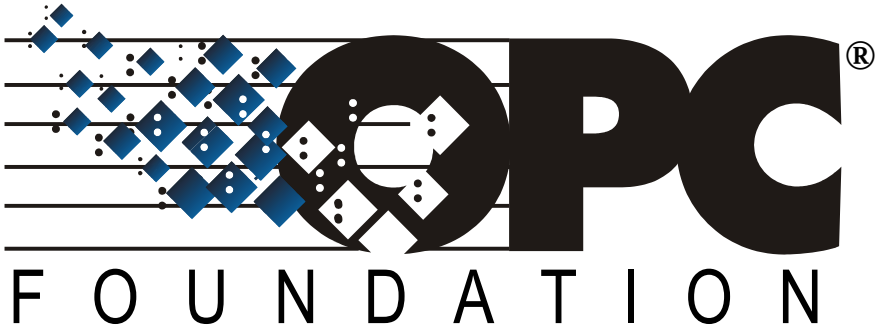

OPC 11030

OPC Unified Architecture

UA Modelling Best Practices

Release 1.00.00

2020-07-21

OPC UA Whitepaper

CONTENTS

Page

OPC FOUNDATION

This document is a whitepaper describing guidelines for creating OPC UA based information models. It is not a formal specification.

1. Scope

This whitepaper is intended to provide guidelines and best practice for information modellers creating OPC UA based information models. It is not a formal specification, i.e. it is not required to follow the recommendations in this whitepaper, but highly recommended.

The whitepaper is organized in several sections addressing different aspects of information modelling, from naming conventions to concrete modelling patterns for specific purposes.

The document is a living document, i.e. it will get extended with additional aspects over time, and will be published at a higher frequency than specifications would be.

2. Naming Conventions

2.1. Naming Conventions for Nodes

OPC UA defines two attributes containing naming information about an OPC UA Node, the BrowseName and the DisplayName.

Recommendations for DisplayName will be given in a later version of this document.

The BrowseName is of DataType QualifiedName, containing a NamespaceIndex and a String. Unless the BrowseName is defined in some other Namespace or there is some specific handling for the BrowseName, the Namespace for the BrowseName should be the one the Node is defined in (i.e. the same Namespace as the NodeId). Nodes defined in a Companion Specification should use the Namespace of the Companion Specification for their NodeIds and BrowseNames.

For the string-part the following naming conventions apply:

2.1.1. General Rules for BrowseNames

All BrowseNames should be upper camel case (also known as PascalCase), that is, all words written without spaces, and the first character of each word is upper case, the other characters are lower case. Examples: ReferenceType, BaseObjectType, Int32

If an acronym or abbreviation is used, upper camel case should also be used. Examples: PortMacAddress (where MAC is an acronym for Media Access Control), NodeId (where ID is an abbreviation for identification), UInt32 (where U is an abbreviation for unsigned).

In general, it is recommended to only use letters, digits or the underscore ('_') as characters for the BrowseName for TypeDefinitions (ObjectTypes, VariableTypes, DataTypes, ReferenceTypes and InstanceDeclarations), unless it is explicitly defined like "<" and ">" for optional placeholders.

Remark: If special chars like "&", "<", etc. are used, the NodeSet-File should define the optional SymbolicName for that Node. This can then be used for code generation.

There is no recommendation on the use of prefixes. Companion specifications may use a prefix because it suits their model. For example, if the Vision companion specification were to define types based on generic concepts (say a state machine), then using the prefix "Vision" may make sense (as in "VisionStateMachineType").

2.1.2. Rules for specific NodeClasses

The BrowseNames for **ObjectTypes** should be suffixed with "Type": Examples: ServerType, NamespaceMetadataType, BaseEventType.

The BrowseNames for **VariableTypes** should be suffixed with "Type". If there is an ambiguity with other Nodes, then the suffix "VariableType" should be used. Examples: PropertyType, BaseDataVariableType.

The BrowseNames for **DataTypes** depend on the type of DataType.

- For Structured DataTypes (subtypes of the Structure DataType) the suffix should be "DataType". Note that the base OPC UA specification does not always follow this pattern. Examples: RedundantServerDataType, TimeZoneDataType.
- Enumeration DataTypes (subtypes of the Enumeration DataType) should not have a suffix. If suffix is used, it should be "Enum". Examples: NodeClass, ServerState.

- Built-in DataTypes can only be defined by the base OPC UA specification. They should not have a suffix. Examples: String, Int32, Byte.
- Simple DataTypes (subtypes of the built-in DataTypes) should not have a suffix. Examples: Image, UtcTime, LocalId.

The BrowseNames for **ReferenceTypes** should not have a suffix. It should describe the relationship from the source to the target. Relationships are typically described as verbs. When using a noun, it should be prefixed with a verb like “Has”. Examples: HasComponent, HasEffect, HasChild, Organizes, Aggregates.

The InverseName of a ReferenceType is only provided for asymmetric References. It should be the inverse Name of the BrowseName, e.g. ComponentOf as the inverse for HasComponent.

For **Objects**, **Variables**, **Methods** and **Views** there are no specific rules, i.e. they should neither be prefixed nor suffixed.

2.1.3. Additional Considerations

The BrowseNames for TypeDefinitionNodes (ObjectTypes, VariableTypes, and DataTypes) shall be unique (as defined in the base OPC UA specification). That means, that you need to make sure that you do not create duplicates inside the Namespace you are creating.

As OPC UA Clients typically only display the string-part of the BrowseName (and thus not the NamespaceIndex), it is desirable not to use the same name across Namespaces. As there are many companion specifications, this cannot be guaranteed, but should be avoided as much as possible, for example by not using names from the base OPC UA specification.

For InstanceDeclarations the BrowsePath shall be unique (defined in the base OPC UA specification). OPC UA Servers are allowed to treat BrowseNames in a case-insensitive manner. OPC UA Clients shall consider case sensitivity of the BrowseNames. Since OPC UA Servers might not handle case sensitivity, information models should not rely on case sensitivity for the uniqueness of BrowsePaths, but instead define BrowsePaths that are also unique when not considering case sensitivity. For example, one TypeDefinitionNode should not have the Properties “Id” and “ID”. Instead, prefix the name with the intended use, such as “NetworkId” and “DeviceId”.

2.2. Naming Conventions for structure Fields

When creating Structured DataTypes, each field of a structure has to have a unique name (as defined in the base OPC UA specification). The naming convention is to use upper camel case for those names. Examples: Offset, DaylightSavingInOffset

Note: In many companion specifications, and in the base OPC UA specifications, lower camel case (e.g. first word starts with a lower character) is used in the specification. However, the NodeSet-File uses upper camel case, and thus the OPC UA Server provides everything as upper camel case.

3. Rules for backward compatibility of Information Models

3.1. Overview

An Information Model defines a contract between OPC UA Applications. OPC UA Clients can rely on the mandatory parts defined in such an Information Model and OPC UA Servers are not required to provide more than the specified mandatory information.

Elements defined in an Information Model are qualified by their Namespace, which is a unique URI. When a new version of such an Information Model is defined, some rules need to be considered in order to re-use the existing Namespace. If those rules cannot be followed, the existing Namespace URI should not be used. However, a new Namespace URI implies that all information is new, that is, even using the same name and NodeId (without NamespaceIndex) for a type, OPC UA Applications will consider it to be a completely different Node with no relation to the one qualified by the old Namespace.

The next clauses describe the rules that must be followed in order to re-use the existing Namespace when versioning an Information Model.

3.2. Rules

3.2.1. Adding mandatory components to ObjectTypes and VariableTypes

It is not allowed to add new mandatory InstanceDeclarations using the ModellingRules Mandatory and MandatoryPlaceholder. To add a mandatory InstanceDeclaration, either create a new Subtype having the mandatory InstanceDeclaration, or a new type independent of the previous one. As alternative, make the new InstanceDeclaration optional but define a Profile requiring the new InstanceDeclaration.

This rule avoids the incompatibility when an OPC UA Client aware of the new version connects to an OPC UA Server only supporting the old version, and the OPC UA Client expects a new mandatory feature not provided by the OPC UA Server.

3.2.2. Adding optional components to ObjectTypes and VariableTypes

It is allowed to add new optional InstanceDeclarations using the ModellingRules Optional and OptionalPlaceholder.

Old OPC UA Servers will just not provide those, and new OPC UA Clients will not expect them to be available in all OPC UA Servers, since they are optional. Since OPC UA Servers are allowed to add anything to the instances, an old OPC UA Client can also easily connect to a new OPC UA Server, potentially just ignoring the new information.

3.2.3. Adding Interfaces to ObjectTypes

It is allowed to add Interfaces to ObjectTypes as long as no new mandatory InstanceDeclarations are added to the ObjectType. That means, a new Interface can be defined based on an existing ObjectType by using some InstanceDeclarations of that ObjectType. Such an Interface may have mandatory InstanceDeclarations already defined in the ObjectType, and in that case the ObjectType can implement the Interface.

3.2.4. Changing the values of an Enumeration DataType

It is not allowed to add or remove values of an Enumeration DataType. It is allowed to limit the enumeration values, creating a subtype with a subset of the existing values defined by the subertype.

Old OPC UA Clients would not expect new values, and old OPC UA Servers might provide removed values to new OPC UA Clients that would not be able to interpret these based on their built-in knowledge.

If it is foreseeable that future specification versions would need to add or remove enumeration values, or that derived companion specifications would want to extend them, then one of the MultiState VariableTypes should be used instead of an Enumeration DataType.

3.2.5. Changing the fields of a Structure DataType

It is not allowed to add or remove optional or mandatory fields of a Structured DataType or change the name or DataType of a field.

This would change the encoding and OPC UA applications might not be able to encode / decode the information anymore with built-in knowledge.

3.2.6. Changing the Type Hierarchy of ObjectTypes or VariableTypes

It is allowed to add subtypes into the existing Type-Hierarchy.

It is not recommended to insert types inside the existing hierarchy, as shown in Figure 1. In the example, a NewType is added as supertype of Subtype_B and Subtype_C.

If absolutely necessary, the following rules apply: The new type shall not add mandatory InstanceDeclarations unless they have been defined before on each subtype. The new type shall not define additional constraints (defined in the text of the specification), unless those constraints were already valid for all its subtypes before. The newly added type might be abstract or concrete. In a nutshell, the new type shall not add anything to the subtypes that would not be allowed to be added to the subtypes directly.

It is not allowed to add new types inside the EventType Hierarchy, i.e. in the part of the ObjectType Hierarchy inheriting from the BaseEventType, as EventTypes are used for filtering, and new OPC UA Clients might use the new EventType as a filter in an old Server, which would not work.

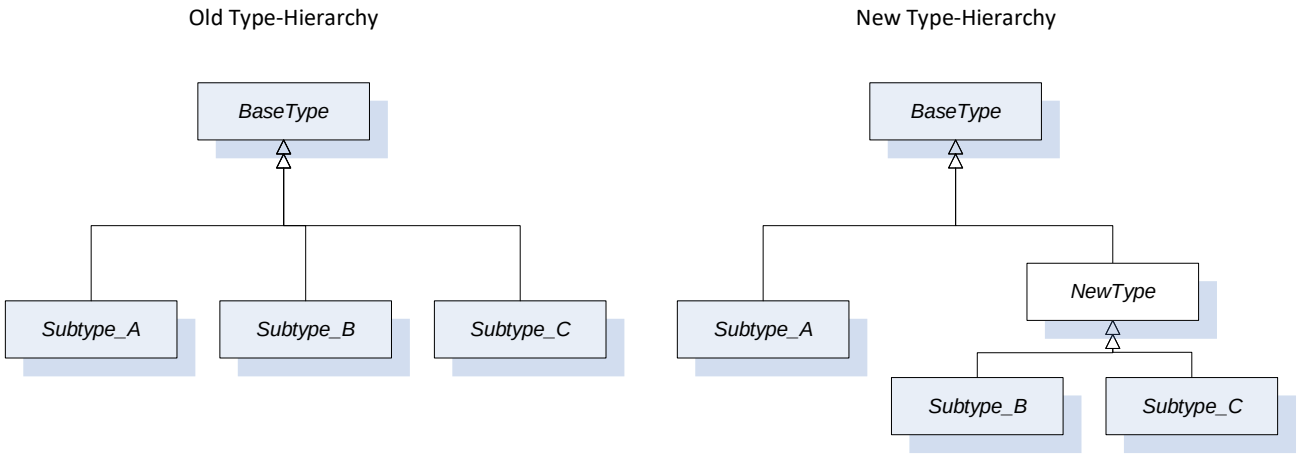

Figure 1 Example of adding types inside an existing type hierarchy

The reason to allow changing the type hierarchy is that OPC UA Clients aware of the old type hierarchy would not browse the type hierarchy, and therefore an old OPC UA Client would work with a new OPC UA Server. A new OPC UA Client would also not browse the type hierarchy of the old OPC UA Server, the old OPC UA Server would just not have any instances of the newly added types, which the new OPC UA Client has to expect anyway.

3.2.7. Changing the Type Hierarchy of DataTypes

It is allowed to create new subtypes of existing DataTypes.

It is not recommended to insert types inside the type hierarchy of DataTypes (similar to ObjectTypes and VariableTypes, see Figure 1). If absolutely necessary, the following rules apply:

- For all DataTypes no new constraints (written as text in the specification) should be added, but existing constraints common to all the subtypes can be added to the new supertype. It shall be allowed, that all subtypes are created based on the new DataType.
- For abstract DataTypes there are no additional rules.
- For simple DataTypes (using the encodings of the built-in DataTypes) there are no additional rules.
- For Enumeration DataTypes the new supertype may have more Enumeration Values than the subtypes. In that case, it shall have all Enumeration values defined in at least one of its subtypes. If the numeric values would conflict, it is not possible to create such a supertype.
- For Structured DataTypes the structure shall have no fields that are not already defined in all subtypes having the same DataType or a supertype of the DataType.

3.2.8. Changing the Signature of a Method

It is not allowed to change the signature of a Method. Even adding new optional Arguments to an existing Method would change the signature, so this is not allowed.

It is allowed to add metadata to the existing arguments of a Method, e.g. the EngineeringUnits Property (done with HasArgumentDescription References, see section 5 for an example). This does not include changing arguments from mandatory to optional or vice versa.

It is also allowed to change the description text of a Method parameter, as long as this does not change the semantics of the argument.

If the signature of a Method would change, new OPC UA Clients might call old OPC UA Servers with the wrong signature (e.g. if an argument has become optional), or old OPC UA Clients the new OPC UA Server with wrong arguments (e.g. if an argument has become mandatory). OPC UA Clients with prior knowledge of the information model do not necessarily read the Method arguments from the OPC UA Server, and even if they would, their implementation to call the Method would not work anymore.

If the intention is to change the signature of a Method, then a new Method with a new signature and a new BrowseName shall be defined instead.

3.2.9. Changing the DataType of a Variable

It is not allowed to change the DataType, ValueRank or ArrayDimensions of a Variable, neither for an InstanceDeclaration, nor for a standardized Variable like ServerStatus.

Even refining an existing DataType to a subtype is a breaking change. New OPC UA Clients connecting to an old OPC UA Server would expect the subtype, but might get the supertype instead.

3.2.10. Changing the TypeDefinition of an InstanceDeclaration

Changing the TypeDefinition of an InstanceDeclaration is not allowed, unless it is being changed to a subtype of the already specified one. Using a subtype is only allowed, if the subtype does not add any mandatory InstanceDeclarations or constraints (defined in the text of the specification), and does not refine the DataType (in case of a Variable).

3.2.11. Changing the Semantics of ReferenceTypes

ReferenceTypes typically restrict their usage by limiting which SourceNodes and TargetNodes are allowed. Currently such restrictions are only specified in text in specifications.

Changing the rules, either allowing more or less restrictions, should be avoided. However, OPC UA Clients should consider the results when following References (e.g. the ReferenceDescription containing NodeClass and TypeDefinition) and they should be able to handle unexpected results. Therefore, changing the rules is not strictly forbidden, as long as it does not contradict other parts of the information model, e.g. because a TypeDefinition is using the ReferenceType in a way that is no longer allowed in the new version.

3.2.12. Changing the Type Hierarchy of ReferenceTypes

It is allowed to add subtypes to the existing ReferenceType hierarchy. It is not allowed to move types in that Hierarchy, e.g. making a hierarchical ReferenceType a non-hierarchical ReferenceType, as the filtering would behave differently. It is also not allowed to add new ReferenceTypes inside the Hierarchy. This would work for old OPC UA Clients, since they would just not use those unknown ReferenceTypes for filtering, but not for new OPC UA Clients connecting to old OPC UA Servers, since they would expect the ReferenceType to be available and use it for filtering information available in the OPC UA Server.

3.3. Strategies for Breaking Changes

This section will discuss strategies to follow when breaking changes have to be made in a later version of the document.

4. How to define StatusCodes in Companion Specifications

In OPC UA StatusCodes are used on different levels to indicate:

- if service calls have been executed successfully (e.g. a Read call was successful)
- if operations inside the service calls have been executed successfully (e.g. for each Attribute to be read in a Read call), or the status of individual data like in MonitoredItems.

The base OPC UA specification defines those StatusCodes. It is not allowed for companion specifications or vendors to define additional StatusCodes. OPC UA provides the mechanism of DiagnosticInfo (see OPC UA Part 4) to provide more specific information.

In very rare cases, companion specifications may have the need for additional StatusCodes. For example, the FDI specification had a need to indicate if a value was currently edited by the specific OPC UA Client. In this case, the companion specification authors shall contact the base OPC UA working group and the needed StatusCodes will be added to the base specification.

5. How to return application-specific statuses in Methods

5.1. Overview

OPC UA uses StatusCodes to identify the success of operations, including Method calls. However, StatusCodes cannot be extended, and often it is desirable to receive a more detailed application-specific, program-readable reason, why a Method call was not successful.

The following pattern should be used to return application-specific statuses of a Method. The last argument of the output arguments of a Method should contain an application-specific error code. The DataType should be an Integer, preferably an Int32. The information model might provide a description of the values in the Method metadata.

In the following, an example of such a Method definition is given, using the template for companion specifications.

5.2. Example

If there is an application-specific error in the Method execution, the Method should return an Uncertain StatusCode and the application calling the Method shall only consider the Status output argument of the Method providing the application-specific error code.

Signature

```
<SomeMethod> (
   [in]  String       <SomeInput>,
   [out] UInt32       <SomeOutput,
   [out] Int32        Status);
```

Table 1 – <SomeMethod> Method Arguments

Argument	Description
<SomeInput>	<description>
<SomeOutput>	<description>
Status	This is an example where the Method needs to return special status information.
	0 – OK – Everything is OK -1 – E_FirstError – Error FirstError occurred -2 – E_SecondError – Error SecondError occurred

Method Result Codes (defined in Call Service)

Result Code	Description
Uncertain	The value is uncertain. A concrete reason is defined in the Status Output-Argument.

Table 2 – <SomeMethod> Method AddressSpace definition

Attribute	Value				
BrowseName	<SomeMethod>				
References	**NodeClass**	**BrowseName**	**DataType**	**TypeDefinition**	**Others**
HasProperty	Variable	InputArguments	Argument[]	PropertyType	M
HasProperty	Variable	OutputArguments	Argument[]	PropertyType	M
HasArgumentDescription	Variable	Status	Int32	MultiStateValueDiscreteType	M

The InstanceDeclarations of the <some>*Type* have additional *Attributes* defined in Table 3.

Table 3 – <some>Type Additional Attributes

Source Path	Value
<SomeMethod> Status EnumValues	[{"value":0, "displayName":{"locale":"en", "text":"OK"}, "description":{"locale":"en", "text":"Everything is OK"}}, {"value":-1, "displayName":{"locale":"en", "text":" E_FirstError "}, "description":{"locale":"en", "text":"Error FirstError occurred "}}, {"value":-2, "displayName":{"locale":"en", "text":" E_SecondError "}, "description":{"locale":"en", "text":"Error SecondError occurred "}}]

Note that the last table is defined in the context of the ObjectType <some>Type containing the Method <SomeMethod>.

_____