

AS4 .NET Component – Software Design



# Table of Contents

[Table of Contents 3](#_Toc460996149)

[Document history 4](#_Toc460996150)

[1. Purpose 5](#_Toc460996151)

[2. Layered Architecture 6](#_Toc460996152)

[3. Domain Core Layer 7](#_Toc460996153)

[4. Application Layer 9](#_Toc460996154)

[5. Extension Layer 10](#_Toc460996155)

[6. Frameworks 12](#_Toc460996156)

# Document history

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision** | **Date** | **Created by** | **Short Description of Changes** |
| v1 | 15/08/2016 | Stijn Moreels | Initial version |
| v2 | 30/08/2016 | Stijn Moreels | Spelling Check + Global Annotations |
| v3 | 30/08/2016 | Stijn Moreels | Reformat style |
| v4 | 06/08/2016 | Stijn Moreels | Signing Approach |
| v5 | 12/09/2016 | Stijn Moreels | Update structure document + interaction schema agents |
| v6 | 16/09/2016 | Stijn Moreels | Providers section |
|  |  |  |  |
|  |  |  |  |

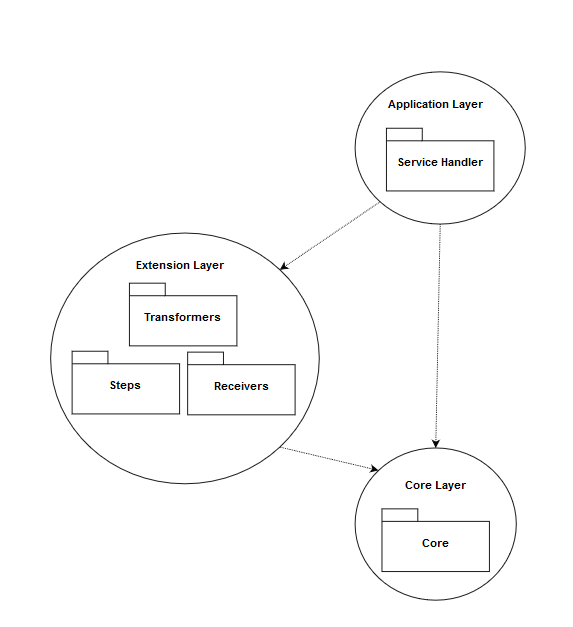
# Purpose

The purpose of this document is to describe the global technical design of the Everis project (a.k.a. AS4 project). Some critical core design points are explained to have a global awareness of the software design of the project.

# Layered Architecture

The AS4 application is separated in several layers. Since the Core is the essential part of the application, it could also be seen as the **Domain Layer** of the project. Like any layered architecture, all the dependencies go in one direction only.

* The **Domain Core Layer** contains core functionality and provides the services (and interfaces) to the upper layers. The core definitions of the AS4 protocol are also describes in here.
* The **Application Layer** is actually the layer that uses the core to provide a system for the presentation layer. This layer contains besides the delegation of functionality also the mappings that’s needed to provide a connection form the Business Application to the MSH.
* The **Extension Layer** contains all the agents that are directly coupled to the Business Applications. The agents can be extended to have a personalized AS4 adapter for your communication. The Steps and *Receivers* are exposed interfaces and used steps inside the core that can be extended by the user.



# Domain Core Layer

**Name:** Domain Core Layer

**Purpose:** In the Domain Core Layer, the core functionality of the AS4 protocol is described.

**Functionality Serializers:** What’s really important is the **Serialization** of the message so it can be send and retrieved from other parties. The message itself (*AS4Message*) has two methods to perform the serialization/deserialization can be called from the *Send Agent*.

The message itself asks at the *Global Registry* to retrieve the right *Serializer* (all have to be registered in this Global Registry). These serializers are registered with the Content Type by which they perform the serialization (MIME or SOAP), this way the client (in this case the Send Agent) has only to know which content type the incoming AS4Message has.

The functionality described in the serializers itself calls somethings external parties (like the *Soap Envelope Builder* or the Mapping functionality described inside the AutoMapper Profiles, See 6 Frameworks) to have the right functionality to perform the serialization.

**UML Schema Serializers:**

**C:\Users\stijn\Downloads\EVERIS(3).png**

**Description Serializers:** The *Global Context* is responsible for the context in which the application runs, this includes the *PMode*configuration, the *Settings* file… Since we use only this single context, there’s only a single truth at a single place where these items can be retrieved for those who needed.

The *DataStoreContext* uses it for the connection string, the *Transformers* use t for the right pmode, the *Receivers* use it for the configuration settings of the *Receivers* (and all these items are found in the settings file or pmode files).

**Functionality Signing:** The Signing is performed by some collection of classes. Inside the *AS4Message*, the property *SecurityHeader* is used to have wrapper for the Security Header inside the Soap Envelope of the AS4 Message.

There’s several parts of the Security Header that has to be constructed, that’s why all these parts all collected inside a *Builder* class to have more control of the creation of the header.

**UML Schema Signing:**



**Description Signing:** There’s need for several *Security Token References* and *Signature Algorithms*. These Abstract classes define an interface for all the needed implementations. Because of this approach, we can easily add/edit/remove specific implementations of *References* or *Algorithms* because the **AS4Message**, **SecurityHeaderBuilder** and the **SecurityHeader** itself all uses the abstract class.

# Application Layer

**Name:** Application Layer

**Purpose:** The application layer is all about delegation and knowing where the functionality is placed instead of having the functionality described itself. That’s why you’d find only the responsibility to create *IAgent* implementations from the settings file (where the different types are described).

So this layer contains the startup and the responsibility of the creation of the Agents.

**UML Schema:**

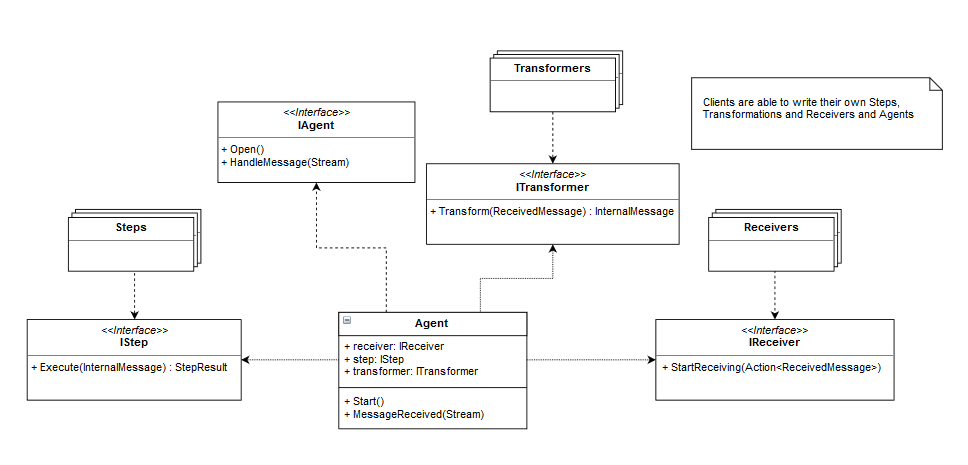


# Extension Layer

**Name:** Extension Layer

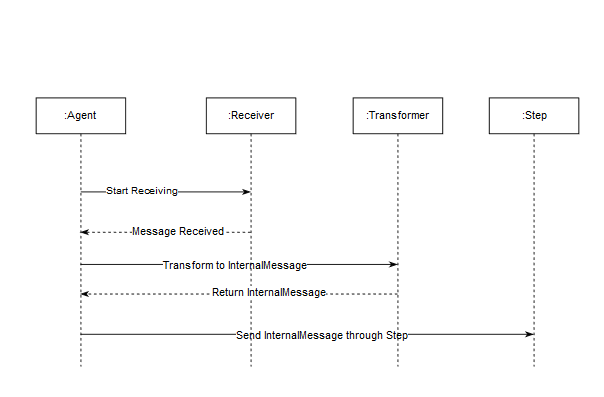
**Purpose:** The whole development process of the project has the *Extensibility* in mind, and this layer for sure. Almost every *Service* has to be extendable for the users: *Agents*, *Transformers*, *Receivers*, *Steps*…

**UML Schema Agents:** The following diagram shows the different relationships between each component. All the interfaces are exposed by the Core and are not part (of course) of the presentation layer, but for readability purposes these are included.



**Description Agents:** When you look at this diagram you see that the *Agents* are actually the central point of the system and wrap all the needed functionality. The *Agent Base* is an abstract class to have a consistent *Open()* method for all its child’s. When clients want to write an *Agent* from scratch, they can use the *IAgent* interface (this interface is used everywhere and not the abstract class).

The idea is to have an *Agent* class which is just responsible of the connection of the *Receiver*, *Transformer* and the *Steps*. This way we have a high-level term if we speak about “Submit”, “Send”, “Deliver” and “Notify” (which are the AS4 protocol actions).

**Interaction Schema Agents:** Because the flow of each agent is the same, there’s no need to have a *Submit Agent* or *Send Agent*; but the *Agent* class is made with the extensibility in mind and so can the chaining of the three different components in the agent be adapted.

## Steps

**Purpose:** The steps are used to perform actions on the AS4 Message, for example: Compressing, Encrypting, Signing…

Each step does something with the message and sends it through to the next step. To have this kind of functionality on a clean manageable way, we use two classes for our Steps: *OutExceptionDecoratorStep* and *CompositeStep*.

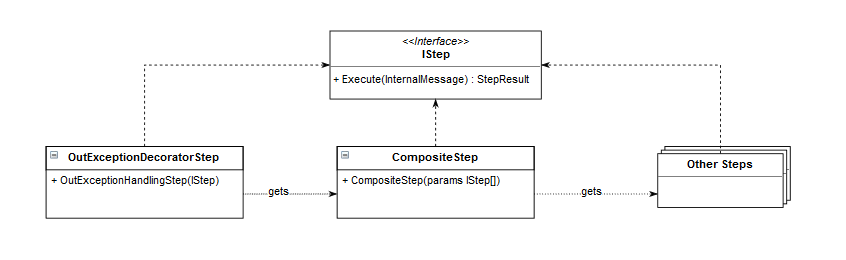
The “Other Steps” is used to define all the other steps (compressing, encrypting, signing…), all these steps gets injected inside the Composite Step (see *Composite Pattern*) which defines the gets/sends mechanism.

The Composite Step itself is *decorated* by the *OutExceptionDecoratorStep* (see *Decorator Pattern*). This step defines the Exception Handling approach for the *IStep* implementations.

By using this mechanism, we can easily add/edit/remove steps and still have the same flexibility of our Exception approach and the composite mechanism.

The exception approach will be explained in later chapters.

**UML Schema Steps:**



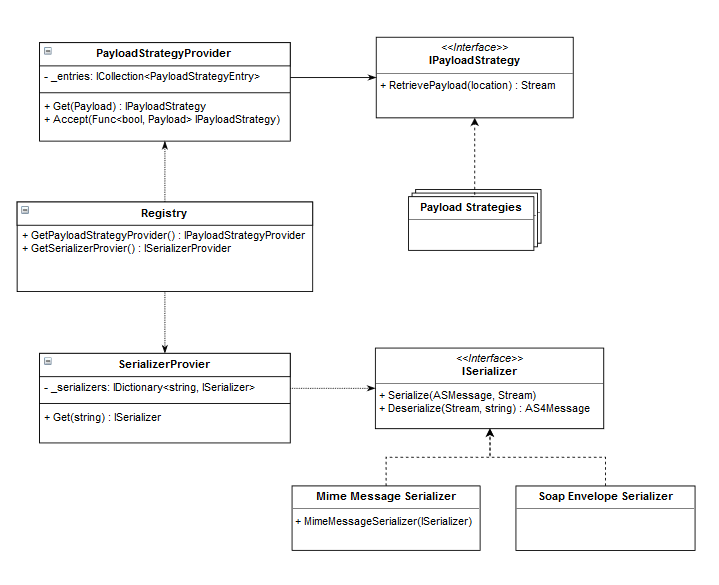
## Providers

**Purpose:** on multiple locations throughout the application, we need to perform a search based on run-time information. An example are the Payloads. Every payload has a location and may need another way to retrieve the actual payload stream.

The problem is because of this scenario, we cannot inject at compile-time the needed dependency, so we need some kind of registration that can give us the right dependency.

The solution is a combination of two patterns: *Strategy* and *Registry Pattern*.

**UML Schema Providers:**



In this scheme we have two important concepts that are combined together: The *Strategy* interface (*IPayloadStrategy* and *ISerializer*). Each kind of strategy has a *Provider* which is responsible of the distribution of the implementations of the strategies. The *Registry* is responsible for the distribution of the different *Providers*.

This way we have a three-way-layered system (*Strategy – Provider – Registry*) which decouples the dependencies.

## Receivers

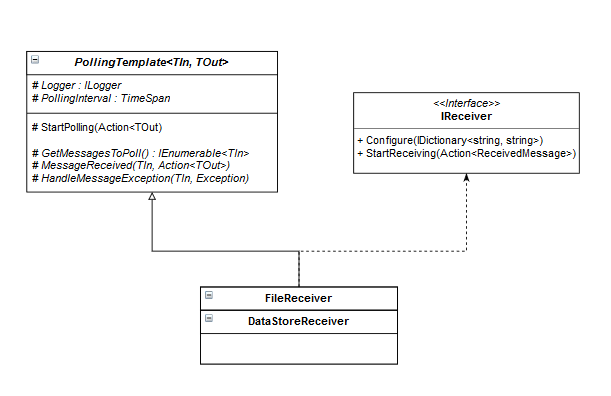
**Purpose:** Exception can occur on different places and must therefore be handled differently. Although, some handling actions are similar and that’s the reason why we extracted these actions inside the *OutExceptionDecoratorStep*.

This step updates the Data Store with the given exception information, and that action is the same over the whole component. The exception that are thrown and are related to the *Receiver*, are also handled by the *Receiver* (place file on file system, send HTTP message to extern party…).

**Schema Receivers:** Following diagram shows how this approach is being implemented. *Receivers* can make use of the *PollingTemplate* (see *Template Method Pattern*) to have the polling functionality included inside its implementation as *Receiver*.

If there’s some exception occurred, the *PollingTemplate* will execute the *abstract* method that has to be implemented by each *Receiver*. This way each implementation of a *Receiver* can handle its own handling functionality. This way we have completely separated the receiving messages and exception handling functionality from the polling functionality; and has the *PollingTemplate* none connection with the *Receivers*.

**UML Schema Receivers:**



## Transformers

**Purpose:** Transformers are used to “transform” the *ReceivedMessage* (coming from the *Receives*) to an *InternalMessage* (used inside the *Steps*).

The question arises what the difference is between *Serializing* and *Transforming* and the fact that we create an *AS4Message* from a *SubmitMessage* inside the *Steps* and not as *Transformer*. The reason is the following:

* **Transformers**are used to transform to an *InternalMessage*. The source of this transformation is a Stream which contains whatever format of the given message. It’s the responsibility of the *Transformer* that we have an *InternalMessage* afterwards.
* **Steps** are responsible for manipulating the *InternalMessage*. When we want to send it as MIME or SOAP, or we want to create an *AS4Message* from a *SubmitMessage*, or want to Sign/Encrypt/Compress our message… that’s all possible and is all in the scope of Manipulating the *InternalMessage*. The reason why we allow creation of *AS4Messages* is because *SubmitMessages*, *AS4Messages*, *DeliverMessags*… are all internal messages (Messages conform to the AS4 standard).

# Frameworks

The following frameworks are used to outsource certain tasks:

* **Name:** AutoMapper  
  **Reason:** theirs need for several mappings between objects. When messages arrive at the agents, when the messages leave the agents… **AutoMapper** is used to perform these mappings in a clean way. All *Profiles* (mapping schema for a mapping between two types) are using the same naming convention: *[name of the mapping] Map*.
* **Name:** MimeKit  
  **Reason:** for the MIME serialization of the AS4 Messages, we use **MimeKit** to perform these actions. This framework can parse incoming streams to *MimeMessages* objects; which then can be used to reassemble the original AS4 Message.
* **Name:** Entity Framework Core  
  **Reason:** we have to use an abstraction layer for our datastore, that’ why we use **Entity Framework Core** for this task. This framework allows us to create an *Anticorruption Layer* between the actual dataset (which can be of any provider) and the Domain Model. It also has functionality to store the data in memory; which is very useful in testing scenarios.
* **Name:** NLog  
  **Reason:** the project contains several Exception Strategies so logging is also a required functionality. All the loggings are centralized by the **NLog** framework. This framework is called by the *Singleton* that the framework provides.
* **Name:** Fluent  
  **Reason:** when messages arrive or leave, some validations are required. To have a solid and future-proof system, we’ll use **Fluent** to validate each object. This framework requires and apart class to perform the validation.
* **Name:** Moq  
  **Reason: Moq** is used to mock the dependencies of classes. This framework allows us to test multiple execution paths throughout the application.
* **Name:** xUnit  
  **Reason:** For the Unit and Integration Tests, the Framework **xUnit** is used. This framework has a lot more flexibility than the standard MSTest library and is practically useful when you want to reuse tests (with theories and auto-data for example). All tests follow the convention:
  + *Given[name of the component to test/name of the test]Facts* as class name
  + *Then[path to test]Succeeds/Fails* for methods of that class. This approach helps in tracking the tests.