

AS4 .NET Component

Software Design



# Table of Contents

[Table of Contents 3](#_Toc467162130)

[Document history 4](#_Toc467162131)

[1. Purpose 5](#_Toc467162132)

[2. Glossary 6](#_Toc467162133)

[3. Layered Architecture 7](#_Toc467162134)

[4. Domain Core Layer 8](#_Toc467162135)

[4.1 Serializing 9](#_Toc467162136)

[4.2 Security Header: Signing 10](#_Toc467162137)

[4.3 Security Header: Encryption 11](#_Toc467162138)

[5. Application Layer 12](#_Toc467162139)

[6. Extension Layer 13](#_Toc467162140)

[6.1 Settings 15](#_Toc467162141)

[6.3 Agents 16](#_Toc467162142)

[6.3.1 Submit Agent 16](#_Toc467162143)

[6.3.2 Send Agent 16](#_Toc467162144)

[6.3.3 Receive Agent 17](#_Toc467162145)

[6.3.4 Deliver Agent 17](#_Toc467162146)

[6.3.5 Notify Agent 18](#_Toc467162147)

[6.4 Steps 19](#_Toc467162148)

[6.4.1 Submit and Send Steps 19](#_Toc467162149)

[6.4.2 Receive Steps 20](#_Toc467162150)

[6.5 Providers 21](#_Toc467162151)

[6.6 Receivers 22](#_Toc467162152)

[6.7 Transformers 23](#_Toc467162153)

[6.8 Senders, Uploaders and Retrievers 24](#_Toc467162154)

[7. Infrastructure Layer/Datastore Layer 25](#_Toc467162155)

[8. Integration Tests 26](#_Toc467162156)

[8.1 Purpose 26](#_Toc467162157)

[8.2 Implementation 26](#_Toc467162158)

[8.2.1 Before: Cleanup 26](#_Toc467162159)

[8.2.2 Arrange: Start Application 26](#_Toc467162160)

[8.2.3 Act: Send Message 26](#_Toc467162161)

[8.2.4 Assert: Notify Message 27](#_Toc467162162)

[8.2.5 After: Stop Application 27](#_Toc467162163)

[8.2.6 Template 27](#_Toc467162164)

[9. Code Style Guidelines 28](#_Toc467162165)

[9.1 Code Structure 28](#_Toc467162166)

[9.2 Comments 28](#_Toc467162167)

[9.3 Class Names 28](#_Toc467162168)

[9.4 Function Names 28](#_Toc467162169)

[9.5 Abstraction Level 28](#_Toc467162170)

[9.6 Tips 28](#_Toc467162171)

[10. Frameworks 30](#_Toc467162172)

[11. Refactoring Notes 31](#_Toc467162173)

[11.1 Security Header to Sign Strategy 31](#_Toc467162174)

[11.2 Blown-up Decorator to Undecorated Steps 32](#_Toc467162175)

[11.3 Duplicate Sign Functionality to single Singing Step 32](#_Toc467162176)

# 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 |
| v7 | 19/09/2016 | Stijn Moreels | Add Glossary |
| v8 | 27/09/2016 | Stijn Moreels | Add Receive Steps |
| v9 | 07/10/2016 | Stijn Moreels | Refactoring Notes for the Security Header to the Sign Strategy |
| v10 | 31/10/2016 | Stijn Moreels | Adding Release Notes, Polly Framework and Encryption Strategy |
| v11 | 31/10/2016 | Stijn Moreels | Rename class names and replace UML Schemas |
| v12 | 9/11/2016 | Stijn Moreels | Adding Integration Test documentation |
| v13 | 17/11/2016 | Stijn Moreels | Adding Release notes, formatting guidelines |

# 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.

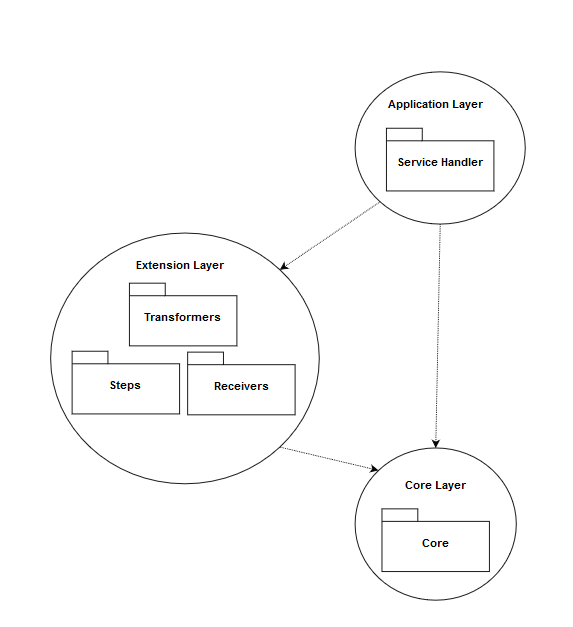
# Glossary

|  |  |  |
| --- | --- | --- |
| **Name** | **Domain Type** | **Description** |
| **Agent** | Service | Agents contains 1 *Receiver*, 1 *Transformer* and 1 *Step* (but the most of the time the *Step* is a *Composite*) |
| **AS4 Message** | Entity | AS4 Messages are an object presentation of a send AS4 SOAP/MIME message. |
| **Internal Message** | Entity | Internal Messages are used throughout the application and contains 1 *AS4Message*, 1 *Submit Message* |
| **Provider** | Repository | Providers are objects which provides services throughout the application. (ex. *SerializerProvider*) |
| **Received Message** | Entity | When a *Receiver* received something, the form in which it returns it is a *ReceivedMessage*. |
| **Receiver** | Service | A *Receiver* can be configured to listen on anything and returns a *ReceivedMessage*. |
| **Serializer** | Service | There’s two serializers present: a SOAP and MIME Serializer. This serializers serialize an *AS4Message* to a Stream. |
| **Step** | Service | A *Step* is used to describe a single operation that has to be performed by an *Agent*. (ex. *EncryptAS4MessageStep*) |
| **Submit Message** | Entity | A *SubmitMessage* describes the form in which the *SubmitAgent* will receive a message. |
| **Transformer** | Service | *Transformers* transform an *ReceivedMessage* to an *InternalMessage*. |

# 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

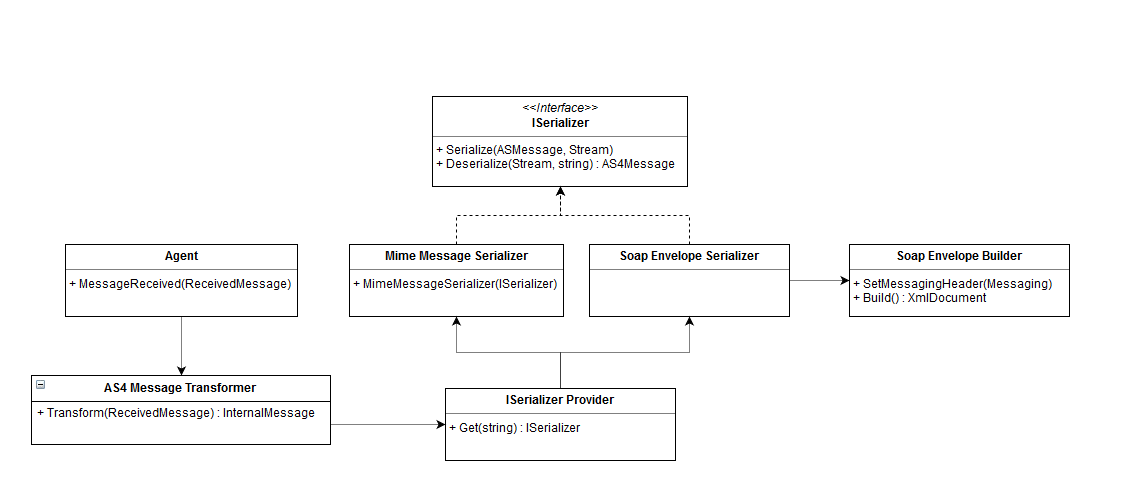
**Purpose:** In the Domain Core Layer, the core functionality of the AS4 protocol is described.

## Serializing

**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 6.5 Frameworks) to have the right functionality to perform the serialization.

**UML Schema Serializers:**

## Security Header: Signing

**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.

The actual signing is performed inside the *ISigningStrategy* which can be inserted in the *SecurityHeader*.

**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**, **SignStrategyBuilder** and the **SignStrategy** itself all uses the abstract class.

## Security Header: Encryption

**Functionality Encryption:** The actual work is performed in the *IEncryptionStrategy* (the implementation inherit from *EncryptedXml*).

The strategy uses several other data classes to encrypt the *AS4Message*. The only part of the message that will be encrypted, are the *Attachments* send with the message. No attachments, no encrypted message.

**UML Schema Encryption:**



**Description Encryption:** the *SecurityTokenReference* classes can be reused in the *Encryption* strategy. There’s a *Builder* which’s responsible for the creation of the strategy.

# 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:**

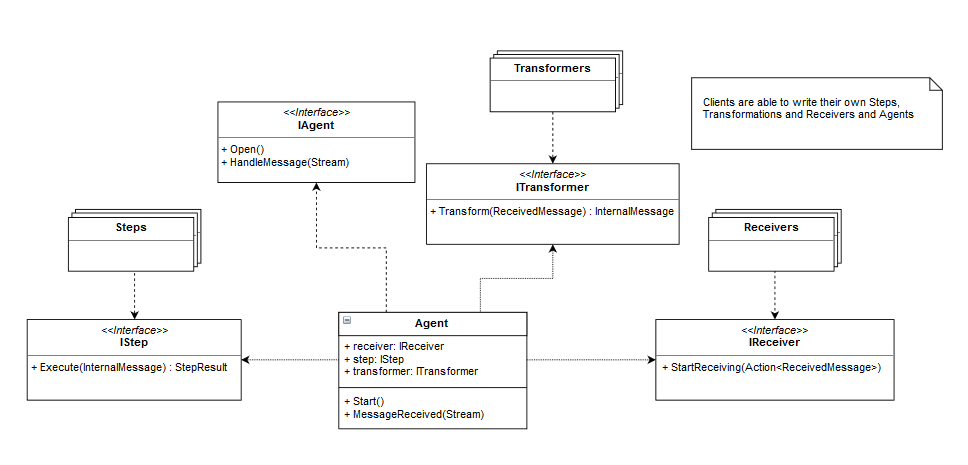


# Extension Layer

**Name:** Extension Layer

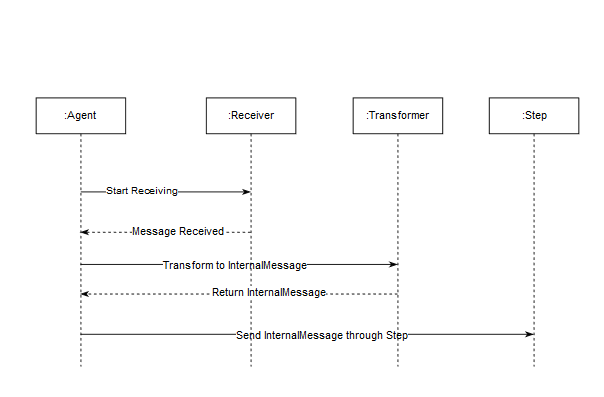
**Purpose:** The whole development process of the project has the *Extensibility* in mind, and this layer for sure. Almost every *Service* has to be extendable for the users: *Agents*, *Transformers*, *Receivers*, *Steps*…

**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.

## Settings

The *Config* 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).

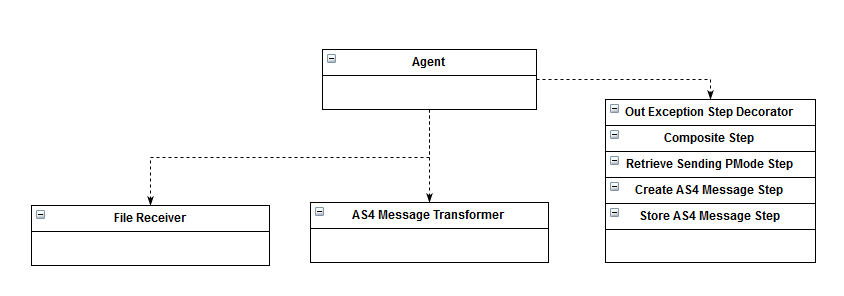
Some custom settings can be added to this file as well. This can be credentials for example.

## Agents

The following items describe the high-level composition of the agents.

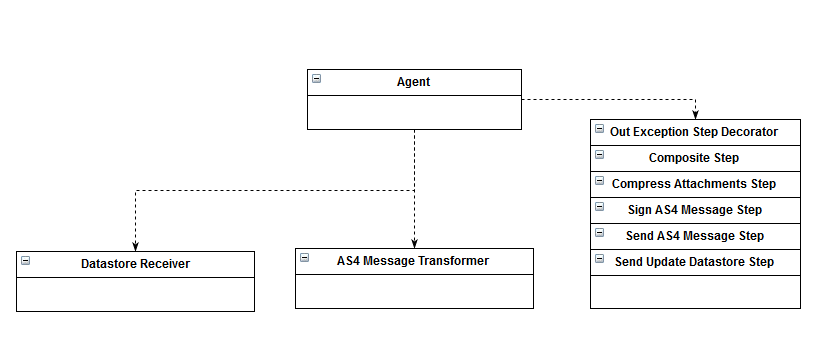
### Submit Agent

The Submit Agent polls on a configured endpoint and stores the transformed message in the datastore.



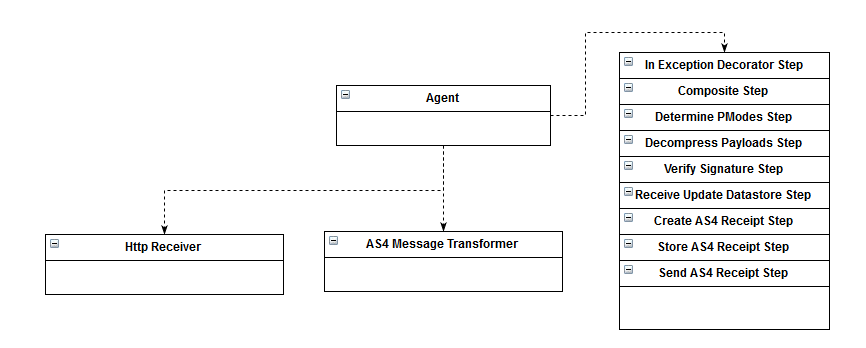
### Send Agent

The Send Agent polls on the Datastore and adapts the message before it sends it to the configured party.



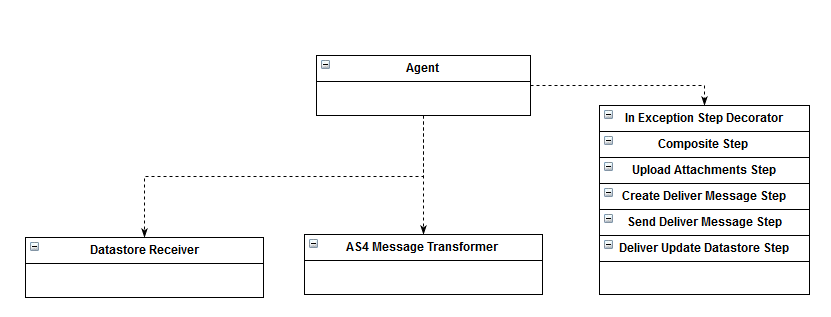
### Receive Agent

The Receive Agent gets the send message to the component and sends a receipt to the requested party.



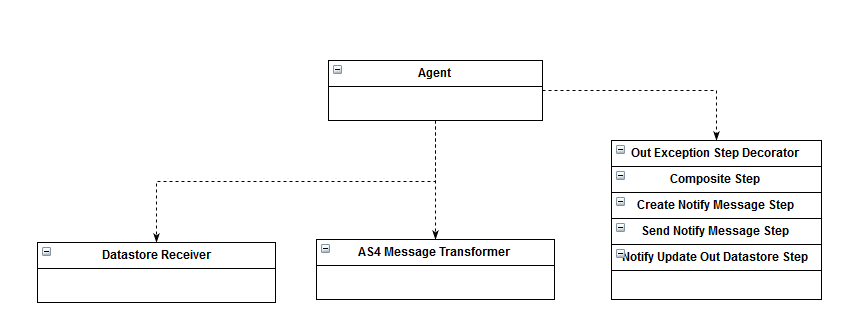
### Deliver Agent

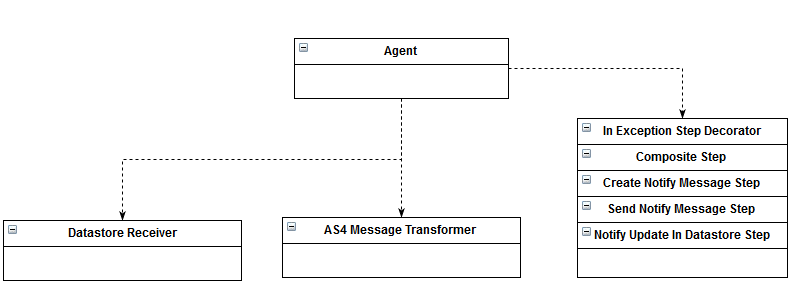
The Deliver Agent polls on the Datastore and delivers the incoming messages to a configured endpoint.



### Notify Agent

The Notify Agent polls on the Datastore and notifies the Business Application for incoming and outgoing messages.





## Steps

### Submit and Send 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*.

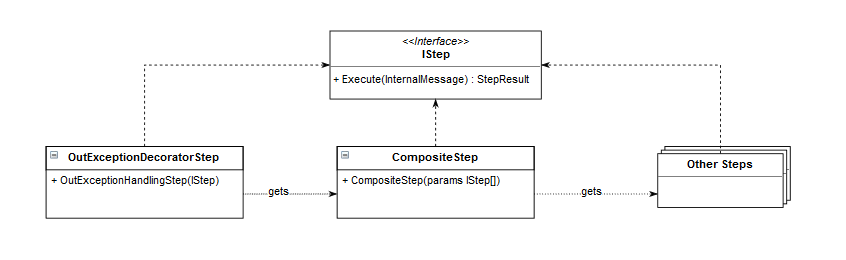
The “Other Steps” is used to define all the other steps (compressing, encrypting, signing…), all these steps gets injected inside the Composite Step (see *Composite Pattern*) which defines the gets/sends mechanism.

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:**



### Receive Steps

**Purpose:** the steps for the Receive agent are the reverse of the ones for the Submit/Send. In this case, we need to Decompress, Decrypt and Verify the incoming message. There’s also need for an *ErrorHandlingDecoratorStep* which handles the exceptions for the executed steps.

Besides that, we need to send an answer to the party who’s sends the request. This can be *Sync* and *Async* performed. Sync means that we will send the response to the same party that sends the request; Async means that we will use a different endpoint to send the response to. The requested party will get an empty SOAP body instead.

As you can see, there’s always need for an answer to the requested party (Receipt or Empty SOAP). That’s why a communication mechanism is needed between the *Steps* and the *Receiver*. At first it seems logical to create a *Sender* which is responsible for these two options, but after further analysis the *Receiver* should be responsible for the handling of its own receiver-specific configuration. It would be wrong to split this knowledge into two classes.

The *HttpReceiver* will receive a message and send it to the *Agent*. The *Agent* will send it through its given *Steps*. The *Steps* are composed with two *Decorators*: one for the exception handling and one for the *Receipt* handling. In this last one the decision is made whether the *Receipt* must be send back to the requested party or to another endpoint.

The *Receipt* itself will be created in the very last *Step* and is send as *StepResult* back to the *HttpReceiver*. In case of an error the same approach can be made as for a *Receipt* but instead of a *Receipt* is being created in the *CreateAS4ReceiptStep*, an *Error* is being created and send to the requested party through the *Receiver* OR via an extern endpoint and an empty SOAP body will be send back to the *Receiver*.

The *Undecorated Steps* will only be executed when theirs an exception being thrown during the execution of the *Decorated Steps*. Both *Receipt* and *Error* can be signed, so the same step like the *Send* operation is being used. (the settings.xml defines the *Decorated* and *Undecorated Steps*).



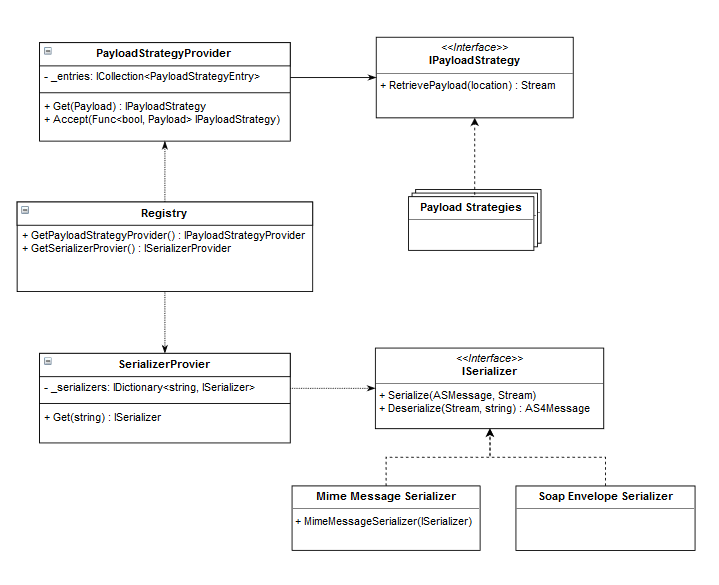
## Providers

**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 registration that can give us the right dependency.

The solution is an *Abstract Factory* with the exception that the models are created only once.

**UML Schema Providers:**



In this scheme, we have two important concepts that are combined: 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. The *Providers* will also be used in the selection of *Senders*, *Uploaders* and *Retrievers*.

## Receivers

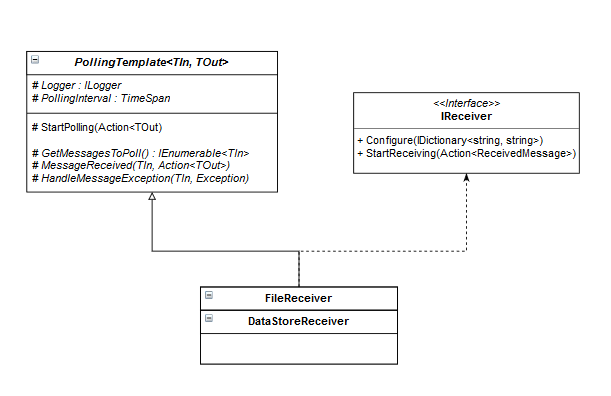
**Purpose:** Exception can occur on different places and must therefore be handled differently. Although, some handling actions are similar and that’s the reason why we extracted these actions inside the *OutExceptionDecoratorStep*.

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 must 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).

## Senders, Uploaders and Retrievers

The *Deliver* and *Notify* operation has both *Steps* which will “send” the *Deliver/Notify Message* to an external endpoint. As implementation example, an implementation for the *Senders* is being created which will “send” the message to the file system (in a specified directory configured inside the *Sending/Receiving Pmode)*.

The *Submit* operation is responsible for assembling and *AS4Message* ready to send. To do this, we need the *Attachment* contents. That’s where the *Retrievers* come in. As implementation example, and implementation for the *Retrievers* is being created which will “retrieve” the *Attachment* content from the file system (the location of the attachment defined in the *SubmitMessage* will be used to determine which *Retriever* must be used).

The *Deliver* operation will also “upload” the *Attachments* to a given location. The *Uploaders* are responsible for this task. As implementation example, an implementation for the *Uploaders* is being created which will “upload” the *Attachment* content to a given location (configured in the *Receiving Pmode)*.

Why have we explained these three elements in the same section? Because each element is configured the same way. Each element will have at *Type* which is used inside the *Pmodes* to determine which implementation must be used. This way, a *FTPAttachmentRetriever* can be used if you define your own *Type* (*FTP* for example) and use this in the configured *Pmode* where you want to retrieve your *Attachments* from a FTP server.

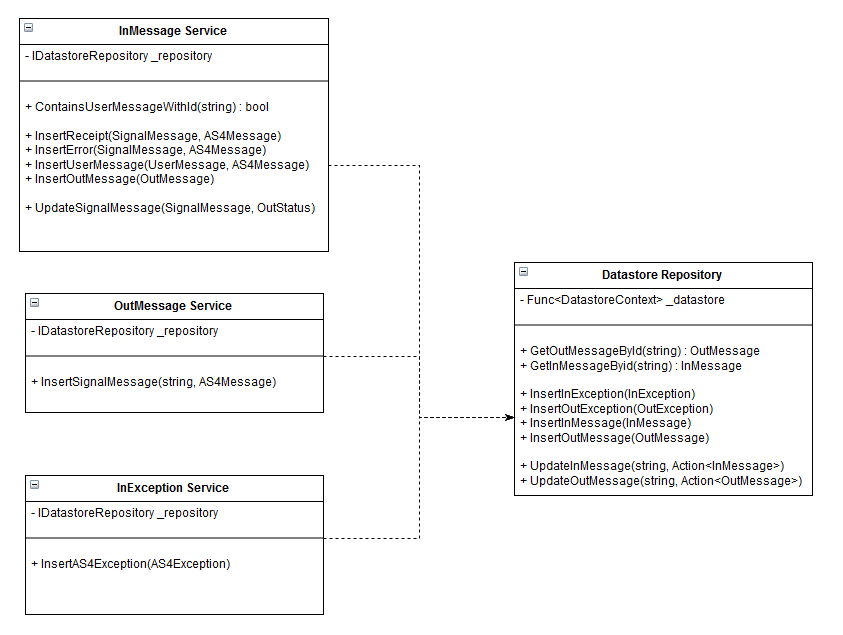
The same can be said for the *Senders* and *Uploaders*.

# Infrastructure Layer/Datastore Layer

The Datastore itself needs to be abstracted from the concrete type of datastore (SQL Lite, SQL Server, Mongo DB...). That’s why we used **Entity Framework Core** for this task. We created an extra infrastructure layer for the **DbContext** (Entity Framework Core Model) and added a single **Repository** which is used in multiple **Services**.

The Repository is used to define basic operations on the Datastore (GET, UPDATE …). The Services are used to have a high-level use of the Datastore and call operations like: *InsertAS4Exception, UpdateErrorSingleMessage*… without the caller knows which datastore, table or another concrete infrastructure is needed to perform the operation.

Because of the combination of Repositories and Services, we can reuse almost everything in multiple *UpdateDatastoreStep(s)*. To be complete: The Repositories contains no Domain logic, only Infrastructure knowledge; the Services knows basic Domain logic and reference the Repository.



# Integration Tests

## Purpose

Four different categories are used to implement our Integration Tests:

* Positive Send
* Negative Send
* Positive Receive
* Negative Receive

**Positive Send** describe all different kinds of sending valid messages but with different configuration (with/without compression, signing, encryption, NRR…).

**Negative Send** describe all different kind of sending invalid configured messages (missing Pmode, non-existing certificate…) or a misconfigured component.

**Positive Receive** describes all different kind of receiving valid messages but with different configuration (with/without compression, signing, encryption, NRR…).

**Negative Receive** describes all different kind of receiving invalid configured messages (missing Pmode, non-existing certificate…) or a misconfigured component.

## Implementation

We thread every Integration Test as black box and assert on the notify, receipt, error or payload messages.

Following structure describes a skeleton of how each integration test is structured:

### Before: Cleanup

Each integration test has its own Pmode(s) and message(s) it sends or receives. The beginning of each test starts with the cleanup of all these files. **Holodeck** will be configured with a new Pmode, all the files in the notify-folders will be deleted, all the accepted send/receive-messages will be deleted…

Sometimes an Integration Test uses the same Pmode or message to send as the test before, but to have a clear separation and no confusion, we use always a different Pmode or message. This way we’re sure that each Integration Test is decoupled (as much as possible).

### Arrange: Start Application

In this phase, we will start an instance of our component which will be used later in the Integration Test. Each Integration Test starts a new instance so we’re sure we haven’t any previous errors and start from scratch.

### Act: Send Message

This phase is responsible for the sending of the message. This can be a **copy** operation that will place a submit message in the \messages\out-folder or a holodeck message that is placed inside the \data\msg\_out-folder. But sometimes this phase is custom code and will send (for example) a wrong signed message to a receiving MSH.

### Assert: Notify Message

After we send the message, we must poll/wait for the message to arrive at the receiving MSH and assert on the message that will be notified. This can be on our end or at holodeck’s.

This phase will take most of the time, because the message must go through the sending and receiving MSH and must be notified.

### After: Stop Application

After the assertion is completed, the component instance must be stopped so it doesn’t conflict with the next Integration Test.

We have built a safety mechanism inside our Build Definition that will try to stop any still-running instance of our component (before and after the build runs).

### Template

Each Integration Test has the same structure it uses, that’s why an *IntegrationTestTemplate* is created which will provide some basic functions:

* Cleanup
* Start/Stop Application
* Poll To

Besides this template, we also created a *StubSender* and *Holodeck* class. The *Sender* will be used in situation where theirs need to send an invalid message (or duplicate), the *Holodeck* class is an interface that assert holodeck-related items (receipts for example).

This way the test itself only contains specific configuration for that scenario and not the common tasks that every test will use.

# Code Style Guidelines

Following sections describe the used formatting guidelines for the implementation. Most of the guidelines are [Microsoft](https://msdn.microsoft.com/en-us/library/ff926074.aspx) based. Only a few are personal flavor.

## Code Structure

Following code structure is used when defining classes:

* Fields (order by private, internal, protected, public)
* Properties (order by private, internal, protected, public)
* Constructors (default first then order by complexity)
* Methods (“simple”, “small” methods first, then by call. Try to place it as close to the caller as possible)

Read only, constants and static fields are place before the actual items.

## Comments

Each (not model) class has a summary block at the beginning of the class describing the responsibility of that class. Each public method has also this summary, public properties are not summarized (yet).

No comment-out code is committed in source control! Besides the responsibility comments, we don’t try to use comments.

## Class Names

When applying a Design Pattern, we always add the name of the pattern to the class name. The name of the class itself is always a noun.

Unit test classes uses always the same name construction: **Given** Class Name **Facts**.

## Function Names

Names of functions is always a verb. Try to add in the name of the function the reason why you send arguments or return arguments (if any); so, it makes sense to send or retrieve something from the function.

Unit test functions uses always the same name construction: **Then** Name of Test Case

## Abstraction Level

As always we try to have a clean code architecture with no mixture of abstraction levels. Functions should (for example) be logically constructed and not on multiple levels exists.

## Tips

As final topic, some tips:

* **Private as possible:** try to hide everything at the beginning and start opening if there’s need to;
* **Classes/Functions should be short:** a class with over 300 lines is too much, a function over the 30 lines is also kind of long;
* **Three parameters max**: in every function and class, we use maximum three parameters.
* **Column With:** 120 lines is used as guideline for the column with;
* **Try-Catch convention:** when writing a try-catch block, start your method with **Try** Name of the method. In the Try block, write the method without the “try”. This way of writing is used around the application;
* **Two indentations:** maximum used amount of indentations in method is **2**. So, a combination of a foreach and if statement is two for example.

# 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:** Polly  
  **Reason:** to have a short auto-retry mechanism for the datastore, we implemented this with the **Polly** framework. This framework allows us to define *Policies* for which a given method must be retried.
* **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 and finding the tests.

# Refactoring Notes

## Security Header to Sign Strategy

When creating the Security header for the Signing, we didn’t involve any encryption specification what leads us to a refactoring challenge when we must insert the encryption functionality.

What we must do is refactor the *SecurityHeader*(which now contains signing functionality and knowledge of the Security Header Envelope tag) so it only contains specifics about the Security Header Envelope tag itself. All the signing knowledge must be extract into a new class to which the *SecurityHeader* will delegate.

Following steps are taken in this order to refactor this in a secure way:

1. Create a *SignStrategy* implementation which contains the signing functionality of the *SecurityHeader* (normally we would copy this file from Source Control, but because the team is so small and there’s no history needed we’ve done this local).
2. *GetSecurityXml* in the *SecurityHeader* becomes *GetXml* and creates the security tag inside the *SecurityHeader* and the signature and references will be added later in the *SignStrategy*.
3. Create a *SignStrategyBuilder* implementation the signing functionality of the *SecurityHeaderBuilder*
4. Add a property in the *SecurityHeader* to which the builder will delegate the *Build Parts*
5. Extend the *Serializers* to also serialize/deserialize the *SignStrategy*
6. Extend the *SecurityHeader* to delegate Verify and Sign to *SignStrategy*
7. Extend *SignAS4MessageStep, CreateAS4ReceiptStep* and *ReceiveExceptionDecoratorStep* to also create a *SignStrategy* implementation
8. Delegate extra exposed info from the *SecurityHeader* to the *SignStrategy*.
9. Remove unneeded functionality inside the *SecurityHeader*
10. Delegate signing functionality to the *SignStrategy*
    1. *SecurityHeader.Sign(ISignStrategy)*
    2. *ISignStrategy.Sign()*
    3. *SignedXml.ComputeSignature()*)

## Blown-up Decorator to Undecorated Steps

The settings.xml defines several *IStep* implementations that needs to be decorated. This “decoration” is used to have a solid-exception/error handling system when executing the steps. In the *Receive* operation, an *Error* must be returned to the requested party if there’s any *Exception* thrown. This *Error* can be signed if the *Pmode* defines this.

The whole functionality to update the datastore with the *Exceptions* and *Errors*, creating an *Error* message AND add the *Signing* functionality was just too much for a single *Decorator* to perform.

We must expand the settings.xml declaration to define *IStep* implementations that doesn’t need to be part of the“decoration” (defined with an “**undecorated**” attribute). This way we can define our *CreateAS4ErrorStep* and reuse the *SignAS4MessageStep* **after** the “decorated” steps so they can be executed when an *Error* must be send.

## Duplicate Sign Functionality to single Singing Step

Several messages needed to be signed. User- and Signal Messages. In a first approach, the *Receive* operation would also create a *SigningStrategy* which would also be configured the same way the *SignAS4MessageStep* is implemented.

It was only logical to reuse this step in several other scenario’s. To do this, we would need a *SendingProcessingMode* in the *Receive* operation instead of a *ReceivingProcessingMode*. The reason is because the step uses the *Security.Signing* options configured in the *SendingProcessingMode*.

To solve this problem, we reference a *SendingProcessingMode* in the *ReceivingProcessingMode* so we can use not only the signing configurations but also the *Pull/PushConfiguration* when the *ReceivingProcessingMode* is configured with a *Callback* response.

Another reason why we chose this solution is that we’re sure that when we enter the *SignAS4MessageStep* we have created the same environment as we would have in the *Send* operation (when signing *UserMessages*).

The result is that we have a single signing step which is used to sign *User*-, *Receipt-* and *Error* *Messages* without duplicating any signing functionality.