Document Title	Demonstrator Design of Functional Cluster Diagnostics
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	844

Document Status	published
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	R20-11

Document Change History

Date	Release	Changed by	Description
2020-11-30	R20-11	AUTOSAR Release Management	• No content changes
2019-11-28	R19-11	AUTOSAR Release Management	• DM updated to send negative response, when DoIP protocol version is incorrect • Bugs fixed : wrong msg-namelen used when writing to UDP socket, HandleUDPMessage always receives 1 byte instead of DoIp Header length • DM updated to handle multiple DoIP clients with same IP address • code cleanup : Auto-generated bindings and Source files used commonly across DM and unit test application moved to a separate library • Removed generated API tables • Changed Document Status from Final to published
2019-03-29	19-03	AUTOSAR Release Management	• Security registry initial implementation added • Tracking of elevated sessions added • Change in MANIFEST.json to run Diagnostics Manager in parking state • Unit tests refactoring

2018-11-02	18-10	AUTOSAR Release Management	• UDS 0x27 SecurityAccess service processor added • UDS 0x19 ReadDTCInformation service processor partial implementation added • Transport Protocol Plugin APIs added
2018-05-02	18-03	AUTOSAR Release Management	• Deviations to specification updated • Known limitations updated • UML diagrams added
2017-11-22	17-10	AUTOSAR Release Management	• Deviations to specification added • dcm_config json renamed to dm_config json • Stated interface naming mismatch to specification due to SWCL introduction • Changed statement about debounce allocation to events • Statement about DEM part added • Updated with service interface design considerations
2017-03-31	17-03	AUTOSAR Release Management	• Initial release

Disclaimer

Table of Contents

1 Introduction

This document describes the design (approach and decisions) of the Functional Cluster Diagnostics for AUTOSAR's Adaptive Platform Demonstrator.

The decisions taken for the AUTOSAR Adaptive Platform Demonstrator may not apply to other implementations as the standard is defined by the specifications. Nevertheless, the Demonstrator may supplement and ease the understanding of the specifications.

This implementation of Diagnostic Management software provides a reference implementation that is meant to be used in prototype projects as well as to provide a basis for production-grade implementations. The main goal here was to provide proof of concepts for Diagnostic Management features specified in the Diagnostic Management specification of AUTOSAR Adaptive. Optimization for memory consumption and speed have been considered but they haven't played a major role in overall software design and implementation. Furthermore, no static or dynamic code analysis has been performed. Therefore, any series production project deriving from this implementation will have to further fulfill the safety constraints described in the industry standards.

As a service component of the Adaptive Platform, the implementation of Diagnostic Management is subject to dependencies on Execution and Communication Management.

1.1 Known limitations

Limitations of the Demonstrator implementation of the Functional Cluster Diagnostics.

- Not all of the SWS_DM requirements are (fully) implemented and are listed below:

SWS_DM req	Comment
SWS_DM_00005	DoIP is supported but not fully implemented. A minimum set of DoIP services are implemented in order to enable communication via DoIP with external testing tools.
SWS_DM_00011	DM implementation only supports pseudo parallel client concept.
SWS_DM_00012	Variant B is not implemented. For Variant A, see ara::diag::dcm::connection::DoIpHandler::UdpHandler.
SWS_DM_00042	No support for cancellation at interface ara::diag::common:IDiagnosticManagerPluginFactory.
SWS_DM_00047	Security Levels are currently not implemented.
SWS_DM_00049	Silent refusal not implemented, BusyRepeatRequest is returned (see ara::diag::dcm::conversation::Conversation::Handle Message).

△

SWS_DM_00051, SWS_DM_00052, SWS_DM_00180, SWS_DM_00182, SWS_DM_00183, SWS_DM_00184, SWS_DM_00185	Protocol Priorities are not implemented. Requests are queued in order of arrival.
SWS_DM_00061, SWS_DM_00062, SWS_DM_00063, SWS_DM_00247, SWS_DM_00370, SWS_DM_00247, SWS_DM_00371, SWS_DM_00372, SWS_DM_003713, SWS_DM_00374	Service 0x19 ReadDTCInformation only partly implemented.
SWS_DM_00103	Security Access Permission checks are not implemented.
SWS_DM_00104, SWS_DM_00127	Not all mentioned UDS Services are implemented. Internal Service Processors are placed within ara::diag::dcm::service.
SWS_DM_00106, SWS_DM_00108	Implementation does not provide ServiceInterfaces but provides interface ara::diag::dcm::service::IService Validation.
SWS_DM_00112	Environmental Conditions not configurable, hence check not implemented.
SWS_DM_00163	Currently only a 1:1 mapping between events and dtcs is supported.
SWS_DM_00174	Handler for ActiveDiagnosticSessionDataIdentifier not implemented.
SWS_DM_00177, SWS_DM_00190	Handling of ApplicationErrors raised by DidHandler not implemented.
SWS_DM_00193, SWS_DM_00194, SWS_DM_00195	Requirements regarding user-defined fault memory as part of 0x14 ClearDiagnosticInformation are not implemented, because the demonstrator does not support user-defined fault memory.
SWS_DM_00236, SWS_DM_00249, SWS_DM_00250, SWS_DM_00270, SWS_DM_00271, SWS_DM_00172	Service 0x27 SecurityAccess not implemented.
SWS_DM_00070, SWS_DM_00071, SWS_DM_00037, SWS_DM_00379	DiagnosticMonitor related requirements that have not been implemented yet (because related code is not yet implemented).

- There are the following limitations in aspect of the API of the DiagnosticManager:

Interface	Limitations
DiagnosticProtocol	Extended Status, Indentifier and Status not implemented.
SecurityAccess	Not implemented at all.
StorageCondition	Not implemented at all.
Indicator	Not implemented at all.
DataElement	Not implemented at all.

- For externally handled UDS Services tracing of associated requirements is not possible. This applies to the following SWS_DM requirements:

SWS_DM req	UDS Service
SWS_DM_00129, SWS_DM_00130	Services related to addressAndLengthFormatIdentifier.
SWS_DM_00234, SWS_DM_00235, SWS_DM_00268, SWS_DM_00269	Service 0x11 ECUReset.

▽

△

SWS_DM_00140, SWS_DM_00251, SWS_DM_00252, SWS_DM_00197, SWS_DM_00198, SWS_DM_00199	Service 0x28 CommunicationControl.
SWS_DM_00128, SWS_DM_00131	Service 0x34 RequestDownload.
SWS_DM_00134, SWS_DM_00136	Service 0x35 RequestUpload.
SWS_DM_00137, SWS_DM_00138, SWS_DM_00139	Service 0x36 TransferData.
SWS_DM_00141, SWS_DM_00142, SWS_DM_00143	Service 0x37 RequestTransferExit.

1.2 Deviations from specification

- There are the following deviations in aspect of the API of the DiagnosticManager:

Interface	Deviations
DiagnosticProtocol	CancelProtocol(): at least name change required.
GenericUDSService	Renaming still open.
ServiceValidation	Validate(): changes in parameters open according to new proposal required.
ServiceValidation	Confirmation(): changes in parameters open according to new proposal required.
DataIdentifier	Read(): PossibleApplicationErrors is still missing.
DataIdentifier	Write(): PossibleApplicationErrors is still missing.
RoutineService	Start(): MetaInfo parameter and PossibleApplicationErrors are still missing.
RoutineService	Stop(): MetaInfo parameter and PossibleApplicationErrors are still missing.
RoutineService	RequestResult(): MetaInfo parameter and Possible ApplicationErrors are still missing.
DTCInformation	Implementation must be adapted to SWS.
EnableCondition	Implementation must be adapted to SWS.
OperationCycle	Implementation must be adapted to SWS.
DTC	Implementation does not use three-byte values for the dtc Value_ field
ReadDTCInformation	Uses 4byte dtcValue_ for responses instead of high, middle and low byte structure (see above)
ReadDTCInformation	Uses 4byte DTC count instead of 2 byte as specified by UDS, cuts off the upper two bytes
ReadDTCInformation	No support for "DTCStatusAvailabilityMask" as specified in ISO throughout the whole demonstrator code

2 Overview on architecture

2.1 Service Interface Design Considerations

The following sections summarize design considerations about the service interface definition in the Diagnostic Management.

2.1.1 Event Memory Management

The event memory management sub-cluster of the Diagnostic Management (DM) holds essential domain objects (DiagnosticEvents and DTCs) typically with high multiplicity value. These objects shall be accessible and also observable for the Adaptive Application (AA) via middleware. Defining service interfaces for that purpose is not obvious and therefore several use cases shall be thoroughly analyzed. The following paragraphs enumerate issues FT-DIA faced during discussions and aim to summarize along which considerations the current status of service interfaces has been reached.

2.1.1.1 Instance per Configured DiagnosticEvent

The most obvious approach for defining DiagnosticEvent related service interfaces is to create them in the scope of a single event and provide an instance per configured DiagnosticEvent. This however might cause severe performance issues in the middleware during service discovery due to the enormous number of parallel connections. The fear of this performance penalty resulted in other proposals: some of which are completely ruled out, but some are left open for future consideration. For simplicity's sake and in order to rely on only already available middleware technology, the most obvious approach is planned to be used: Service interfaces shall be designed to be instantiated for every configured DiagnosticEvent. The most important arguments for this are the severe impact of all the other approaches on complexity, sometimes even on usability or feasibility. And since the fear of performance issues is not proven by measurements, it makes sense to go into this direction.

2.1.1.2 Fixed vs. Configuration-dependent Interface Definitions for Notifications

DM offers notifications about internal transitions: interested parties are informed about status byte changes and monitor re-initialization for DiagnosticEvents. One idea for decreasing service discovery needs is not to provide dedicated interfaces for these transitions, but instantiate one single configuration-dependent service interface. This interface would contain so many SOME/IP events or fields as many DiagnosticEvents are present in the configuration. On one side this decreases the number of interfaces, but on the other side causes severe configuration issues. The configuration-dependent interface depends on the superset of all configured AAs with diagnostic error management impact. That would mean that by using a common SI, all AAs would depend on each other and therefore, this approach is ruled out.

2.1.1.3 Using General Interfaces

Another idea for decreasing the number of instances is to use general interface definitions for manipulating DiagnosticEvents. General interfaces consist of methods con-

taining DiagnosticEvent identifiers on their argument list determining which object a request is related to. Since there is no such thing as a free lunch, this solution also generates a problem to be solved on the application or middleware side. Identifiers are derived from the DM configuration and need to be available for AA enabling the correct usage of the services provided by DM. There are different approaches how this could be technically solved. One of these is to create an additional layer between the application and the middleware being aware of event identifiers and ensuring transparency for both the middleware and the application. Since such layers are currently not supported, this approach is not realized.

2.1.1.4 Notifications Triggered by Mass Operations

Some external triggers on DM might cause a burst of notifications (for example after executing the UDS request clearAll, status byte change notifications shall be published for most of the configured DiagnosticEvents). In case of such mass operations (and provided that general interfaces are used) notifications shall not be sent out separately, but one single notification event shall be published with a vector containing the notification information for all relevant objects. Since currently no general interfaces are used, this approach is not supported.

2.1.1.5 Separating Manipulation from Notifications

According to the currently used approach, besides using interfaces being relevant for one single DiagnosticEvent, also operations for manipulating purposes are separated from notification SOME/IP events and fields. Logically the triggering of DiagnosticEvent related changes is not connected to the observation of DM-internal status transitions.

2.1.1.6 DiagnosticEvent summary

Currently for each configured event a DiagnosticEvent SI is instantiated. This enables setting event status and on demand manipulating events.

2.1.1.7 Notification about Event-related Data Changes

In some cases AA needs to be notified about changes of the event related data. In the classic AUTOSAR platform, besides being notified, SWCs have the possibility to read the updated freeze frame records. In the adaptive world it would be practical to put notification and read function together by defining a SOME/IP event containing the updated freeze frame record. In order to rule the complexity caused by the N:1 mapping between DiagnosticEvents and DTCs (event combination feature), freeze frame information shall be available on a per DTC basis contrary to the interface defined in the

classic platform (GetEventFreezeFrameData returns freeze frame for an event). The use-case for providing information to the AA about DTC-related data is not clarified. Since the granularity of data description can be different for different use cases (providing the whole freeze frame or providing data per DID), C++ interfaces are planned to be used (plugin interface concept) ensuring the application specific access to the stored data.

2.1.1.8 Exceptions

The updated metamodel provides the possibility to define ExtendedApplicationErrors to service interfaces. Considering the language-binding, this results in the usage of C++ exceptions. The current design does use this option only partially. The rationale is: all the methods of service interfaces are based on provided C/S operations from the classical platform having only one possible error: E_NOT_OK. That means the operations are typically expected to be successfully run, no expected errors are foreseen. An error can only happen due to configuration, integration problems or programming errors. Transforming this to adaptive: service interface definitions do not prescribe the usage of 'checked' exceptions, since they are not foreseen. However 'unchecked' exceptions might be thrown, which is implementation specific. The client using DM event memory management related services is not forced to catch exceptions, unexpected errors can be handled on an arbitrary level.

2.1.1.9 Clearing DTCs by the Application

The ISO 14229 UDS standard so far has not provided the ability to clear DTCs located in other memory origins than the primary. However the problem was solved by running specific routines and clearing DTCs located in the secondary or any user defined memory from application side, which was supported by the classic C/S interface ClearDTC. Since the update of 0x14 ClearDiagnosticInformation UDS service is in progress with options to determine the memory origin to be cleared, we are not forced any more to clear DTCs by the application. As soon as the updated UDS specification is published, the ClearDTC service interface shall be removed.

2.2 Design approach / Design principles

The Adaptive Autosar Diagnostic Manager is the equivalent of the Diagnostic Stack in Classic Autosar (DCM & DEM). Besides a common base, the implementation splits accordingly into a DCM and a DEM subcomponent (see the following class diagram of DiagnosticManager). The following subsections give an overview on the design of these parts. In contrast to Classic Autosar, the interface between DCM and DEM subcomponents is not described in the specification document of the Diagnostic Manager. The Diagnostic Manager is designed to support various extensions of its functionality.

Even some of its built-in functionality can be replaced by external components. For this purpose, the Diagnostic Manager provides an interface whose implementation is equally called a *Plugin* or a *Mediator Component*. Finally, the last section describes the implementation of Service Interfaces which is a topic of cross-cutting concerns.

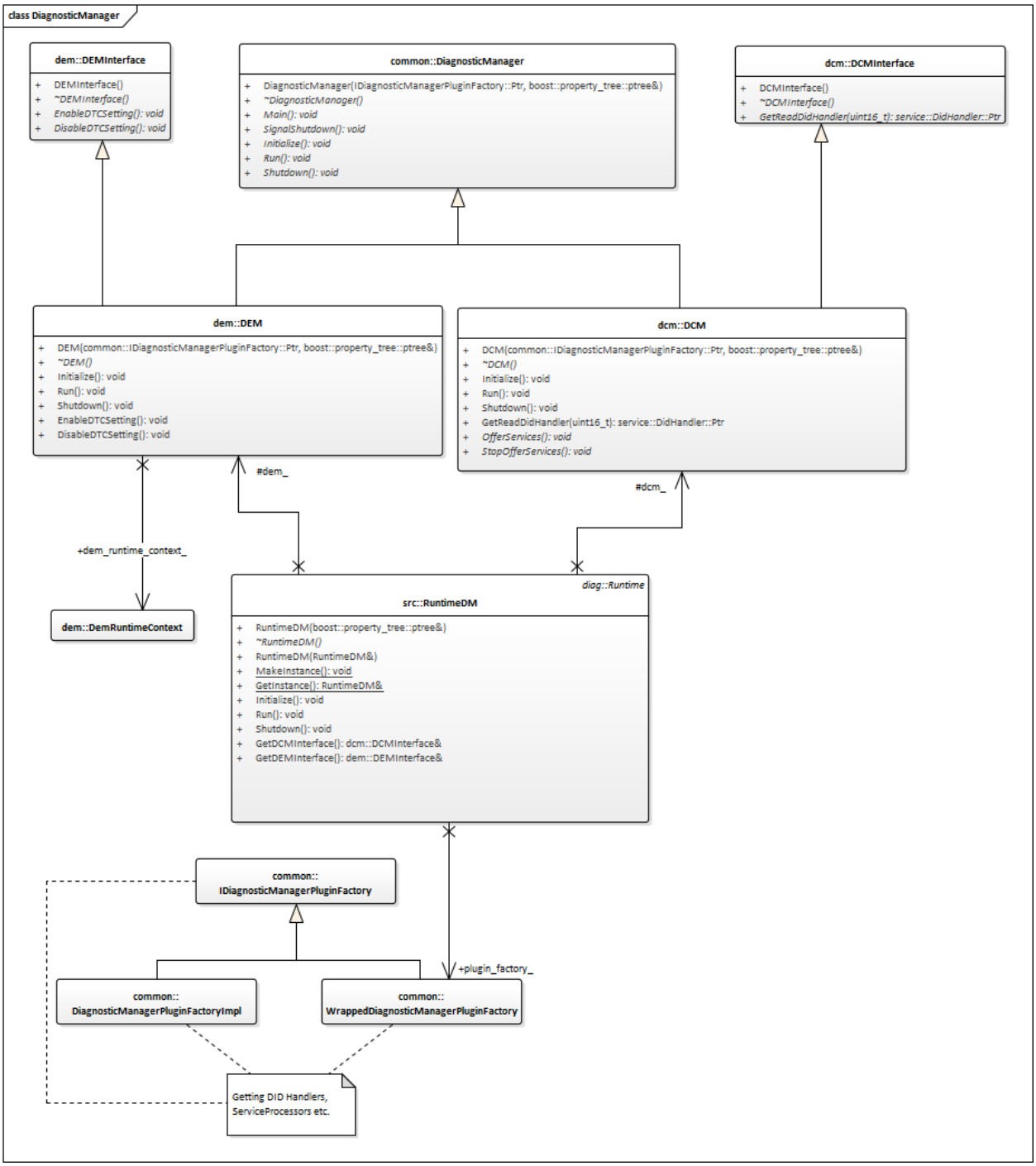

Figure 2.1: Class diagram: DiagnosticManager

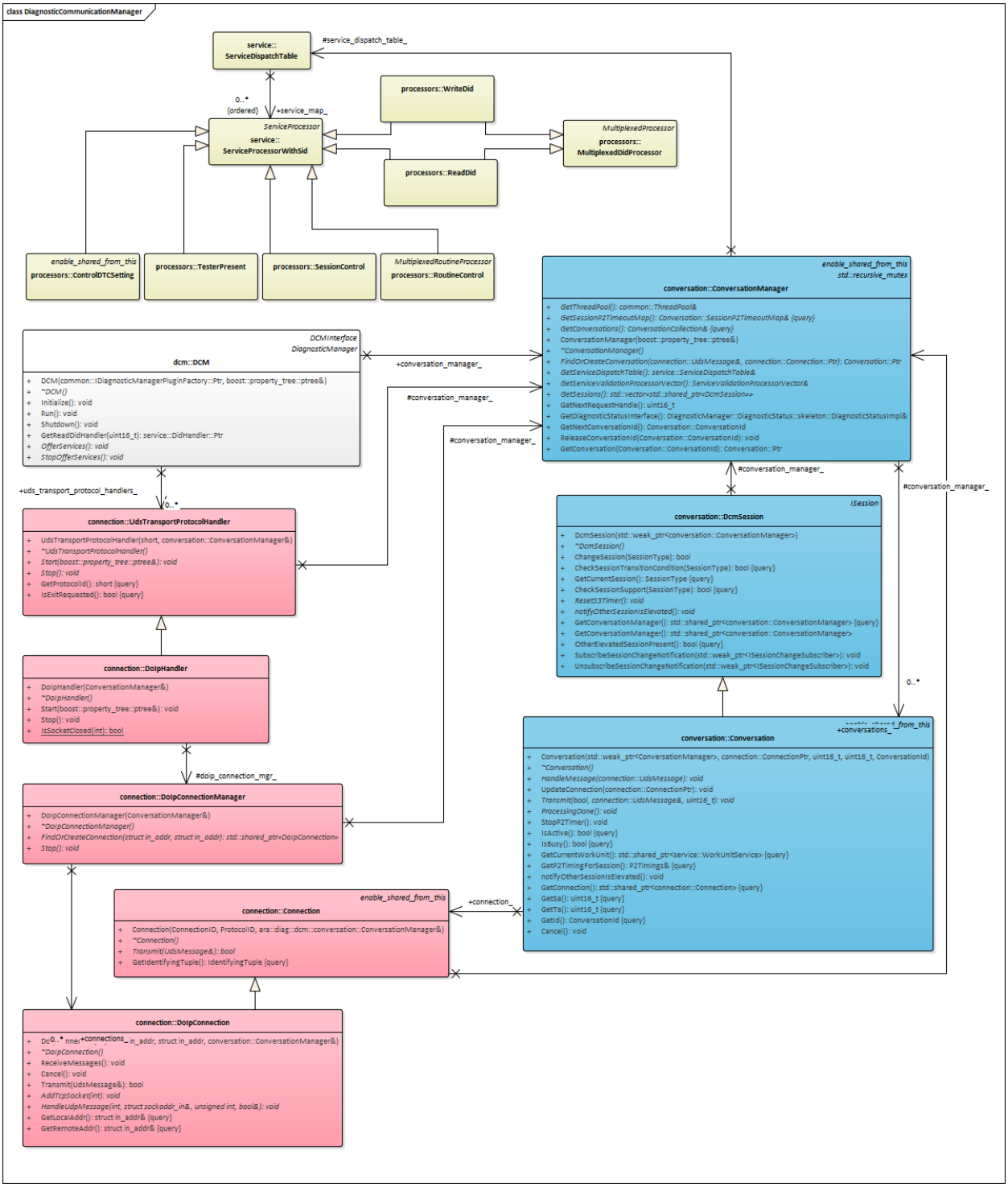

Figure 2.2: Class diagram: DiagnosticCommunicationManager

2.2.1 Common

The common base for the Diagnostic Manager is the singleton instance of ara::diag::RuntimeDM which provides methods for initialization, running and shutdown

of the Diagnostic Manager as an Adaptive Application. As member variables, Runtime DM maintains instances of ara::diag::dcm::DCM and ara::diag::dem::DEM, as well as ara::diag::common::WrappedDiagnosticManagerPluginFactory. The latter is responsible for extensions to the Diagnostic Manager.

The Diagnostic Manager is configurable via a json-file named *dm_config.json* which is located in the *etc* folder of the project. This configuration file is loaded by RuntimeDM and is used for initialization of the DCM, DEM and PluginFactory.

The DCM and DEM both implement the common base class ara::diag::common::DiagnosticManager which provides common lifecycle management methods. The execution of DCM and DEM subcomponents is triggered by RuntimeDM, where each is running in a separate thread.

2.2.1.1 Internal interfaces of DM

Several diagnostics tasks require communication between the subcomponents DCM and DEM, so e.g. in order to store snapshot records the DEM requests data provided by the DCM. Therefore, each component provides an interface to the other component, see ara::diag::dcm::DCMInterface and ara::diag::dem::DEMInterface. In order to get access to these methods, each component stores a reference to the implementation of the interface associated to other component.

2.2.2 DCM

The DCM currently is subdivided into two parts: The *Core* that implements network communication and UDS and the *Plugins* that implement the actual processing of Diagnostic Requests.

2.2.2.1 Internal interface of DCM

The method ara::diag::dcm::DCMInterface::GetReadDidHandler provides an ara::diag::dcm::service::DidHandler associated to a given DataIdentifier. With this the requester (in our context this is the DEM subcomponent) is able to decide on his own when and how to trigger the collection of DID data by calling ara::diag::dcm::service::DidHandler::HandleReadDid method synchronously or asynchronously.

2.2.2.2 DCM Core Features

In overview, the DCM Core supports the following functionality:

- DoIP (for UDS and CANoe)

- UDS decoding
- Session Handling
  - S3 Timer
  - ara::diag::dcm::service::SessionControl
  - ara::diag::dcm::service::TesterPresent
- ResponsePending
  - P2 Timer
- Base Implementation of diagnostic Services
  - ara::diag::dcm::service::ReadDid
  - ara::diag::dcm::service::WriteDid
  - ara::diag::dcm::service::RoutineService
- Plugin Architecture to support additional Diagnostic Services

In the following, we will explain selected features in more detail.

2.2.2.2.1 Network Communication

Currently, the DCM supports communication via DoIP only. It contains a minimal Do IP implementation that is geared towards enabling transmitting UDS messages via Do IP. While the handling of DoIP DiagnosticMessages is fully-featured, processing other messages is done in a very limited way. The DM will also respond to any Routing ActivationRequest with a hardcoded RoutingActivationResponse. The TCP socket of the DM is always open, regardless of RoutingActivation.

For information on how to extend the DoIP communication capabilities or how to extend the DM with support for additional network protocols, see **Adding new Network Communication Capabilities** (3.2.1).

2.2.2.2.2 Diagnostic Service Handling

The DM Core is responsible for decoding a message received over the network to a UDS Message. For further handling, the DM uses the UDS Service Identifier (SID) to find a ara::diag::dcm::service::ServiceProcessorWithSid instance that was configured to handle this SID. For some diagnostic services defined in the UDS standard, the DM core provides Service Processor implementations:

- ara::diag::dcm::service::SessionControl
- ara::diag::dcm::service::ReadDid

- ara::diag::dcm::service::WriteDid
- ara::diag::dcm::service::RoutineService
- ara::diag::dcm::service::TesterPresent

Note that either of these can be replaced with a custom implementation **provided by a plugin** (3.2.2).

The implementation of SessionControl and TesterPresent can be used as is. However, ReadDid, WriteDid, and RoutineService can be extended further. For each of the services implemented by these service processors, the UDS message contains one or more identifiers (Data Identifier, Routine Identifier) that specify, e.g., an application port to contact for executing this diagnostic request in part or in full. When a UDS message for one of these services is received, the respective Service Processor uses an internal lookup table to find a handler instance (ara::diag::dcm::service::DidHandler or ara::diag::dcm::service::RoutineHandler, respectively) which then knows how to, e.g., read the data for a given Data Identifier. These handlers can then use IPC technology (e.g., ARA::COM) to communicate to other processes in order to satisfy the diagnostic request.

The DM Core does not offer any handler implementation. Any handler implementation must be **provided by a plugin** (3.2.2). Note that there is still work going on to provide a general-purpose ARA::COM Plugin.

2.2.3 DEM

The DEM implements the handling of Diagnostic events, binding of DTCs to them and the handling of the DTC status. DTC's and their implying Freeze Frames are stored in a file on the filesystem. The path of a DTC can be configured. In the point of time when the Freeze Frame content is collected, the DM triggers a read of the associated DID's through the DCM Interface and writes it to the related file. A different debouncing algorithm can be allocated to each Diagnostic Event and processed separately. Additionally, Diagnostic Events support aging features where the related DTC content will also be cleared.

2.2.4 Service Interfaces

Service Interfaces currently do not match those described in the SWS. The mismatched/missing interfaces are described to the best of our current ability in the Limitations/Deviations chapter.

2.3 Used OSS components

This sections lists all OSS components used in the implementation of the Functional Cluster, their architectural context and the rationale for their use.

OSS	Version	Explanation	Binding	License	License Version
Boost	1.58.0	Property Tree Implementation	static library	BSL-1.0	1.0
GTest/GMock	1.8.0	SDE (Unit Testing)	static library	BSD-3-Clause	NA
Code Coverage.cmake	NA	SDE (incorporate code coverage metrics into CMake build system)	Configuration	BSD-3-Clause	NA

3 Configuration and Extensibility

The Adaptive Autosar Diagnostic Manager is the equivalent of the Diagnostic Stack in Classic Autosar (DCM & DEM). The DM currently consists of two parts: The *Core* that implements network communication and UDS and the *Plugins* that implements services whose service interfaces require some customization like DataIdentifier and RoutineService.

A basic implementation of the DEM is also currently available.

In the following, we will give a brief description how to configure and extend the Diagnostic Manager. We will first focus on how to use and configure the functionality provided by the DM Core before showing how to extend the Core functionality with additional plugins. Finally, we will give a brief overview on how to extend the network communication capabilities of the Diagnostic Manager, i.e., how to implement new diagnostic protocols.

3.1 DM Configuration

The DCM part of the Diagnostic Manager is configured through the file `etc/dm_config.json`. The configuration file contains three sections: `Network`, `Conversation`, and `ServiceProcessors`.

3.1.1 Section "Network"

The `Network` section contains the socket configuration for the DoIP Protocol Handler. `local_ip` sets the listen address for both the TCP and UDP sockets. `UdpPort` sets

the port to use for UDP communication and `TcpPort` sets the listen port for new TCP connections. Both port values are given as decimal numbers.

3.1.2 Section "Conversation"

The `Conversation` section contains the settings for UDS conversations with testers. Parameter `WorkerThreads` denotes the number of worker threads, i.e., the number of UDS requests that can be processed in parallel. Note that for each tester, at most one request is processed at any given time.

The parameter block `p2Timings` configures the values for the P2 and P2* timers in each of the four hard-coded sessions. The unit for these settings is milliseconds. The values are given as decimal numbers.

3.1.3 Section "ServiceProcessors"

The parameter block `ServiceProcessors` configures the service dispatch table of the DCM. It contains an entry for each SID that the DM can service. Every entry must contain at least two values: `SID`, denoting the UDS Serivce ID (given in decimal) and "class", denoting the ara::diag::dcm::service::ServiceProcessorWithSid subclass that implements the service for the given SID. All other entries are dependent on the ServiceProcessorWithSid implementation. The ServiceProcessors provided by the DM core either take no additional options or take exactly two additional options:

- `allowedSessions`: An array containing the numeric IDs of the hard-coded sessions, in which this service is allowed to be accessed
- `multiplexer`: A parameter set used by ara::diag::dcm::service::DidHandler and ara::diag::dcm::service::RoutineHandler to identify which Data Identifier/Routine Identifier to map to what middleware service.

All numeric values are given as decimal numbers.

3.2 Extending the DM Core

This chapter describes how to extend the DM Core. The first subchapter describes how to add an specific transport protocol instead or in addition to DoIP (called `Custom TP` in the following figure), while the second subchapter describes how to add an plug-in for diagnostic service processing.

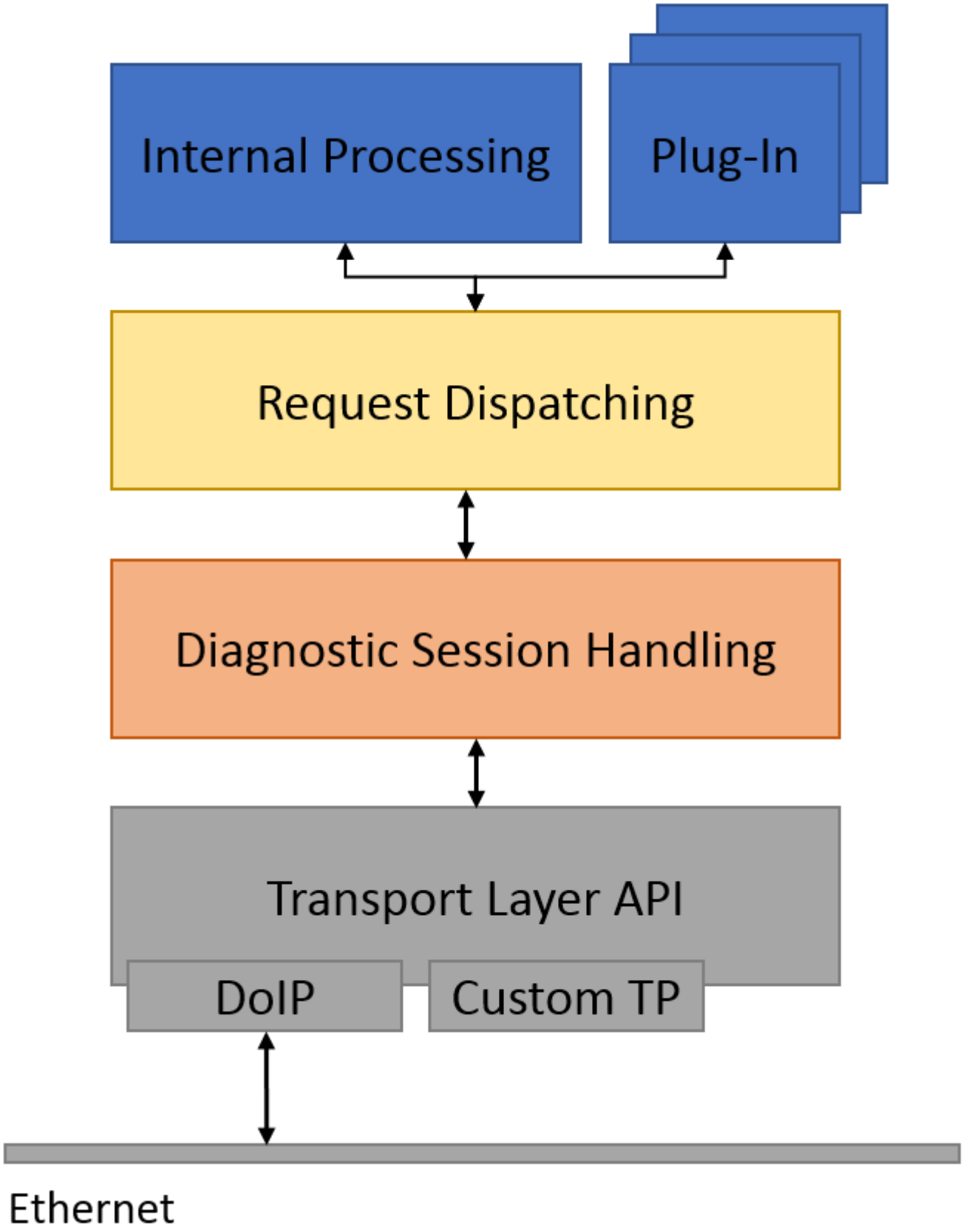

Figure 3.1: Diagnostic Service Handling

3.2.1 Extending the Network Communication Capabilities

In general, Network Communication for a given protocol is implemented in terms of two classes:

- A subclass of ara::diag::dcm::connection::UdsTransportProtocolHandler that represent the communication capability over a network technology or protocol
- A subclass of ara::diag::dcm::connection::Connection that represents the communication relationship between two nodes using the same network technology or protocol.

3.2.1.1 Extending DoIP

If you are looking to extend the DoIP support in the DCM, look at ara::diag::dcm::connection::DoIpHandler and ara::diag::dcm::connection::DoIpConnection. Most likely, you will want to extend ara::diag::dcm::connection::DoIpConnection::ReceiveMessages() and ara::diag::dcm::connection::DoIpConnection::HandleUdpMessage().

3.2.1.2 Adding a new Network Protocol

If you are looking to add support for a new communication technology into the DM, you must provide additional subclasses of ara::diag::dcm::connection::UdsTransportProtocolHandler and ara::diag::dcm::connection::Connection that handle the communication and provide ara::diag::dcm::connection::UdsMessage objects to the upper layers of the communication. See ara::diag::dcm::DCM::Initialize() for how to instantiate your new classes.

3.2.2 Extending the Diagnostic Service Processing with Plugins

Plugins for the DM consist of the following:

- A *plugin folder* located in `plugins/`. The name of the Plugin Folder defines the name of your plugin.
- A file `plugins/pluginFolder/CMakeLists.txt` informing CMake about the existence of your plugin at compile time.
- A subclass of ara::diag::common::IDiagnosticManagerPluginFactory informing the DM about your plugin at runtime, located in the namespace ara::diag::plugins::*pluginName*. Furthermore, this class will provide runtime and storage to code in your plugin.

Note: You can find an example plugin under `plugins/plugin_template/`. Feel free to copy the code to your own plugin directory and extend it there.

There is also the option to build a plugin in a separate project into a shared object and load it dynamically during runtime, for more information about this refer to **Dynamic Loading** (3.2.2.3).

3.2.2.1 DM Plugin Example

Below we will use the `DcmCalculator Plugin` that can be found in `ara-api/diag/diagnostic-manager/src/plugins` as an example. To contribute a new plugin, proceed as follows:

1. Select a name for your plugin. In this example, we will call the Plugin `dcm_calculator_plugin_typed`.
2. Create the plugin folder `plugins/dcm_calculator_plugin_typed/`
3. Create a Class ara::diag::plugins::dcm_calculator_plugin_typed::DcmCalculator PluginTypedPluginFactory. Choose any location in your plugin directory for the header file, e.g., `plugins/dcm_calculator_plugin_typed/dcm_calculator_plugin_typed_plugin_factory.h`.

Currently, you can contribute subclasses of ara::diag::dcm::service::ServiceProcessor WithSid (handling a complete UDS Service), subclasses of ara::diag::dcm::service::Did Handler (extending the Core-provided ReadDid and WriteDid implementations) and subclasses of ara::diag::dcm::service::RoutineHandler (extending the Core-provided RoutineService implementation). All instances must be created by methods of your PluginFactory, in the example of the `DcmCalculatorPluginTypedPluginFactory`. The `DcmCalculator Plugin` provides subclasses of:

- ara::diag::dcm::service::DidHandler:
  - `DcmCalculatorWriteDidHandler_Input1`
  - `DcmCalculatorWriteDidHandler_Input2`
  - `DcmCalculatorReadDidHandler_Output`
- ara::diag::dcm::service::RoutineHandler
  - `DcmCalculatorRoutineHandler_Add`
  - `DcmCalculatorRoutineHandler_AddArg`
- ara::diag::dcm::service::ServiceValidation
  - `DcmCalculatorServiceValidation`

If your plugin requires additional setup or data structures, implement the respective code in the constructor of your PluginFactory.

3.2.2.1.1 Building Plugins

To enable your plugin, set the CMake option `PLUGIN_customCode` to `ON` or `SHARED`. For the `DcmCalculator Plugin` example, run the following in your build tree: `cmake -DPLUGIN_dcm_calculator_plugin_typed=ON` or `cmake -DPLUGIN_dcm_calculator_plugin_typed=SHARED`, `ON` will embed the plugin in the DiagnosticManager binary, `SHARED` will compile the plugin into a shared object for dynamic loading. We will discuss **Dynamic Loading** (3.2.2.3) later.

Simple-Build Plugins

In case your plugin only needs to build the files under its directory tree, i.e. it does not need to define additional targets and their dependencies, link external libraries, build sources from additional paths, or have generated files, follow these steps:

1. Create `plugins/customCode/CMakeLists.txt` with the following contents:

```
DEFINE_DM_PLUGIN("custom_code_plugin_factory.h")
```

   The argument to `DEFINE_DM_PLUGIN()` is the relative path from the `CMakeLists.txt` to the header file declaring CustomCodePluginFactory.

2. You have now defined an empty plugin. Proceed to implement any code you want to contribute to the DM.
3. CMake will automatically add all `*.cpp` files in `plugins/customCode/` and all of its subdirectories to the compile of the DM, the include directory will have the Project root directory and `source` folder. Note that there is a slight pitfall: The directory `plugins/` is always part of the include path, regardless of the set of plugins that has been enabled.

Advanced-Build Plugins

In case the Simple-Build is not sufficient, follow these steps:

1. Create `plugins/customCode/CMakeLists.txt`
2. Use the created `CMakeLists.txt` file to define any additional targets, sources, builds, includes, etc.
3. Call the macro

```
DEFINE_DM_MANUAL_PLUGIN(
    PLUGIN_HEADER
    PLUGIN_SOURCES
    PLUGIN_INCLUDES
    PLUGIN_GENERATED_SOURCES
    PLUGIN_CMAKE_TARGET_DEPENDENCIES
    PLUGIN_ADDITIONAL_LIBS)
```

where:

1. `PLUGIN_HEADER` plugin main header file
2. `PLUGIN_SOURCES` a list of plugin source files to be built
3. `PLUGIN_INCLUDES` include directories
4. `PLUGIN_GENERATED_SOURCES` source files that CMake would ignore if not present, note that they still need to be included in `PLUGIN_SOURCES`.
5. `PLUGIN_CMAKE_TARGET_DEPENDENCIES` additional target dependencies that should be made before the build, e.g. generation targets.
6. `PLUGIN_ADDITIONAL_LIBS` libraries added to either the executable or the plugin shared object, note that find_package and find_library commands should be called in the plugin's own CMakeLists.
7. Optional 7th argument: plugin class name, if not given `${PLUGIN_NAME}PluginFactory` will be used by default, where `PLUGIN_NAME` is taken from the plugin source directory.

3.2.2.2 DM Plugin Runtime Initialization

When starting, the DM first instantiates the ara::diag::common::IDiagnosticManager PluginFactory implementations found in the active plugins. It then queries these instances for ServiceProcessors and Handlers according to its configuration file. For ServiceProcessors, it passes the SID, class name, and configuration subtree to the Factory. If the factory can provide a ServiceProcessor for the respective configuration subtree, it must return a std::shared_ptr to a new instance of the ServiceProcessor. If it cannot provide a ServiceProcessor, it must signal this by returning an empty std::shared_ptr. The DM continues to try and find a Factory that can provide an implementation until any Factory does so or until it has queried every factory once.

When using ara::diag::dcm::service::ReadDid and ara::diag::dcm::service::WriteDid for the respective UDS service, these ServiceProcessors will use the given `root_factory` to find their respective Handler objects.

The new ServiceProcessor may optionally use the given root_factory object to query for additional Handler objects. It again provides the Factory object with the configuration subtree for the requested handler. As with ServiceProcessors, the Factory must return a std::shared_ptr to a new instance, if it can provide a handler, or an empty std::shared_ptr otherwise.

3.2.2.3 DM Plugin Dynamic Loading

The following is needed for a plugin to be dynamically loaded:

1. For a plugin to be dynamically loadable it needs to Declare the functions `GetDMPluginObject` and `DeleteDMPluginObject`. Refer to `dummyMiddleware` plugin for a sample of their declaration and implementation.
2. For DiagnosticManager to attempt to dynamically load a plugin it needs the path of that plugin to be added to the configuration under "Plugins". Check `etc/dcm_cfg.json` for an example of adding this path to the configuration.

The user has 2 options for building plugins in shared objects to be loaded later:

3.2.2.3.1 a) Building from inside the DiagnosticManager build tree

`SHARED` option for a plugin builds it into a shared object instead of linking its code in the executable, to give the option to build a plugin and dynamically load it later, dynamic loading gives the possibility to build a plugin from even a completely separate project and dynamically load it in DiagnosticManager in runtime.

Additionally the preprocessor macro `PLUGIN_IS_SHARED` is defined for all the plugin sources in case the `SHARED` option is selected for that plugin, to facilitate adding/removing code that is specific to the `SHARED` or static option if the plugin is built in DiagnosticManager build tree.

3.2.2.3.2 b) Building from a completely separate project

Please refer to the documentation of DMCommon Library and example under `diag/doc/lib_dm_common_example`.