shows a scenario for storing data associated with a unique identifier (key) successfully. 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.

11.2.1.5 Get Storage Size

11.2.1.5.1 Scenario: File Store

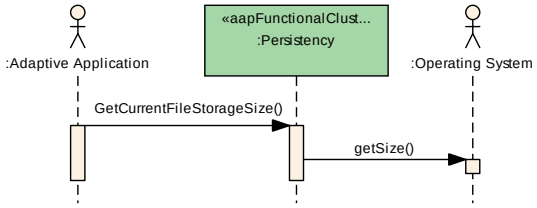

Figure 11.11: Scenario for determining the size of a FileStorage

Figure 11.11 shows the scenario for monitoring the storage space of a FileStorage. The `Adaptive Application` needs to call GetCurrentFileStorageSize() to determine the current size.

11.2.1.5.2 Scenario: Key Value Store

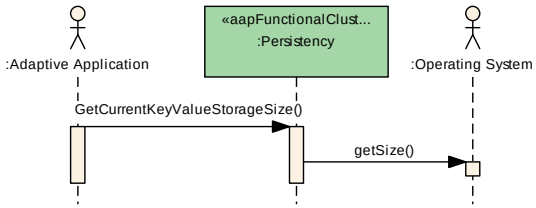

Figure 11.12: Scenario for determining the size of a KeyValueStorage

Figure 11.12 shows the scenario for monitoring the storage space of a KeyValueStorage. The `Adaptive Application` needs to call GetCurrentKeyValueStorageSize() to determine the current size.

11.3 Security

11.3.1 Firewall

11.3.1.1 Switch Firewall State

11.3.1.1.1 Scenario: Switch Firewall State Successfully

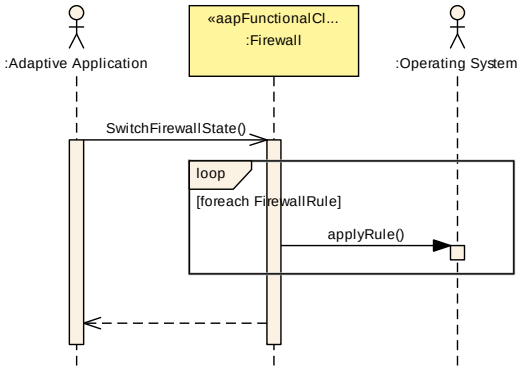

Figure 11.13: Scenario: Switch Firewall State Successfully

Figure 11.13 shows the scenario for successfully switching the state of the `Firewall`. The `Adaptive Application` triggers the state switch by calling SwitchFirewallState(). The `Firewall` will then apply the `FirewallRule`s configured for the request state

in an implementation-specific way (e.g., using tools provided with the TCP/IP stack of the Operating System).

11.3.1.2 Report Security Event

11.3.1.2.1 Scenario: Report Security Event Successfully

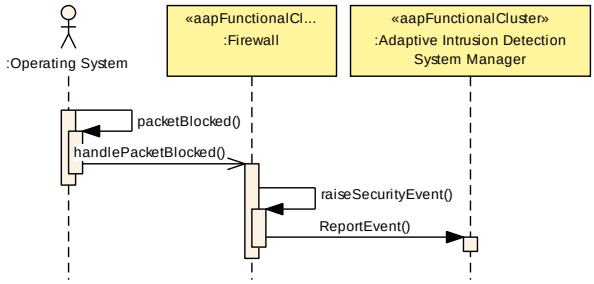

Figure 11.14: Scenario: Report Security Event Successfully

Figure 11.14 shows the scenario for successfully reporting a Security Event to Adaptive Intrusion Detection System Manager. 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 Adaptive Intrusion Detection System Manager by calling ReportEvent(). Adaptive Intrusion Detection System Manager will then handle the Security Event accordingly.

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

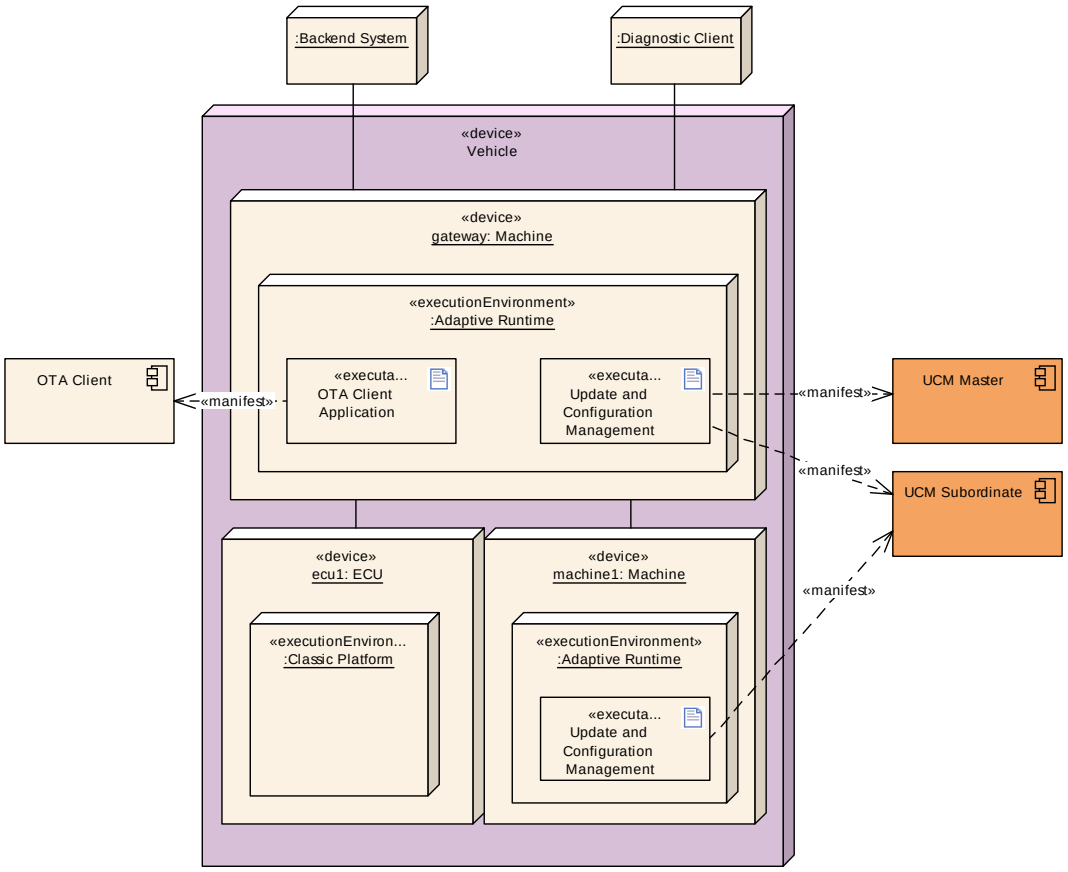

Figure 12.1: Exemplary vehicle software update scenario

`Update and Configuration Management` allows to install and update software on the AUTOSAR Adaptive Platform and AUTOSAR Classic Platform. For the AUTOSAR Adaptive Platform, `Update and Configuration Management` also allows to remove software. The software packages can be received either from a `Diagnostic Client` or from a specific `Backend System` for over-the-air updates. In a vehicle, one `Adaptive Application` takes the role of a master that 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

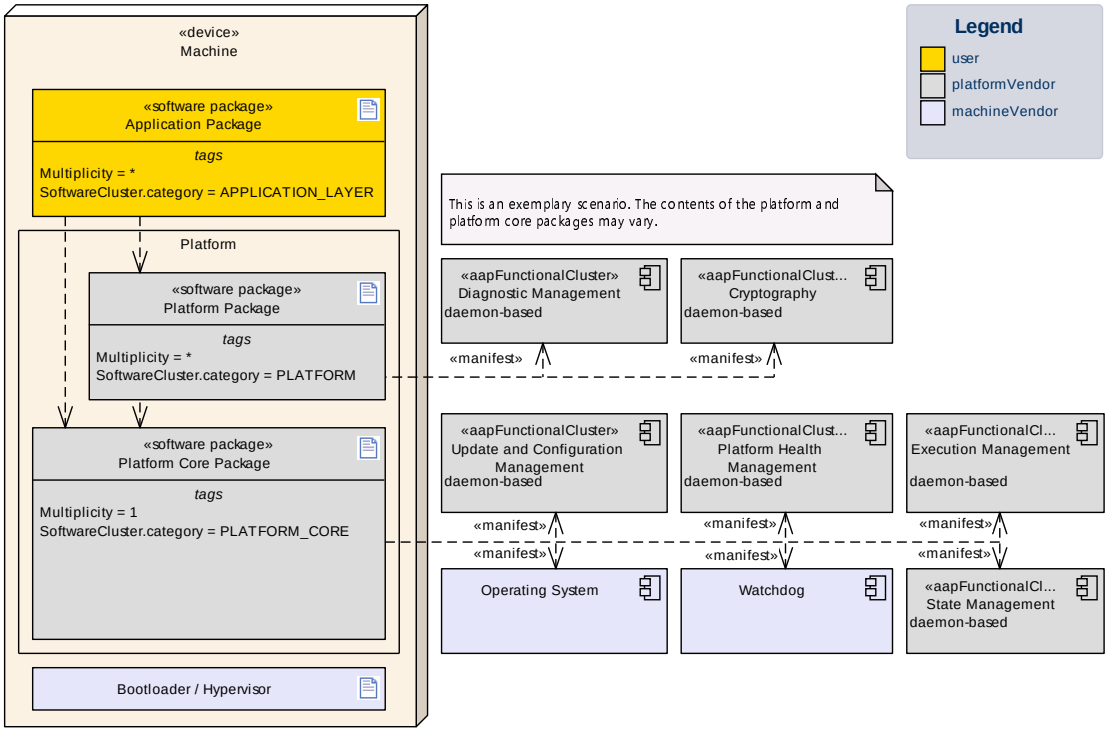

Figure 12.2: Exemplary deployment of Software Packages on a Machine

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, 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 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 Cluster`s depend on. Figure 13.1 provides an overview of these entities, their logical relationships, and the `Functional Cluster`s that depend on them. For the sake of brevity, this overview uses simplifications and abstractions over the actual specifications in the [9, manifest specification].

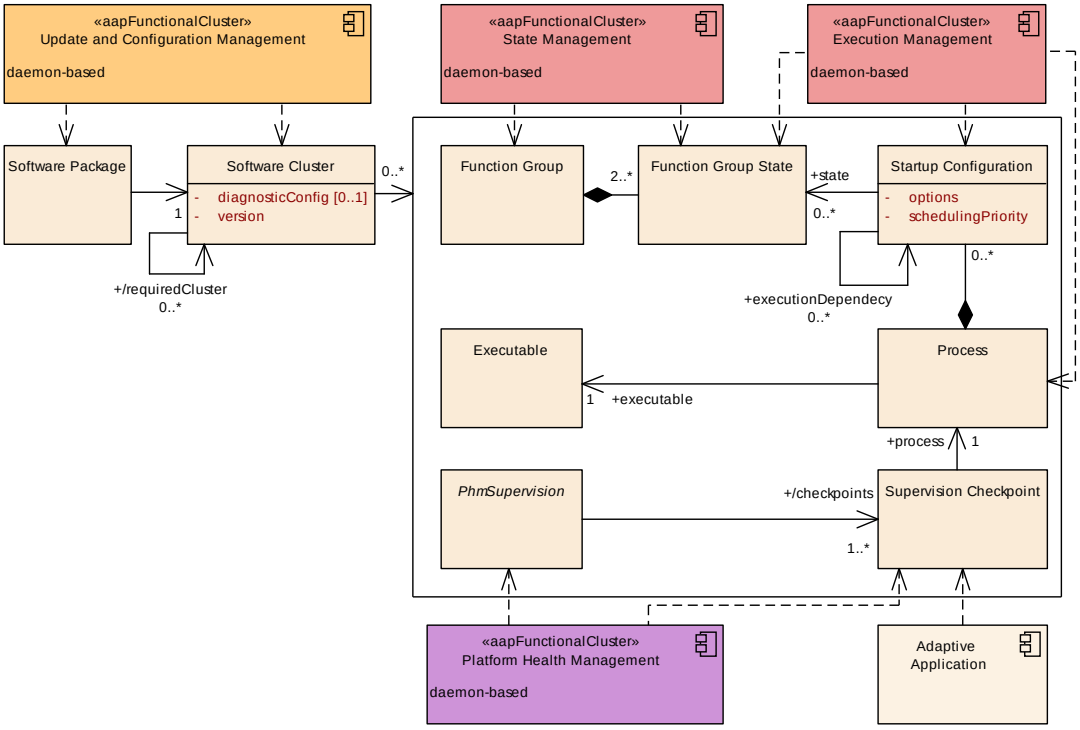

Figure 13.1: Overview of platform entities and their logical relationships

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 `Executable`s (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 `Process`es (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 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 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. A Software Cluster may also specify structural dependencies to Sub Software Clusters in order to build larger installable units. The top of such a structural dependency hierarchy is called a Root Software Cluster. Please note that a Software Cluster is only used to configure deployment aspects. A Software Cluster is not a runtime entity.

A `Root Software Cluster` may specify a diagnostic configuration, in particular, a diagnostic address. In contrast, a `Sub Software Cluster` may depend on a diagnostic configuration of its `Root Software Cluster`. The diagnostic configuration applies to `Process`es that are (transitively) contained in a `Root Software Cluster` and its `Sub Software Cluster`(s). That means, at runtime, any diagnostic event produced by those `Process`es 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 Group`s with `Function Group State`s and associating `StartupConfiguration`s of `Process`es 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 Group`s, do not need to have a unique (simple) name within the overall model because their path is still unique. This allows to design `Software Cluster`s independently, for example, by external suppliers.

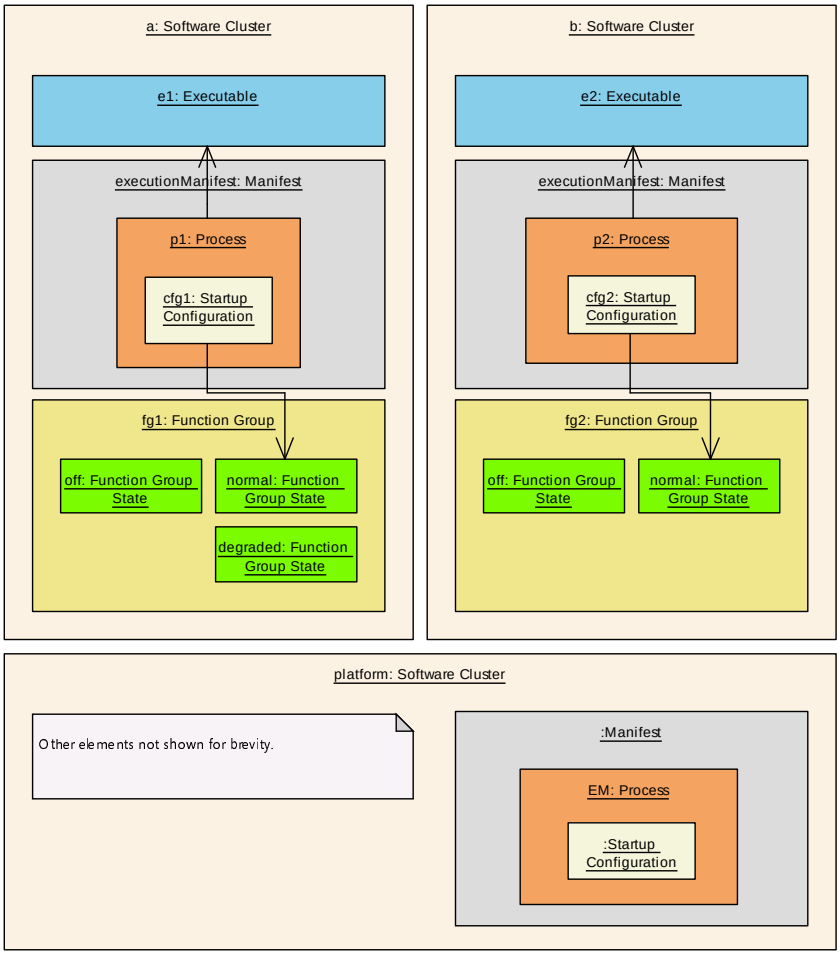

Figure 13.2: Exemplary `Software Cluster` during application design

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).

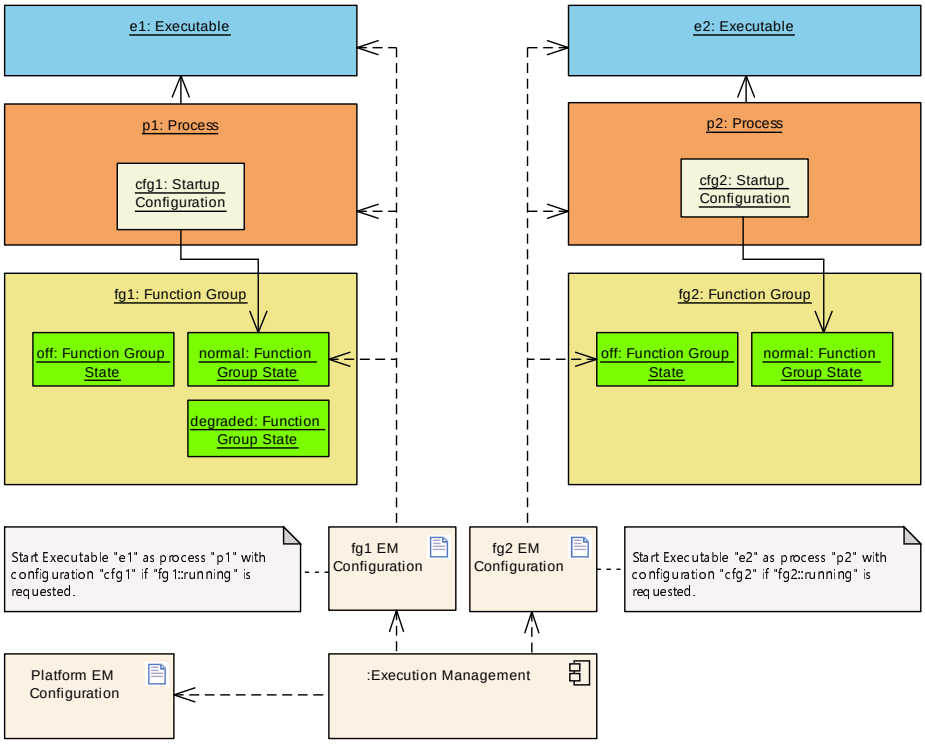

Figure 13.3: Impact of exemplary Software Cluster during runtime

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

A diagnostic configuration, in particular a diagnostic address, may only be assigned to a `Root Software Cluster` in the `Manifest`. Nevertheless, the mapped `DiagnosticContributionSet`(s) may be distributed across several `Software Clusters`. This concept provides a lot of flexibility in assignment of a single diagnostic address to `Software Clusters`. 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.

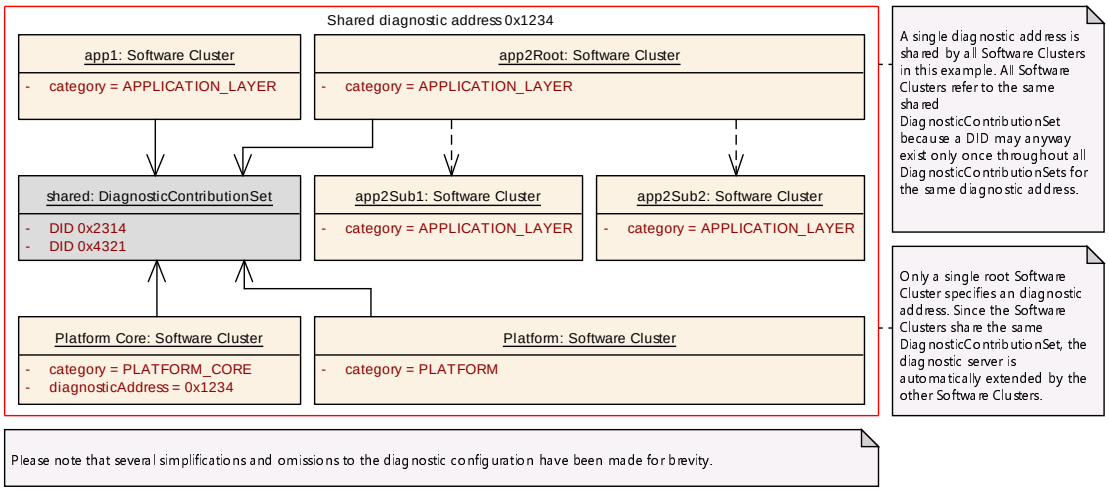

Figure 13.4: Example defining a single diagnostic address for the whole `Machine`

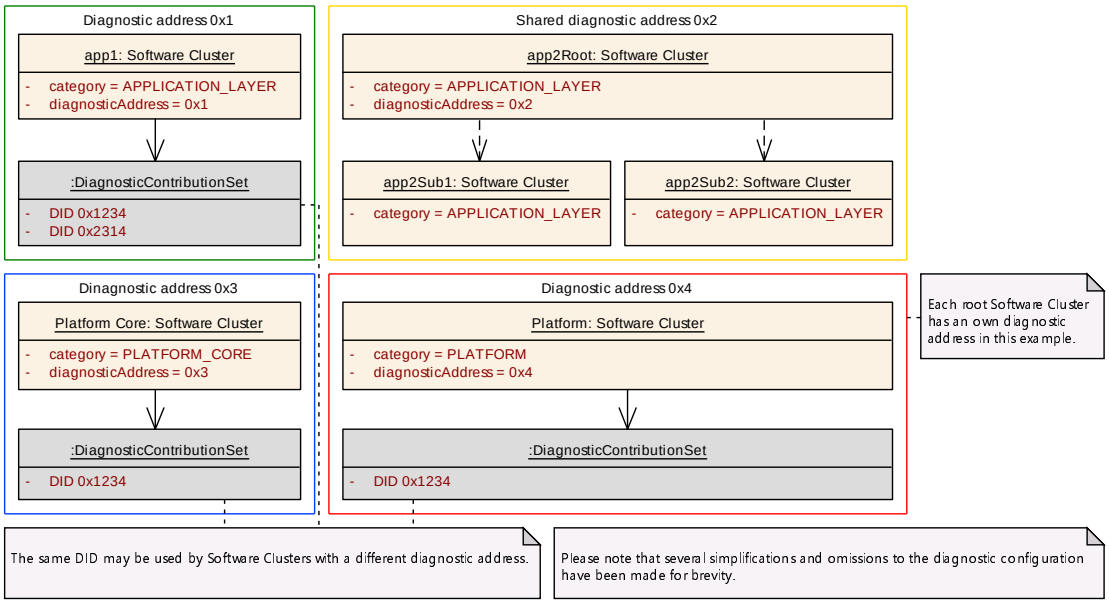

Figure 13.5: Example using one diagnostic address for each `Software Cluster`

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).

It is expected that a Violation or Corruption will not be experienced by a user of the AUTOSAR Adaptive Platform (i.e. an application developer), unless there is something seriously wrong in the overall system. For example, faulty hardware may lead to a Corruption. A Violation may occur if basic assumptions about resource requirements are violated, or the user runs the AUTOSAR Runtime for Adaptive Applications in a configuration that is not supported by its vendor.

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 Secure Communication

AUTOSAR supports different protocols that provide communication security over a network. Integrity of messages can be ensured by the end-to-end protection offered by the [10, AUTOSAR E2E library]. End-to-end protection assumes that safety- and security-related data exchange shall be protected at runtime against the effects of faults within the communication link. Such faults include random hardware faults (e.g. corrupt registers of a transceiver), interference (e.g. electromagnetic interference), and systematic faults in the communication stack. The configuration of end-to-end-protection is done via Service Instance Manifest on level of Service events, methods, and fields (notifier, get, and set methods). Confidentiality and authenticity of messages can be ensured by dedicated configurations for the individual transport protocols (e.g. TLS, SecOC) in the Service Instance Manifest on level of Service events, methods, and fields (notifier, get, and set methods).

14 Risks and Technical Debt

This chapter lists and rates risks associated with the overall architecture of the AUTOSAR Adaptive Platform in Section 14.1. These risks usually might cause that some of the quality attributes of the AUTOSAR Adaptive Platform are not (fully) meat. Section 14.2 lists technical debt of the AUTOSAR Adaptive Platform that may impact its maintainability.

14.1 Risks

14.1.1 Risk Assessment

This document categorizes risks according to their severity. The severity is a function of the probability and the impact of a risk. The probabilities are categorized as follows:

- **very low** - probability is less than 1 thousandth
- **low** - probability is between 1 thousandth and 1 percent
- **medium** - probability is between 1 percent and 10 percent
- **high** - probability is between 10 percent and 50 percent
- **very high** - probability is more than 50 percent

The impact of a risk is categorized as follows:

- **very low** - at most one quality scenario will take additional significant effort to be satisfied
- **low** - more than one quality scenario will take additional significant effort to be satisfied
- **medium** - at most one quality scenario is not satisfied with small gaps
- **high** - at most one quality scenario is not satisfied with big gaps
- **very high** - more than one quality scenario is not satisfied with big gaps

The final severity of a risk is then calculated according to table 14.1.

	Probability				
Impact	very low	low	medium	high	very high
very low	low (1)	low (2)	low (3)	medium (4)	medium (5)
low	low (2)	medium (4)	medium (6)	high (8)	high (10)
medium	low (3)	medium (6)	high (9)	high (12)	high (15)
high	medium (4)	high (8)	high (12)	extreme (16)	extreme (20)
very high	medium (5)	high (10)	high (15)	extreme (20)	extreme (25)

Table 14.1: Risk Severity Matrix

14.1.2 Risk List

No architectural risks were identified yet.

14.2 Technical Debt

No technical debt has been identified yet.

References

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

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

[3] Glossary
AUTOSAR_TR_Glossary

[4] Main Requirements
AUTOSAR_RS_Main

[5] General Requirements specific to Adaptive Platform
AUTOSAR_RS_General

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

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

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

[9] Specification of Manifest
AUTOSAR_TPS_ManifestSpecification

[10] Specification of SW-C End-to-End Communication Protection Library
AUTOSAR_SWS_E2ELibrary