Document Title	Demonstrator Design of Functional Cluster Communication Management
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	842

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	• Add Instance Specifier to Instance Identifier translation. • Add Fire and Forget methods. • Add support for Fields in DDS. • Add Service Versioning. • Remove ara::per usage in ara::com. • Add support for invalid values. • Use new APIs for getting events with SOME/IP.
2019-11-28	R19-11	AUTOSAR Release Management	• Add support for TLV. • Introduce new code generator. • Add support for ApApplicationErrors. • Remove SecOC code. • Removed generated API tables. • Changed Document Status from Final to published.
2019-03-29	19-03	AUTOSAR Release Management	• Built-in vsomeip e2e protection is now functional. • DDS Service discovery (skeleton side) implemented. • Build system cleanups

2018-11-02	18-10	AUTOSAR Release Management	• Include DDS Events handling and serialization. • Update the APIs to use ara::core data types. • Fix and update current deviations.
2018-05-02	18-03	AUTOSAR Release Management	• Remove Field combinations limitation. • Add implementation-reserved SOME/IP ids for events and methods. • Update vsomeip version to 2.10.10.
2017-11-22	17-10	AUTOSAR Release Management	• Include E2E design information. • Update vsomeip version to 2.6.2. • Fix included images.
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 Communication Management for AUTOSAR's Adaptive Platform Demonstrator.

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

1.1 Known limitations

This is the list of limitations the reference implementation currently has:

- *No common receive pool for events and fields.* The middleware specification obviously expects a common pool of received data per event/field. All instantiated proxies then take shared ownership for all data that is valid for their specific caching strategy.
- *The SOME/IP id value of 0xFFFE cannot be configured in the deployment of Events and Methods (implementation-reserved).* The limitation comes from the current design of offering all possible Field combinations (getter, setter, notifier).

Additionally, the following features are part of the specification but are not implemented yet in the demonstrator:

- DDS Network Binding serialization/deserialization does not support data types ENUM_TYPE and ASSOCIATIVE_MAP

1.2 Deviations from specification for SOME/IP Network Binding

The actual demonstrator implementation has the following deviations from the specification:

Basic types

- float32 and float64 are not serialised in IEEE754 format
- no byte order unification but pure dump into the serialisation stream

Struct data types

- attribute sizeOfStruct is not written into the output stream
- attribute sizeOfStructLengthFields is ignored

Strings

- unicode encoding not supported

- check on size of string depending on encoding (eg UTF-16LE and UTF-16BE strings shall have an even length) not supported
- no padding of unused memory in the serialiser with 0
- network byte order conversion between UTF-16LE and UTF16-BE not supported
- fixed length strings and dynamic strings are treated the same (ara::core::String)
- string size from the model is ignored

Fixed-length strings

- size of string is always written
- size of string is read as len when parsing, and then len chars are read
- no '\0'-terminator expected

Dynamic-length strings

- size of string assumes 1 byte per character

Arrays

- size is always written (no difference between fixed and dynamic length)
- size indicates number of elements, not size in bytes
- attribute sizeOfArrayLengthField is ignored (written size is always 32 bits)
- model is not queried for size of fixed-length arrays
- no support for optional elements/parameters

Union

- serialization of unions is not supported
- unions should be serialised with length (if sizeOfUnionLengthFields > 0), type and element field

Maps, Unordered Maps, Sets, Unordered Sets

- AUTOSAR_SWS_SOMEIPTransformer doesn't specify maps, set, unordered_map and unordered_set
- elements of type 'map' and 'unordered_map' are serialized like a vector of 2 values (vector of pair)
- elements of type 'set' and 'unordered_set' are serialized like a vector of a single value
- duplicated keys are read from the serialized stream and only the first value is taken, the next ones are ignored

2 Overview on architecture

2.1 Design approach / Design principles

2.1.1 Overview

This C++14 implementation of ara::com focuses on providing a complete, stable and well-documented reference implementation that is meant to be used in prototype projects as well as a basis for production-grade implementations and as a source to compare other implementations to.

While some care was taken to also optimize it in terms of memory consumption and CPU time, it was not the primary goal and no runtime measurements were made in order to improve resource consumption based on numbers. In addition, no static or dynamic code analysis was done, which would be necessary for safety relevant products. It is up to the series project to achieve those goals.

It is recommended to be familiar with the ara::com API before reading this document. Please refer to the AUTOSAR explanatory document to get an overview of the API and its design principles.

The following paragraphs describe the main design goals, their rationales and how they were achieved.

2.1.2 Minimize the output of code generation

2.1.2.1 Rationale

Since AUTOSAR models all data types, interfaces and software components in terms of its own meta-model, we need to generate all interface types from source models and translate any given information to a format our compiler understands. Many modern languages including C++11 provide features that are capable of generating code inside the compiler and therefore we need to decide how much code is generated by the ara::com generator, how much is expressed as C++ templates and code and what is even done at runtime, e.g. when dealing with instances which is typically handled using service discovery instead of static code.

We decided to reduce the amount of code generated by the ara::com generator as much as possible, using C++ features like inheritance and templates to produce the desired output. We did this because we believe that utilizing code generation facilities of the target compiler will result in higher quality code and it will also leverage optimization potential as this is one of the primary goals of modern C++ compilers. In addition, it will also keep the code generator smaller as we require less text template code.

2.1.2.2 Design

In order to achieve this goal, we use two main features of C++11: Generic programming and inheritance. The following chapters will shed light into how we utilize these techniques to minimize code generation.

2.1.2.2.1 Generic programming for types and static information

Most of the middleware code is built around user-defined data types and method call signatures. Since this data is treated mostly the same, we utilize generic programming aka templates to write code that can be reused by the ara::com generator by simply specifying the concrete type. The C++ compiler will then generate code that adapts to fit the needs of the required data type.

As not all types behave the same, we use template specialization if needed. This is especially true when it comes to encoding objects into a format that can be sent over the wire. The current implementation of marshaling using template mechanisms works by defining a standard way how to serialize objects and template specialization whenever a type needs special treatment. Examples for the latter are containers and strings. Any other type for which no specialization exists is then serialized using the default, non-specialized template classes. They support two kinds of types: Whenever a type is TriviallyCopyable (`http://en.cppreference.com/w/cpp/concept/TriviallyCopyable`), it is serialized by simply doing a byte-wise memcpy into the output buffer. If not, the (class-type) is required to be enumerable by providing a special method that calls a provided functor on each member. This method is per default generated by the ara::com generator for every struct it emits:

```
 {C++}
struct Struct1Type {
  bmw::paads::fs::helloAdaptiveWorld::Struct2Type sb2;
  ara::stdtypes::Int32 s;

  using IsEnumerableTag = void;
  template<typename F>
  void enumerate(F& fun) {
    fun(sb2);
    fun(s);
  }
};
```

As can be seen, the functor itself needs to be a templated function as the argument will have the same type as the currently enumerated struct member type. The signature of the functor should, therefore, look like this:

```
 {C++}
template<typename T>
void functor(T& value);
```

A second aspect that can be solved using templates concerns the static non-type information that is taken from the AUTOSAR model. This mainly includes SOMEIP service-

method- and event IDs. This information is needed by the VSOMEIP binding whenever a method call or an event shall be sent via SOME/IP using its central daemon called vsomeipd. Besides the type parameter which is always required, many VSOMEIP base classes therefore take another template parameter called ServiceDescriptor, FieldDescriptor or MethodDescriptor, respectively. As this is static information, the C++ compiler might bake these values directly into the generated machine code whenever it's needed for a call to vsomeipd.

2.1.2.2.2 Variadic templates for function calls

Variadic templates take generic programming one step further by not only providing a placeholder for types but also leaving it open to the developer how many types the user of the class wants to provide. This comes in handy for the middleware as we need to deal with a varying number of arguments in case of function calls.

However, variadic templates aren't always a fortunate choice due to the complexity they introduce into the code. As an alternative, we could deal with this issue using placeholder types and assuming a maximum number of arguments. Unfortunately, this would clutter the code in a way we think is even more complicated, hard to read and maintain plus it would lead to a maximum number of arguments we'd need to accept as implementation specific. Another alternative is the complete move of function call dispatch generation to the ara::com generator. But since this is exactly what we want to avoid, we decided to still use variadic templates and rely on the compiler to generate function signatures with the appropriate number of arguments.

This has again an impact to the marshaling: Serialization and deserialization of the arguments also need to be done in a variadic way. During serialization, a recursive template function is used to serialize the arguments one by one (see the class `Marshaller`) whereas deserialization works by repeatedly calling a template method using the ellipsis operator that expands an expression to a comma-separated list of expressions, replacing type and argument by the respective type and argument of the method signature. See `ara::com::internal::vsomeip::skeleton::ServiceImplBase::UnmarshallAndCall` for the respective code.

2.1.2.2.3 (Virtual) base classes and composition for common code

Besides generic programming, C++ also supports inheritance and composition as another means to share code between several objects. In this implementation, we use inheritance to achieve two goals:

1. *Reuse*

   ara::com provides three communication paradigms: Fields, Events and Methods. While methods and events are very different, both conceptually and in terms of their interface and therefore do not share any code besides data serialization, fields combine event notification and method semantics. Therefore, the classes

that implement these three means depend on functionality of each other. The most prominent usage is the class `Field` that directly inherits all functionality of `Event` and includes objects of class `Method` to implement the required accessor methods of fields. See the Enterprise Architect model (ara-api/com/doc/misc/architecture/ara_com.eap) to precisely see how these classes are related to each other.

2. *Fight code bloat*

   Templates in C++ are a form of code generation that may - if misused - lead to code bloat. While compilers significantly improved on this issue, there are still things we can do to also decrease code bloat the compiler might not be able to detect. The main effect of this reasoning can be seen when looking at the design of almost any templated class of the ara::com reference implementation as it almost always derives from a non-templated base class that contains all methods that are type-agnostic. In concrete binding implementations, it is then possible to separate (and therefore share) all methods that are common for all types from methods that need type information. For a complete picture on how non-typed and typed classes are related, please see the Enterprise Architect model.

2.1.3 Support multi binding

2.1.3.1 Rationale

One design goal of the ara::com API is the independence from the underlying transport protocol. It is up to the implementation how to connect the API provided to the developer to an IPC mechanism of his choice. As both data driven communication through Events and control flow based communication through Methods are - from the point of view of the service - one-to-many relations, it is possible to offer a service through more than one transport mechanism.

Offering a service and data transfer during service usage can be divided into two parts:

1. Serialization and marshaling - transforming high level data structures into a low level representation that can be transferred over arbitrary channels.
2. Transportation - transferring a low level data chunk from sender to receiver and mapping of request messages to the response messages.

It is valid to assume that either of the two or both will be different depending on various aspects of the underlying ECU architecture. Such aspects include (likely incomplete):

- Location of the service and the client (e.g. local vs. IPC vs. network communication)
- Providing a service over more than one on-the-wire protocol (e.g. Ethernet vs. UDS communication)

- Debug harnesses (during testing to introduce fake data or recording data between processes)

In all those cases, services need to provide their data to multiple network implementations (==bindings) and clients need to be able to utilize different bindings depending on where/how an instance of a certain service is provided. Therefore, we need to be able to provide different bindings for the same service that are selected at runtime, depending on service availability and ECU configuration.

As a side node: The terms "serialization" and "marshaling" are used to distinguish pure type serialization from serializing function calls. In the latter case, we use the term "marshal(l)ing", which is, according to common definition, not a 100% correct. But since we needed another name for it, we just used it :)

This is an example, taken from the VSOMEIP binding. It shows the relation between the classes that implement interface elements and the serializing/marshaling classes.

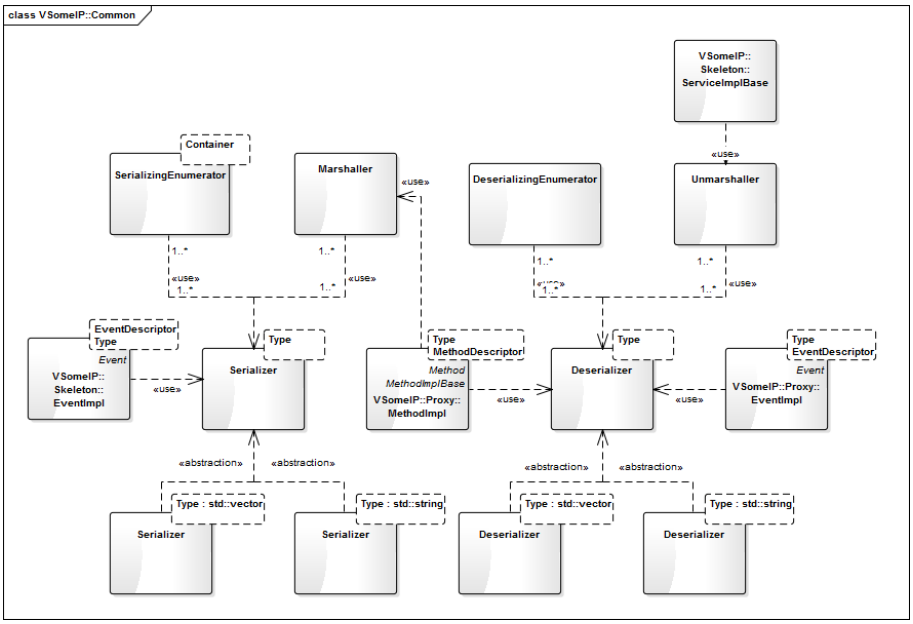

Figure 2.1: Relation between binding classes and VSOMEIP marshaling

2.1.3.2 Design

The implementation of multi-binding is very different, depending on whether we are on proxy or on skeleton side. This becomes clear if we realize that on proxy side, the middleware code will focus on exactly one communication channel which is selected by the service discovery code whereas on server side, the middleware has to serve all possible communication channels which may also lead to duplicate serialization as different channels might have different serialization rules. In the next chapters, we will therefore look at both sides separately.

Proxy

On proxy side, the implementation of ara::com follows the classic interface/implementation pattern where the interface is used by the frontend code to gain access to the implementation which is located within the binding code. For each element the AUTOSAR interface specifies there is one getter method that returns the binding implementation of the respective element. This implementation is then used to initialize the references that reside in the generated frontend class. After this initial setup, access to any element is directly routed through these references to the binding implementation so that there is no further overhead besides the dereferencing and the vtable jump.

This is an example of a generated proxy frontend class that initializes its members using the getter methods from the binding base class:

```
 {C++}
class InterfaceName1 : public ara::com::internal::proxy::ProxyBase<
    InterfaceName1ProxyBase>,
                       public bmw::paads::fs::testService::InterfaceName1 {
 public:
  explicit InterfaceName1(HandleType& proxy_base_factory) :
      ara::com::internal::proxy::ProxyBase<InterfaceName1ProxyBase>(
    proxy_base_factory),
      DataNameA(proxy_base_->GetDataNameA()),
      DataNameB(proxy_base_->GetDataNameB())
  {}

  fields::DataNameA& DataNameA;
  events::DataNameB& DataNameB;
};
```

It is the implementation's duty to figure out which binding to use in case there is more than one path to a specific service. Currently, just the first possible binding is used, but it is recommended to replace this behavior by a smarter algorithm that bases its decision on a cost function that takes resource consumption and reliability of the bindings into account.

HandleType as class factory

As specified by ara::com, the implementation offers a type called `HandleType` that is used as an argument to the service proxy's constructor. This HandleType holds the instance and binding information and is used to instantiate the binding implementations of the service itself (called `Adapter` in case of VSOMEIP) as well as the interface elements. This works by instantiating the binding object as soon as the constructor of the proxy frontend class is called, but before all the getters of the individual interface elements are called during initialization of the element references. This approach is reliable as the order of initialization is given by the order of the class members inside the concrete object and the reference to the binding objects is before any generated interface element.

Skeleton

On skeleton side, the middleware needs to always use all possible bindings the service was configured to use as it is unknown which clients will use the service over which transport. On proxy side, we were able to directly dispatch each action to the binding code using dynamic dispatching through abstract base classes as we only needed to talk to one implementation. On service side, we need to dispatch all requests to several objects at once, so we need a list of references instead. This is implemented through dispatcher classes that hold the list of binding objects. Since those classes are non-virtual, the cost of this approach is reasonably low.

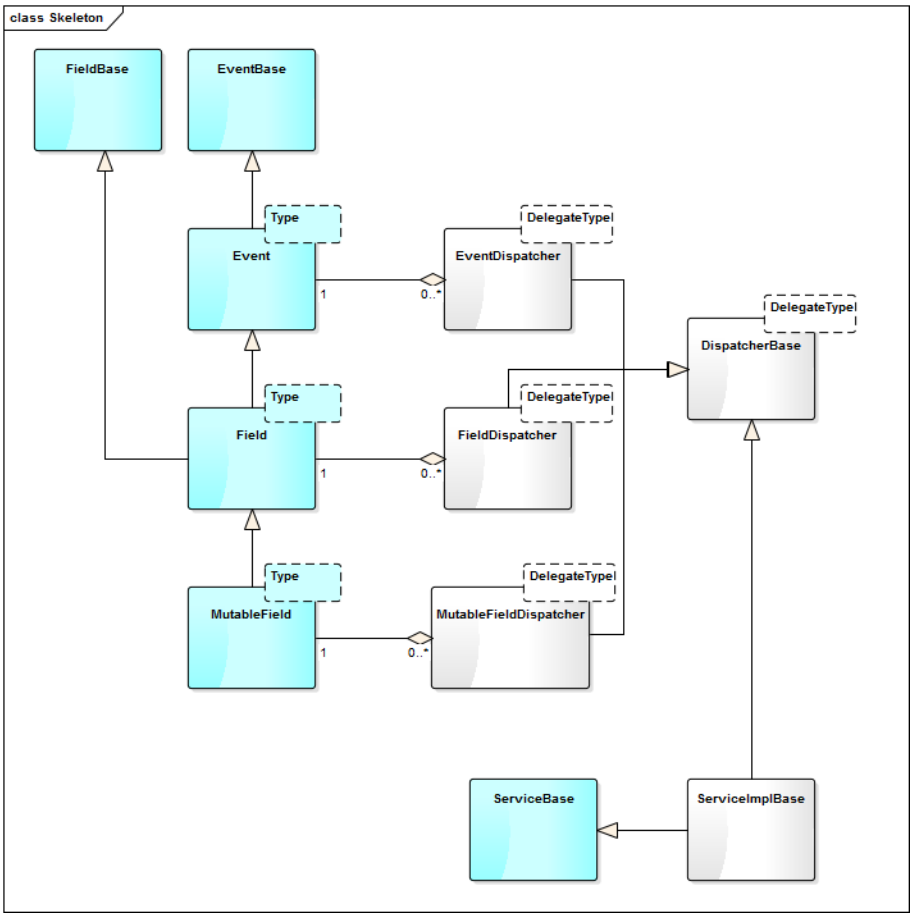

Figure 2.2: Relation of dispatcher and binding classes

Special attention has to be paid to the allocate/send calls a service offers. This part of the API is meant to be used in order to directly send data via binding specific buffers that are allocated such that the number of copies for the data is kept low (e.g. directly into hardware buffers). However, if we need to copy the user data to multiple buffers anyways, it may make sense to only call allocate for one binding (preferably the one that gains the most benefit) and use the normal send call of all other bindings. This of course only works if the allocated buffer can be accessed by all bindings in question, so the middleware runtime does need a lot of information about the implementation details of the bindings.

2.1.4 Type erasure on library boundaries

2.1.4.1 Rationale

As per the specification, service instances on proxy and skeleton side are identified by the C++ type of the interface and the instance identifier. While the latter is clearly something dynamic, the former is static and may only be changed at build time.

Now, in our implementation, we have several ways to identify types of interfaces when the user calls either [Start]FindService or OfferService of the respective Proxy or Service class:

1. We could turn the following calls into templates that take the respective interface type as template parameter.

   This approach fits very naturally into how types should be forwarded in C++ since they become part of the function/method/class name. However, each function/method/class that is templated is in fact a number of classes that are generated on the fly during compilation and thus would scale with the number of existing interfaces. While this is OK in one single binary, it might become tedious if we at some point need to cross binary boundaries, either because we're doing IPC or because we're using dynamic linking. The former wouldn't work at all while the latter would lead to one function call export per class per offered function where function refers to the intial service discovery functions like OfferService or Find Service.

2. We translate the type into a unique ID that is used as an additional parameter for all function calls into the library, indicating the desired type.

   This technique is called type erasure (`https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Type_Erasure`): Instead of using the type information from the compiler, we assign a unique name to a type which can be used at runtime. As soon as we do this, we can use the encoded type information just like any other variable, e.g. pass it around as parameters. Thus, we now only need one call that uses this type information along with a base class and using a `static_cast`, we can always safely restore the C++ type.

   While downcasting is considered harmful in general, we believe that in this case its benefits outweigh the drawbacks and as we do have the type information present, we can also always restore the C++ type whenever we are using the provided instance in the code again.

2.1.4.2 Design

Type erasure is done whenever we either cross a library boundary or if we can reuse a base class type reference of a class we inherit from. The latter saves us some bytes as we do not have to store the same reference twice or introduce a virtual function that does the upcasting for the base class.

Remember that downcasting/type erasure is only done on either a proxy or a service adapter. Both of these class types inherit several class members from one common base class that has the same name as the interface definition inside the AUTOSAR model. One of its members is called `service_id` and contains a unique value of type `ara::com::internal::ServiceId`, which is easily serializable and comparable. Therefore, type erasure is - as expected - a compile time operation.

Restoring the type, however, has its runtime costs in general: Since the variable that holds the type information is now a runtime entity, we also need some form of runtime dispatch as well. However, it turns out that in our case we only need to downcast when we work inside a class that already holds the full type information, so we know what to expect. By using a `dynamic_cast` in debug builds we can also check this assumption and we will get an exception if our assumption is wrong so that we can emit a bugfix as soon as possible.

2.1.5 Static linking of the generated middleware code

The following paragraph is specific for the VSOMEIP binding code. However, it may be applied to other bindings as well and might even be converted into a general approach how a binding shall configure itself.

The general system design of Adaptive AUTOSAR does - at the moment - not allow for dynamic libraries due to safety reasons. It is therefore necessary to link any code that is required by ara::com directly into the target executable. Since the middleware heavily relies on code generation (either by the included code generator or by the C++ compiler) it is crucial for the build system to only include code into the target binary which is really used by the program logic of the application. It is therefore not advisable to simply link one big library containing all possible combinations of interfaces and bindings and relying on the runtime system to pick the appropriate classes to instantiate, depending on the system state and the application.

Since the dynamic approach doesn't work, we need to statically determine which interfaces to include into the final binary. A good spot to achieve this is the place inside the binding specific code where the decision is taken whether a certain service is offered/ made available by the binding. This place only links to the template instantiations that are potentially needed by the system. Therefore, we only need to generate frontends and bindings which are, according to the AUTOSAR model, used by the component by looking at its ports, which is an instance of a specific interface. A port can either be required, which turns it into a service port, or it can be provided, which means that the application needs a proxy for this interface. Since one interface can be used by more than one port and we can have both at least one required and one provided port, it is also possible that an application needs both a proxy and a skeleton for a specific interface.

To accommodate these requirements, the ara::com generator offers the ability to only generate the files one specific application needs, according to the AUTOSAR model. This will already cater the need to only generate the types that are needed. In addition,

it will also generate a so-called mapping file that is used by the implementation to query and instantiate the proxy and skeleton classes if available. The file consists of two parts: At the top there is a per-interface mapping definition between the internal type ID and the classes that provide skeleton and proxy functionality. The second part consists of two functions that return all valid mappings if given either an internal type ID or a SOMEIP service ID. Notice that, in case of a SOMEIP service ID the respective function returns a list of mappings as there is a 1:n mapping between a SOMEIP service and its ara::com interfaces.

2.2 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
GENIVI VSOMEIP	3.1.14.1 (`https://github.com/GENIVI/vsomeip/tree/3.1.14.1`)	used to provide the necessary functionality for SOME/IP and Service Discovery (SD)	dynamic (shared library)	Mozilla Public License	MPL-2.0
BOOST	1.60.0 (`http://www.boost.org/users/history/version_1_60_0.html`)	used to provide the PropertyTree functionality for E2E	static (header file inclusion)	Boost Software License	BSL-1.0
OpenDDS	3.13.2 (`https://github.com/objectcomputing/OpenDDS/releases/tag/DDS-3.13.2`)	used to provide the necessary functionality for DDS Network Binding implementation	dynamic (shared library)	OpenDDS License	N/A

2.3 Class Overview

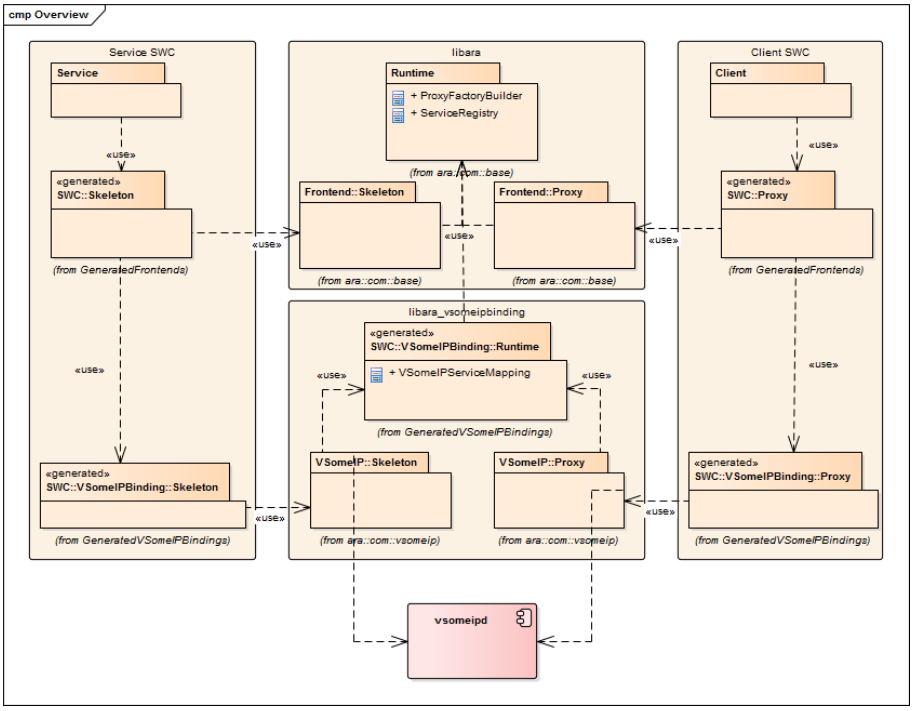

Figure 2.3: Class overview