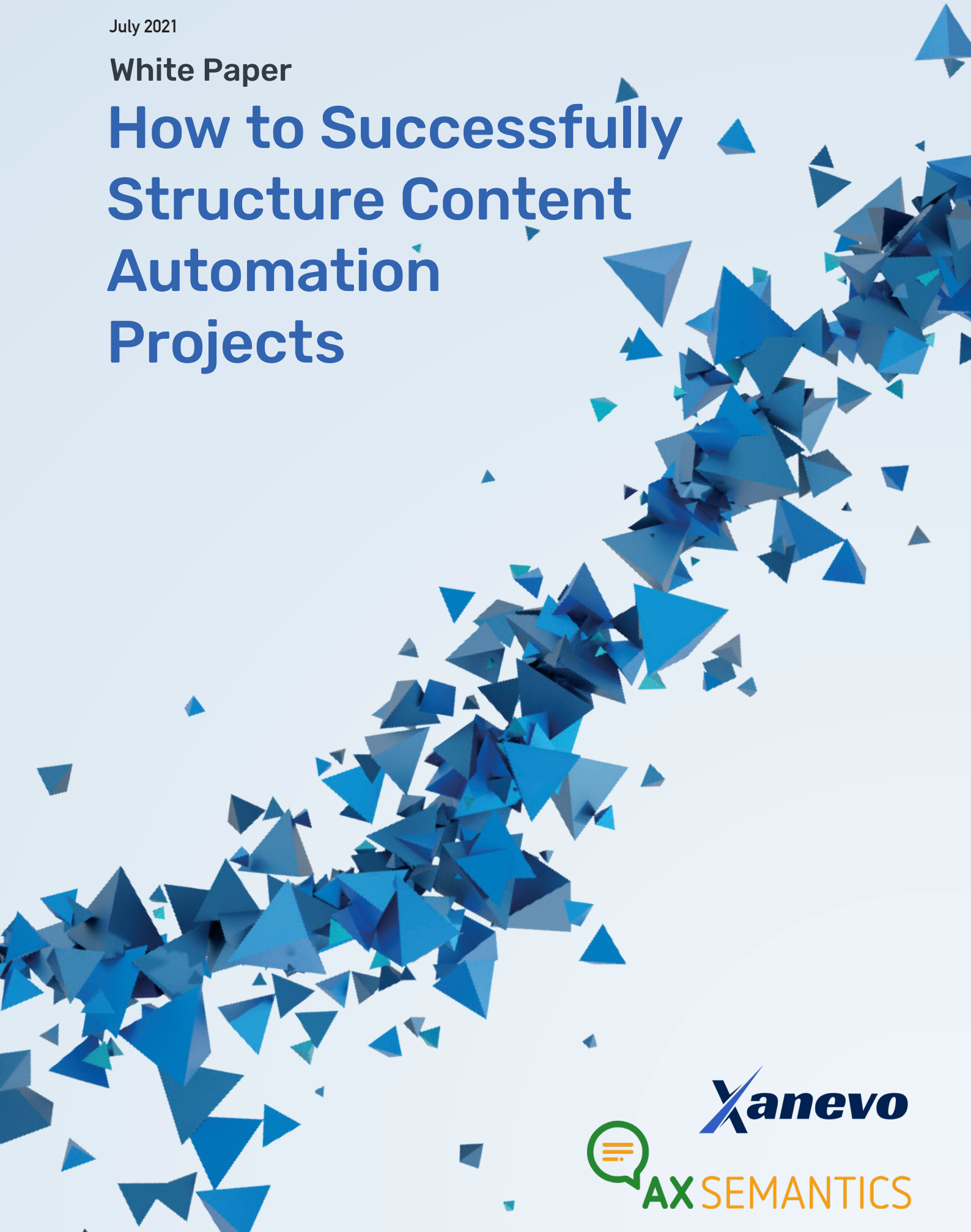
July 2021

White Paper

How to Successfully Structure Content Automation Projects



Xanevo

AX SEMANTICS

Abstract

Automating the generation of content by using low-code software tools, such as AX Semantics' "AX NLG Cloud", has a lot of convenient advantages. For example, these tools usually include an intuitive user interface as well as simple methods for ingesting, transforming and exporting data. As a result the user does not necessarily need to have a professional software engineering background in order to create meaningful software solutions.

Specifically, AX NLG Cloud enables you to create dynamic text results in natural language, such as product descriptions, newsletters or reports. This leads to less manual labor, higher personalization and more website traffic among many more benefits.

A common issue among most low-code tools is that they are usually not designed to comply with coding principles, which leads to shortcomings in code quality and project maintainability.

Applying software engineering methodology to content automation projects can be tricky. We have developed an architecture framework we call OSI (Organize-Serve-Implement) based on the five SOLID principles.

Implementing OSI in content automation projects is especially beneficial when working with multiple developers or when generating more than one language. Time savings and faster development cycles in larger projects are some of the advantages that come with it.

Table of Contents

1. Introduction

In the course of developing content automation projects within AX Semantics' NLG Cloud, there are several steps towards a finished project. These consist of data preparation and analysis, followed by the design and writing of texts, which are then enriched with dynamic content from various data collections.

Providing this dynamic content, which happens in the graphical user interface (TRANSFORM area), can become increasingly confusing for larger projects that process more data and contain more logical functions. This makes it harder to apply changes in later stages of a project and very challenging to hand over tasks to someone else, also because most AX practitioners have their own way of coding and organizing nodes.

At Xanevo, an AX Semantics MSP (Managed Service Provider), we have also experienced these issues and have developed a software architecture methodology to address these inconveniences.

The goal of our proposed methodology is that you will be able to create and **maintain a clear project architecture** throughout your future content automation projects.

Our project strategy is to **divide rich content** into single sentences or paragraphs in order to implement them as **separate programming components**.

This strategy comes along many benefits, some of which are listed below:

- Time savings
- Structured project planning
- Easier budget calculation
- Reproducibility
- Reusability

The following chapters will cover:

1. **Advantages of having a well laid out software architecture**, especially in terms of stakeholder communication
2. **An explanation** on what **SOLID** principles are and how we adapted these from object oriented programming (OOP) to AX NLG Cloud
3. **Our 3-Layer OSI (Organize-Serve-Implement) architecture** in theory and why you should start implementing your projects like we do, along with **real project examples** in which we used OSI (product descriptions in the fashion industry)
4. **Advantages and disadvantages of our architecture** to solidify your understanding on when to use OSI and when to adapt otherwise

Helpful Prerequisites

The paper is certainly the best to follow if you already have some experience with the NLG Cloud. But otherwise it is also possible to focus more on the theoretical chapters (SOLID, OSI) as well as the advantages and disadvantages of the proposed OSI architecture. This will give you a firm understanding of how to approach projects at greater scale and complexity, without damaging key programming principles.

As technical knowledge level is assumed that you have already implemented a content automation project by yourself or in a team. With that you should at least have an intermediate understanding of:

- How to use lookup table, phrase, group, mapping, condition, data and most importantly subgraph nodes
- How variables within TRANSFORM can be created and used in WRITE

You Should Adhere to a Software Architecture - here's why.

There are several good reasons that certainly overwhelm the few disadvantages that come along with having a clean laid out software architecture.

1. Communication

You benefit from a **common understanding of the same structures and rules** within your own team or even across different project teams when working on joint projects.

That leads to having grounds for **estimating how long certain tasks or projects are going to take** since architectural components are comparable, hence past experience helps in estimating time and effort of new tasks.

Rules lead to consent and consent makes it easier to make decisions and to agree upon design choices within a project.

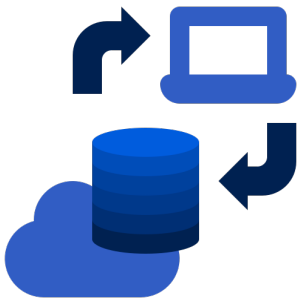
2. Design & Planning

Architecture guidelines **accelerate the initial project planning** phase.

By planning early, the project manager or solution architect is forced to think about **possible risks as early as possible**. This way, project risks can be addressed before becoming a big threat.

Any project can be designed in a way that substructures it into separate modules/components which enables **task pipelining**. This enables a **more agile project setting** as there are less dependencies and more flexibility in assigning tasks or reviews.

3. Transferability



Similar project structures lead to **comparability** and of course **transferability**. Components will be reusable across projects or within a project itself.

Team resources can be allocated more freely since developers will be acquainted with any project's structure.

We will get into more detail on how to achieve a clean project architecture.

2. SOLID Principles

There is a set of principles within the software engineering domain which have been developed to ensure higher quality standards in object oriented programming languages. These so-called SOLID principles are one of many quality principles for software architectures, but probably most commonly known and used across the globe.

This chapter's goal is to look at how we can apply SOLID to AX NLG Cloud.

Important to note:

The SOLID principles and any other software guideline are quality frameworks that require disciplined implementation by the developer at hand.

We would like to emphasize that this is **the foundation of high quality software projects**, so tag along.

So, what does SOLID mean?

As the name suggests these principles are meant for maintaining a solid software architecture. SOLID is usually used by software developers who work with object oriented languages. You might have heard of Java or C#, just to name a few. Here's what SOLID means in the software development world:

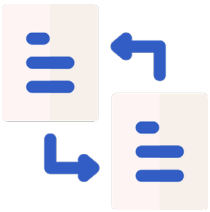
S - Single Responsibility Principle

Every entity, for example a code block, should have a single responsibility, hence can be reused and combined with other entities to create more complex structures.

O - Open / Closed Principle

An entity should be open for extension but closed for modification. If code is never modified, existing functionality most likely won't stop working.

L - Liskov Substitution Principle

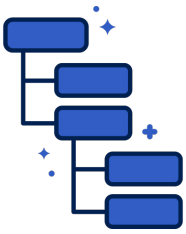


Entities within a program should be replaceable with their subtypes without the necessity to change other parts of the software around that entity.

I - Interface Segregation Principle

It is better to use more specific interfaces instead of general purpose interfaces. Those smaller interfaces, also called “role interfaces”, have a smaller set of functions and are more specialized for a certain use case.

D - Dependency Inversion Principle



Details should depend on abstractions/interfaces. That means that lower level entities should always depend on higher level entities, not the other way around.

We are going to explain these principles in more detail, accompanied by practical examples from AX NLG Cloud projects, where both good and bad practices are compared.

Adapting SOLID for Content Automation Projects

Since AX NLG Cloud is not object oriented like many programming languages are, we can’t exactly apply the SOLID principles to content automation without any prior changes. We made it our mission to adapt SOLID to AX NLG Cloud for improved project quality and maintainability, along with many other advantages.

In this chapter we are going to explain exactly **how to apply all SOLID principles to AX NLG Cloud projects**.

Single Responsibility Principle

This is probably the most important principle out of all five SOLID principles when it comes to planning and developing your project. **Anything within AX NLG Cloud that can be assigned one single responsibility should be assigned exclusively that responsibility**.

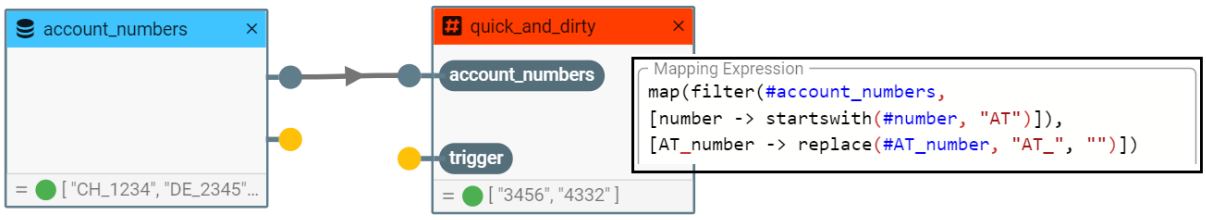
For instance, if your goal is to generate a three sentence paragraph that describes a weather forecast, it might be beneficial to split that into three WRITE statements.

Or, if a code block transforms and filters data (e.g. within a mapping node), you would probably want to split that into two code blocks, one for transforming, one for filtering.

Let's take a closer look at an example

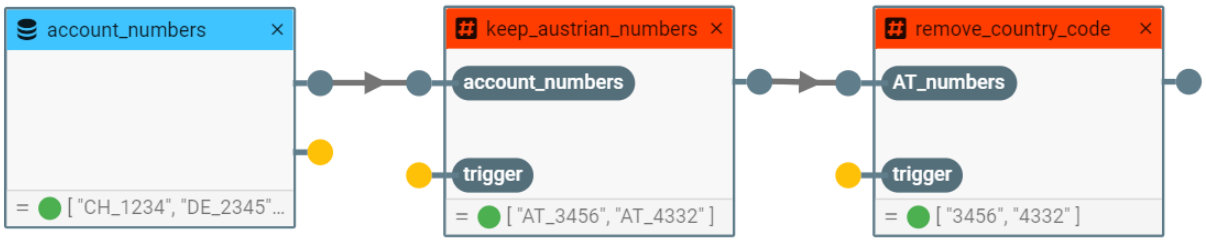
We assume that you have a list of account numbers which start with a 2 letter country code and are separated with an underscore, e.g. "CH_1234" or "AT_4332". But for one of your text sections you only need to have Austrian account numbers without the preceding country code. That means we have two processing steps:

1. Filtering the list of account numbers based on their country code ("AT")
2. Cutting off the country code and the underscore from the account number



As you can see we consecutively called the filter() and map() method to get rid of non-Austrian account numbers first, followed by removing the "AT_" at their beginning. Well, we mentioned before that we might want to split that into two code blocks.

A cleaner solution would be:




```
1: filter(#account _ numbers, [number -> startswith(#number, "AT")])
2: map(#AT _ numbers, [AT _ number -> substring(#AT _ number, 3)])
```

Now we have better readability (you can understand the code just by reading the block titles) and a separation of responsibilities.

This has three major benefits:

1. If any errors were to occur it would be **easier to find the underlying bug** or misbehaviour since the code is split into multiple nodes. Assuming an error would occur in the quick and dirty solution, it would be harder to tell which part of the code is responsible for that error, especially if the block contained significantly more code.
2. It is very often the case that already implemented functions are being changed in the course of a project. If functions are kept separate it is easier to **find all occurrences of a function**.
3. Changes to your code are less prone to errors since the amount of **affected code per change is reduced**.

But isn't there a downside to every improvement as well? There is:

The example shows that we will have to maintain more nodes than before and the project might become messy as a result. A well laid out structure becomes increasingly important. This newly arised issue will be dealt with in the last SOLID chapter.

Open/Closed Principle

The Open/Closed principle states that entities should be open for extension, but closed for modification.

Basically, whenever you have a project where there are several similar tasks, try to **implement the pieces that they have in common and extend those pieces with specifics**.

Let's remember our example from the Single Responsibility Principle where we extracted Austrian account numbers. There might be several similar tasks, such as extracting the account numbers for Swiss accounts as well.

These two tasks have the extraction task in common, the specifics are defined by the different country codes (Austria -> AT, Switzerland -> CH). If we design our project by the Open/Closed Principle, we want to build a reusable and extendable component that is able to extract account numbers for any country. Our approach would be:

```
1 (before): filter(#account _ numbers, [number -> startswith(#number, "AT")])
```

```
1 (after): filter(#account _ numbers, [number -> startswith(#number, #country _ code)])
```

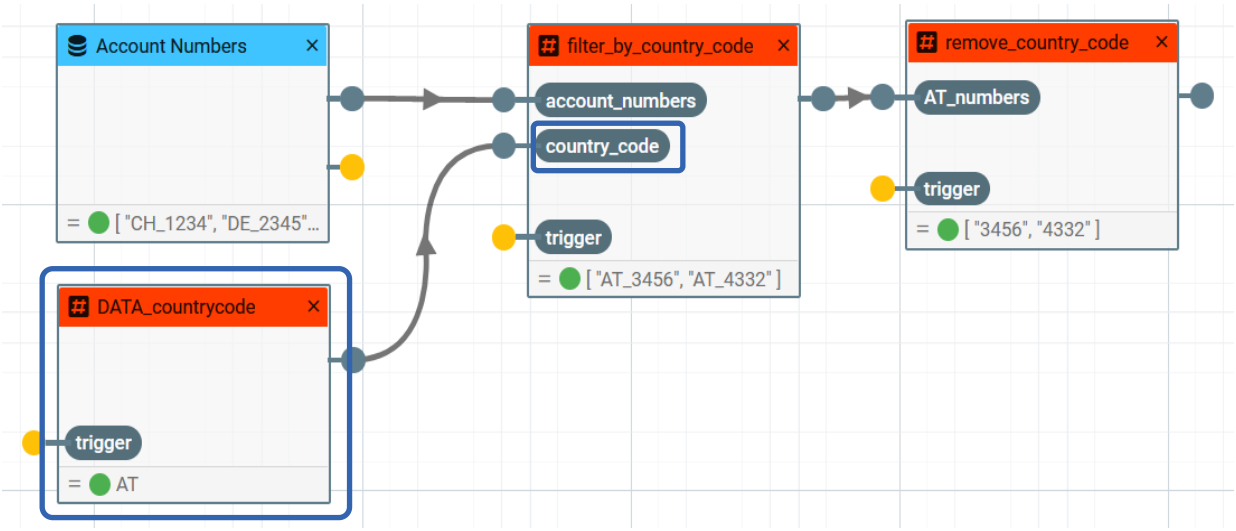
We replaced the explicit country code with a generic country code that gets passed into the mapping node as a parameter.

```
2 (before): map(#AT _ numbers, [AT _ number -> substring(#AT _ number, 3)])
```

```
2 (after): map(#filtered _ numbers, [number -> substring(#number, 3)])
```

Here we simply renamed variables as the old names did not make sense anymore.

As you can see the filter method is now dynamic and will filter for any country that is provided from the outside. These two steps (1 and 2) can now be reused as often as needed, whereas we need to take part of passing the country code as parameter (either from our data object or hard coded string):



The node “DATA_countrycode” simply contains the code snippet “AT” and could also be “DE” or “CH” for the countries Germany and Switzerland respectively (or any other two letter country code).

Remember:

Designing for extendability

...ensures reusability for similar tasks.

...is less error prone than designing a very specific component that has to be modifief again and again for other tasks.

Also, whenever you have multiple developers on a project, this prevents anyone from making changes to each other’s code. This is highly recommended as it is not always possible to determine whether someone else’s piece of code should be changed or not.

Liskov Substitution Principle

“Entities within a program should be replaceable with their subtypes without the necessity to change other parts of the software around that entity.”

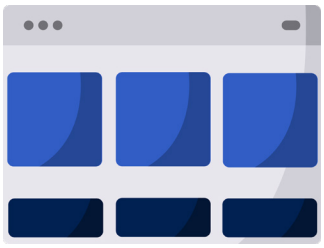
We don’t really have something like subtypes in the AX software for nodes, but AX NLG Cloud is basically built to adhere to the Liskov Substitution Principle since **every node can easily be replaced by one of the same type**.

Interface Segregation Principle & Dependency Inversion Principle

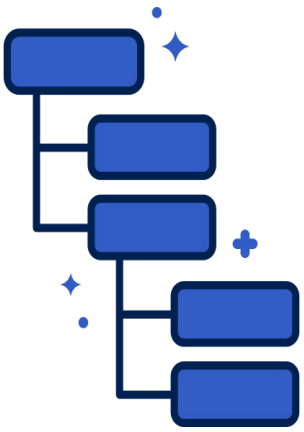
Both principles aim to **ensure loosely coupled software** to prevent overly (and unnecessarily) complex dependencies between parts of your project, which hinder maintainability and readability. With a so-called “closely coupled system” you run into one main problem: If you make a change somewhere in the project it is most likely going to negatively affect other parts as well.

The gist of these principles is:

Interface Segregation
Divide your project into as small
pieces as possible and as large pieces as necessary.



Dependency Inversion
Higher level layers should delegate to lower level layers whereas lower level layers should depend on higher level layers and return results back to the top. This means that logical operations, checks for errors or ensuring the correct data format should happen in lower level layers. They process whatever is being passed downwards to them, whereas higher level layers should be able to trust that they receive the correct or expected results from below.



We will clarify these principles a little further with another example.

Let’s imagine following scenario

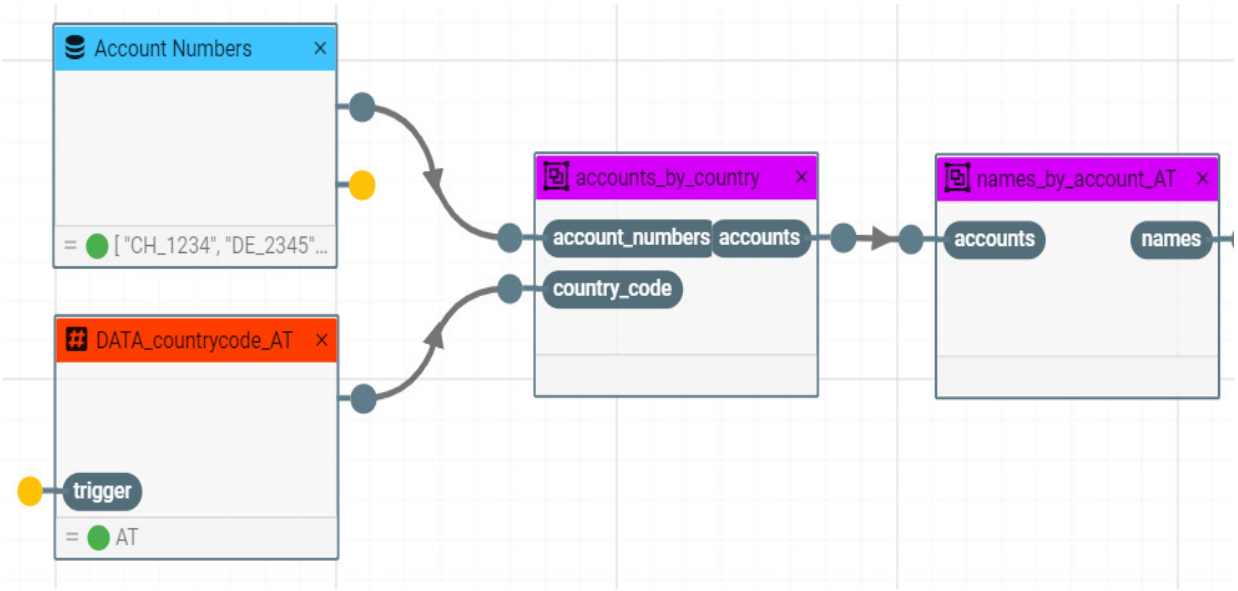
1. Every account number belongs to a client with a pre- and surname
2. Each country has their own lookup table that assigns account numbers to their owner’s name
3. We want to retrieve account numbers from Austrian accounts and get the account holder’s name

Take a moment to reflect on how you would solve this task, given following lookup table:

Lookup Table

„3456": „John Doe"
„4332": „Jane Doe"

Our final output should be [„John Doe", „Jane Doe"] since they hold the accounts AT_3456 and AT_4332. Our approach to that is to package our current implementation (as it already works) and to extend it by the lookup component to transform given account numbers into their respective account owners names:

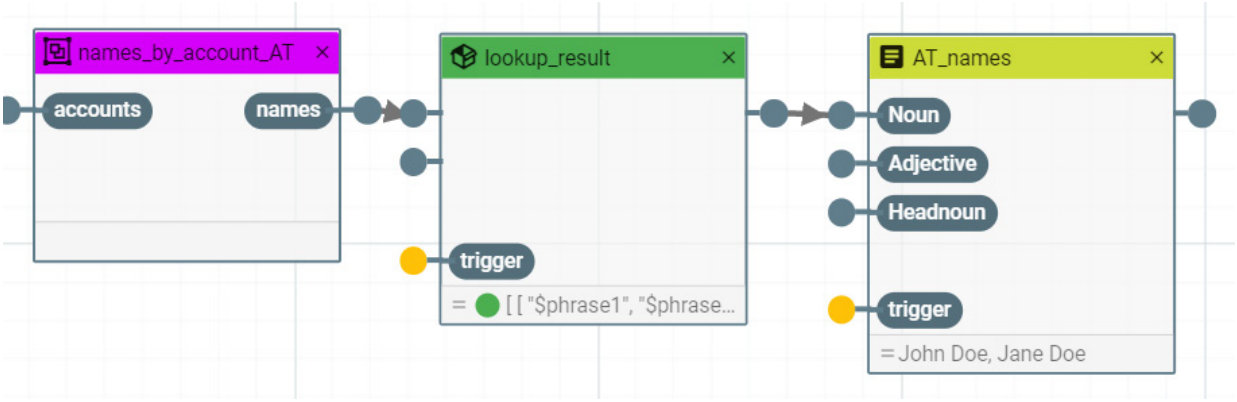


As you can see we have two modules:

- "accounts_by_country": this is our example from before, simply put into a subgraph
- "names_by_account_AT": as the name suggests this module returns the names of Austrian account holders

The second module uses a lookup node that has the contents the given Lookup Table.

The other part that is cut off in the previous image simply demonstrates the output that is returned by our code blocks. And voilà: we receive a list containing John Doe and Jane Doe as result that you can see in the phrase node „AT_names".



In this example we adhered to the other principles as well since we are using modules with single responsibilities which fulfill a greater responsibility in combination.

By segregating our code into "modules" we can reuse them, extend them or swap them out with similar modules depending on the given needs.

By delegating tasks to a lower level we can focus the higher level layer on managing data, structuring different flows and delivering results into variables. Usually we would connect the last phrase node to a variable so we could use the result within the WRITE section.

Every given principle so far complements the others in a way, whereas the last two principles (Interface Segregation & Dependency Inversion) are the foundation of our architecture model that aims to separate different responsibilities across different layers. We will explain the details in the next chapter.

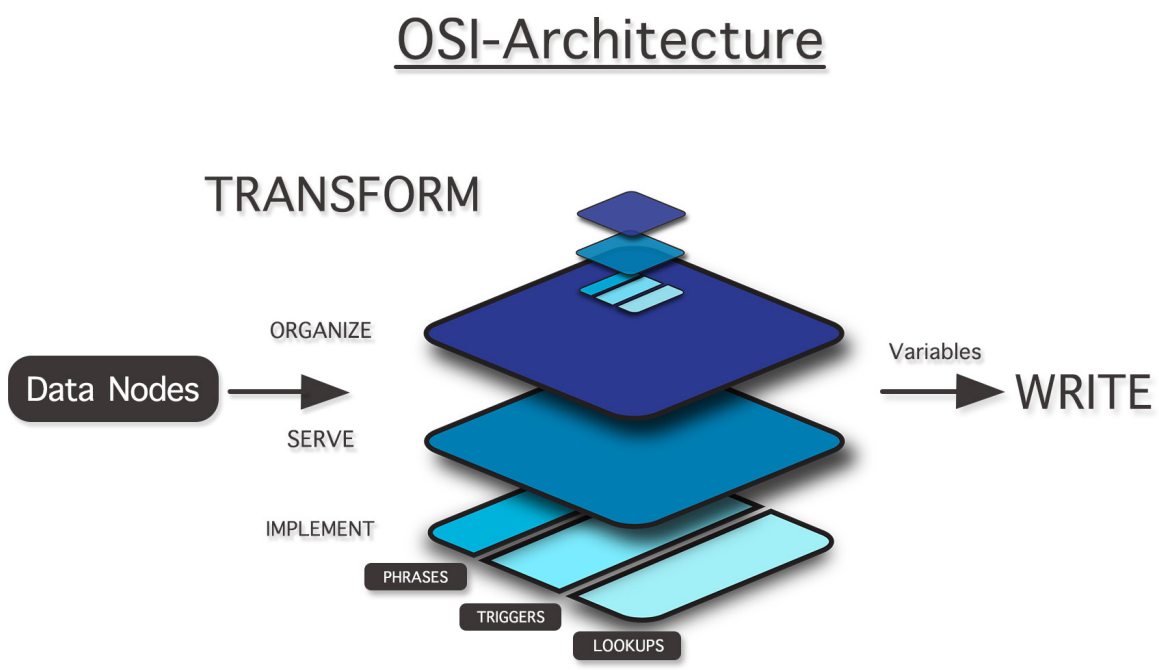
3. OSI Architecture in TRANSFORM

In the previous chapters we explained what the SOLID principles are and how they contribute to better software quality. With an origin in object-oriented programming we can't apply them 1 to 1 to content automation projects which is why we adapted those best practice principles in applicable ways for the AX NLG Cloud.

There are some additional aspects that increase the level of organization and facilitate the handling of the projects:

- Naming conventions for nodes and variables
- A clear project documentation
- Quality Assurance processes, such as peer reviews
- Clean code

SOLID, when done correctly, will ensure quality through a maintainable software architecture, basically the foundation. For this reason we created the OSI-Architecture (**O**rganize-**S**erve-**I**mplement) model for AX NLG Cloud projects



The OSI-Architecture consists of three layers and is highly inspired by the last two SOLID principles: Interface Segregation and Dependency Inversion. A “layer” in this context is just a conceptual layer and not a dedicated feature of the NLG Cloud. The depth of a layer relates to the structural depth of your NLG Cloud project when using subgraphs. This means that the Organize-layer resides at the root level of your project whereas the Serve- and Implement-layer are located within subgraphs below the Organize-layer.

Important to note:

The OSI-layers are purely conceptual, one could say that each layer is an abstraction of a responsibility that can be interpreted and implemented in several different ways.

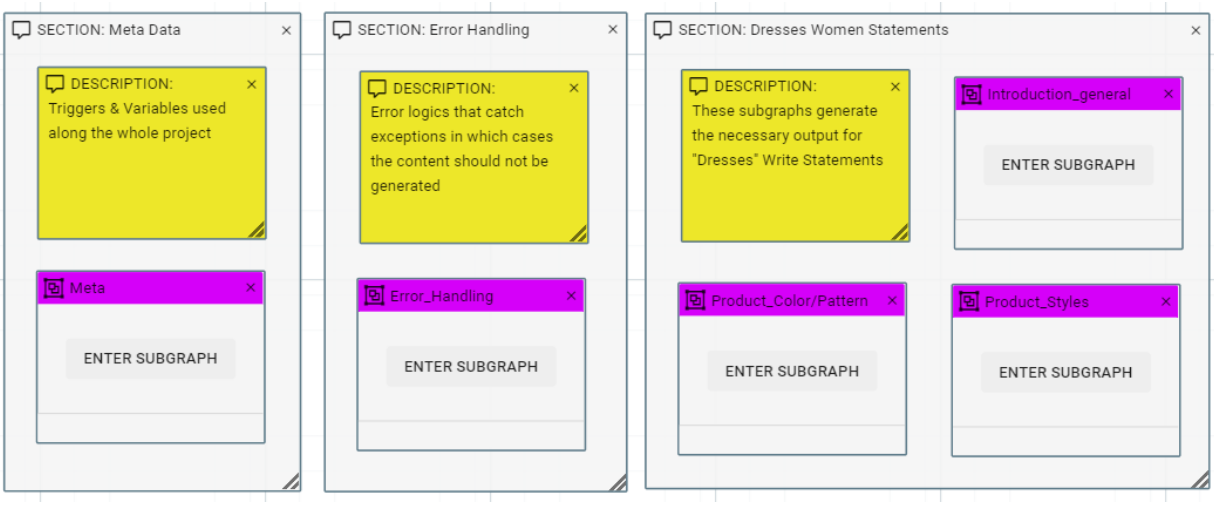
Let’s have a look at the different layers and their responsibilities that are backed up by real project examples with explicit implementations within the areas of Fashion / E-Commerce.

OSI Layer 1 - Organize

Our model’s first layer aims to separate a project into several sections that have no dependencies between each other. This is **the first step to creating a loosely coupled project** and that’s the sole purpose of the ORGANIZE-Layer (O) of our OSI model.

This separation into sections depends on the project size and complexity but usually happens sentence or paragraph wise. Each sentence or paragraph is then implemented in its own subgraph. When looking at the O-Layer, we build it using only sticky note blocks (for documentation purposes) as well as subgraphs. No logic, no computation, no data, no variables, just plain structure the project layout.

You could look at it like building a house. You would usually start with the construction drawings, not the actual construction.



As displayed in the project extract from one of our e-commerce fashion projects, there are three major sections that we usually include in our projects:

1. A section with **metadata and logics that are used across the whole project** and are relevant for most sentences or paragraphs.

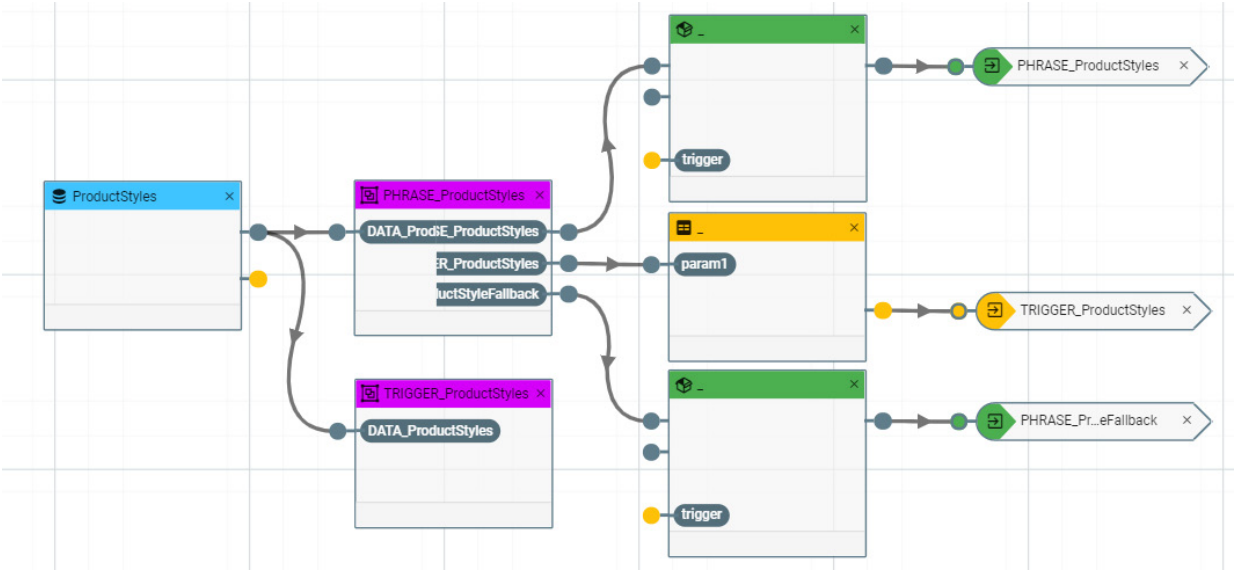
2. The second section that is **responsible for error handling**, contains different logics that can trigger errors. These errors prevent the content from being generated and always deliver a meaningful error message that helps finding the issue later on. These can be especially useful if you have dirty data or to prevent mistakes within the implemented code. Because no developer is perfect.

3. Last but not least there is a statements section. This is where we implement our project statement wise. This means that **each statement in the WRITE section is assigned to one subgraph** of this sticky note section.

OSI Layer 2 - Serve

Since the first layer is all about structuring, our second layer - the SERVE-Layer (S) - is **responsible for managing data, functions and delivering processed data**. The sole purpose of the S-Layer is to serve data that can be inserted into the WRITE section.

We use this layer exclusively to manage data and subgraphs for producing the required output as a variable node. You can think of these subgraphs as condensed mapping nodes because they take inputs, process them and deliver an output that ultimately ends in a variable node. But, if planned correctly, these subgraphs are reusable components that you can use across different projects or within the same project.



Thinking about our house construction metaphor from the previous section, the S-Layer would be the foundation and the supporting elements of a house.

OSI Layer 3 - Implement

Last but not least, the IMPLEMENT-Layer (I) is responsible for **detailing the exact logic behind any trigger or container** that is being used and served to the WRITE section.

This layer is implemented within subgraphs that are used in the S-Layer. They do the heavy computing, they transform data, perform logical operations and return the desired results.

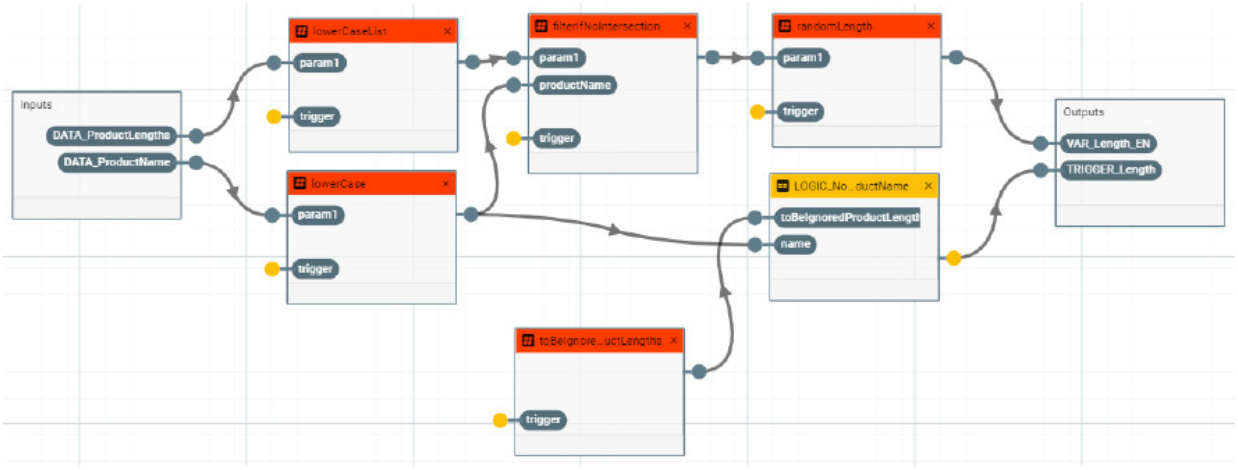
The most common modules that we use are:

- Lookup modules - They help with very specific data transformations, translations, adding adjectives or filtering data that is not allowed to appear in the text output.

- Trigger modules - These have the sole purpose to create lots of triggers that can be used within the WRITE or TRANSFORM section. For instance one could use a trigger module to create triggers for different target groups based on the given fashion collection.
- Phrase modules - The purpose of these modules is to declutter the WRITE section because sometimes it is required to create hundreds or even thousands of variations within a single statement. You wouldn't want to do that with a hundred branches in the WRITE section, trust us.

By Lookup, Trigger or Phrase modules we do not refer to the respective nodes within the NLG Cloud. These modules are meant to be subgraphs that are responsible for implementing logic around lookup tables, conditions for triggers or more complex groups of phrases.

Here you can see an example that shows the contents of a subgraph on the I-Layer that extracts the length of a dress if it is not already contained within the product name:



The usage of this module happens on the lowest layer, the I-Layer or Implementation-Layer because we explicitly implement logics and manipulate data. We exclusively do that below the S-Layer.

Of course, in AX NLG Cloud it is possible to create variables within any layer, thus we could create and maintain data nodes as well as variables within the I-Layer. But by doing that we would give up structural consistency and mix up responsibilities which is why we firmly believe that creating variables should only be done in the S-Layer.

4. Advantages and Disandvantages of OSI

So, what are the disadvantages then? Generally speaking, at some point you might be “over-structuring” your project, especially if you are only planning to implement a small project with one or two components. But sometimes, even for small projects, it might make sense to apply an architecture if the project might scale in the future.

Therefore, our recommendation is to establish architectural guidelines within your project teams that are applicable to all sorts of project sizes.

We summarized key advantages of disadvantages of using OSI:

Pro: 

From a technical perspective **project planning becomes easier and more structured**

Tasks can be pipelined and implemented in parallel (huge productivity boost for teams with multiple developers)

Increased project quality that is more sustainable from a business perspective

Cons: 

More time goes into planning projects which means first productive results are achieved later

There is a node limit for AX NLG Cloud projects and OSI uses more nodes than other projects (though we have never seen a project that actually requires anywhere near as many nodes as the limit allows)

We would also like to point out a few criteria that help determining whether investing time into a software architecture is the right thing to do.

- **In Projects with multiple developers**, tasks can be finished and peer reviewed more efficiently by parallelizing effort with a team.

- **Multilingual projects** can be implemented more quickly as more languages can be added in a modular way

- **Many data sources and data categories** (for instance product categories) usually share common attributes. Reusability across those categories and attributes can be a huge time saver.

- **Agile project settings** are great as OSI allows you to develop with interchangeable and small components, enabling swift changes in less time. Changes also don't break or halt the project.

- **Switching or upgrading databases** is not a problem. New data can be integrated as extension and data changes will most likely not affect major parts of a project as it is not closely coupled.

In addition to the benefits you get from using an architecture model, there are of course costs involved, for example for training your project managers and developers.

Nevertheless, we believe that the **positive aspects far outweigh the initial investment**. These include communication, planning, working in teams, software quality and project flexibility.

Appendix

About Xanevo

We are a managed/full service provider and partner of AX Semantics.

Our mission at Xanevo is to automate unique texts with a strong focus on e-commerce, financial reporting and publishing industries.

We pride ourselves on our quality standards in software development using NLG Cloud, ensuring sustainable and robust projects for our clients. Here, we are guided by Scrum and make use of proven models and principles from the software development world.

We do not want to keep this knowledge and the associated methodology exclusive, which is why we share our expertise in seminars, workshops and other educational content.

About AX Semantics

AX Semantics is a AI-powered, Natural Language Generation (NLG) software company with its roots grounded in content and storytelling. Our sophisticated, yet easy to use SaaSbased software makes automated content generation accessible to customers of all sizes, is used widely within the e-commerce, business, finance and media publishing sectors. Available in 110 languages, AX Semantics works with more than 500 customers, including globally recognized brands like Deloitte, BASF, Ebner & Stolz, Porsche, and Nivea. AX Semantics was named one of the world's world's top five providers of Natural Language Generation platforms by Gartner, and a top emerging company in the NLG market by Forrester.

AX Semantics is changing the way content is created, published and viewed. Our software allows thousands of users to successfully automate text within two days, and gives people the space to develop and nurture their creative originality.

Headquartered in Stuttgart, Germany with an additional office in Sunnyvale, California, AX Semantics is a privatelyheld company backed by Airbridge Equity Partners. Follow us on social at Twitter, LinkedIn, Instagram and Facebook, or learn more at https://www.ax-semantics.com.

Sources to read up on SOLID principles

Robert C. Martin (2017): Clean Architecture: A Craftsman's Guide to Software Structure and Design, ISBN-13: 978-0134494166