



OPC UA Java SDK

Code Generator Manual
Version 3.0.0

Table of Contents

1. Introduction	1
2. About Information Models	1
2.1. Obtaining Information Models	1
2.2. Information Model Versions	1
3. Using Codegen	1
3.1. Running from Command Line	1
3.2. Maven Integration	2
4. Configuration	3
4.1. Mapping NamespaceUris to Java Packages	3
4.1.1. Namespace Prefix	3
4.2. Generation Targets	3
4.2.1. Target Configuration	4
4.3. Method Implementations	5
4.4. Object Initializations	5
4.5. Configuration Parameters	6
4.5.1. Model Directories	6
4.5.2. Namespace Mapping Parameters	6
4.5.3. Generation Target Parameters	7
5. Using Generated Classes in Applications	8
5.1. Client-side Applications	8
5.2. Server-side Applications	8

1. Introduction

The Prosys OPC UA Java SDK Code Generator (later just 'Codegen') can be used to reduce manual workload when using custom information models in the Prosys OPC UA Java SDK. It creates Java classes based on given information models provided in the OPC UA NodeSet2 XML format.

2. About Information Models

Codegen will work with most information models. If you encounter a model that does not generate, let us know.

2.1. Obtaining Information Models

Codegen requires an OPC UA information model in the NodeSet2 format as its input for the generation procedure. The NodeSet2 format stores an information model in XML file(s) conforming to the Information Model XML Schema of the OPC UA specification (v1.03, Part 6, Annex B).

A popular software for creating and managing information models is the [OPC UA Modeler](#) that also enables exporting the information model in the NodeSet2 format. The exported models can then be used as the input for Codegen.

2.2. Information Model Versions

Codegen is currently supporting OPC UA Specification version 1.03 or earlier. OPC UA models based on version 1.04 or later might not work properly with it.

Possible name clashes related to method names

- Codegen will generate a setter and getter for each UA Variable of an UA Object, e.g. `getVar()` and `setVar(...)`. If you define an UA Method with the same name, e.g. `getVar` or `setVar`, and the same argument type inside the same UA Object, then you will get a name clash.
- The generated Java methods do not include the UA namespace in any way. This means that if an UA type defines multiple components with the same BrowseNames that only differ by their NamespaceIndices, then it will result in a name clash.

3. Using Codegen

Codegen can be currently run in two ways:

1. From command line. See chapter [Running from Command Line](#).
2. With Maven. See chapter [Maven Integration](#).

In both cases, the generated code will compile and work in Java 6 or later.

3.1. Running from Command Line

For running Codegen from command line you will need the `java` command to be available from PATH. Codegen requires Java 7 or 8 (but the generated code can be compiled in Java 6 or later).

The parameters of the code generation can be modified through a configuration XML file. Please see the example configuration files in the 'configexamples' folder. Also see the [Configuration](#) chapter for more general information on configuring the code generation procedure.

Codegen is designed to be launched via the supplied startup scripts found in the 'codegen/bin' directory: 'codegen.bat' for Windows and 'codegen.sh' for other operating systems.

The following arguments can be passed to the scripts:

Table 1. Codegen command line arguments

Argument	Info
-c <path>	Path to the configuration XML file
-v	Output debug log
-h	Display help

On Windows, run Codegen with:

```
codegen.bat -c path_to_configuration_file
```

On other operating systems, run Codegen with:

```
./codegen.sh -c path_to_configuration_file
```

3.2. Maven Integration

If you use Apache Maven as your build tool, you can use the experimental maven plugin for Codegen.



This tutorial does not cover Maven usage in general. Please, visit [Apache Maven website](#) for more information.

The Maven plugin for Codegen is provided in the 'maven-install-helper' Maven project. It can be installed to your `.m2` local repository by running `mvn install` in the 'maven-install-helper' folder (see the pom.xml in that folder for more information). Then you can look at the 'maven-integration-sample' project and run `mvn generate-sources` for it.



The generated classes are not added to the build path automatically. If needed, you can use the [Build Helper Maven Plugin](#) to achieve that. Additionally, note that if you use the generation option that copies the model as output, you must use a different output folder for it and use the add-resource goal of the build-helper plugin to add them.

The parameters of code generation can be modified by configuring the Codegen plugin in the associated 'pom.xml' file. See the [Configuration](#) chapter for more general information on configuring the code generation procedure.

4. Configuration

This section explains some of Codegen's configuration and usage concepts.

4.1. Mapping NamespaceUris to Java Packages

OPC UA has the concept of namespaces where each node is in some namespace. Namespaces organize OPC UA nodes according to naming authorities and allow different namespaces to contain same type names. Different namespaces are distinguished by their NamespaceUris. In an OPC UA server, each namespace is also identified by individual indices in the NamespaceArray of the server. They are used instead of the NamespaceUris to allow for a more compact data transfer. However, because the indices are defined during server runtime, they cannot be utilized in the code generation process and instead the NamespaceUris are used to identify namespaces.

Java programming language has the concept of packages which are used to organize classes. Therefore, in Codegen, each OPC UA namespace is mapped to a unique Java package.

4.1.1. Namespace Prefix

Additionally, a prefix parameter can be defined for each generated namespace. The prefix is used to prepend the generated class names of `Ids`, `ClientInformationModel` and `ServerInformationModel`. This helps with identifying the different implementations of these classes when you are registering them in your client or server application. If you don't use the prefix, you can still classify the classes with their fully qualified class names (including the Java package name).

4.2. Generation Targets

Codegen generates Java classes from types (ObjectTypes, VariableTypes, ReferenceTypes and DataTypes) defined in OPC UA information models.



If your models include instances (Objects and Variables), Codegen ignores these, since it can only generate Java classes for UA types. However, when you load the model to your server at startup, these instances will be generated by the SDK. See the Server Tutorial for more information.

Codegen can create the following Java interfaces and classes for each OPC UA ObjectType and VariableType that is defined in the models (using the UA AnalogItemType as an example):

- Common *interface* that has the same name as the UA type: `AnalogItemType`
- Base *class* for the client side: `AnalogItemTypeImplBase`
- Actual *class* for the client side: `AnalogItemTypeImpl`
- Base *class* for the server side: `AnalogItemTypeNodeBase`
- Actual *class* for the server side: `AnalogItemTypeNode`



The base classes should not be modified after generation. The actual classes can be modified, but note that Codegen will override the changes when you regenerate. So you should copy the actual classes to version control to protect them from unwanted changes..

For each UA Method in an ObjectType, the following Java files are created (using the **Resend** Method of the **ServerType** as an example):

- Parameter object *class* for each Method that has more than one output argument. It is used as the return value for the method. It is generated as a nested class of the Common interface.
- Helper *interface* for providing an external method implementation: **ServerTypeResendMethod**.

Additionally, the following classes are generated per model:

- Structure *class* for each UA Structure DataType
- An *enum* for each UA Enumeration DataType
- **Serializers** *class*, which will contain one Serializer per Structure
- **DataTypeDictionaryHelper** *class*, which is used internally by the SDK to handle custom structures
- **ClientInformationModel** *class* that is used to register the generated classes on the client side
- **ServerInformationModel** *class* that is used to register the generated classes on the server side
- **Ids** *class* which contains the identifiers (as ExpandedNodeId) for each generated type and their subcomponents



If a namespace is mapped with a *prefix*, it will be used to prepend the **Ids** and **InformationModel** classes. For example, for a prefix **Test** the output would be **TestIds**, **TestClientInformationModel** and **TestServerInformationModel**.



For Optional Structures and Unions, both of which are specialized forms of Structures, introduced in OPC UA version 1.03, the EncodingMask and SwitchValue are handled automatically by the generated code (in **Serializers**). Please see OPC UA Specification (v.1.03) Part 6 sections 5.2.6 and 5.2.7 for more information.

4.2.1. Target Configuration

Codegen enables you to configure which targets you wish to generate by editing the configuration files. For this purpose, the following target names are defined:

Table 2. Possible generation targets

Target	Description
common	Generate Java interfaces for each type in the model plus Ids, Serializers, Structures, Enumerations and helper classes.
client_base	Generate client-side base class for each type (do not edit)

Target	Description
client_impl	Generate client-side actual class for each type (extends base classes, can be edited)
server_base	Generate server-side base class for each type (do not edit)
server_impl	Generate server-side actual class for each type (extends the base classes, can be edited)
server_model	Special Copy the used information model file to the server-side package

Table 3. Additional generation targets that are combinations of the previous targets

Target	Description
client_all	Combination of client_base, client_impl
server_all	Combination of server_base, server_impl
base_all	Combination of common, client_base, server_base
impl_all	Combination of client_impl, server_impl
all_code	Combination of everything except server_model, i.e. all .java output
all_resources	Currently only server_model, i.e. all other than .java output



If you generate the base and actual (i.e. "impl") targets in different code generation configurations, remember that both are always required.

4.3. Method Implementations

Instead of defining method implementations in the actual classes, the SDK also supports implementing them out of the actual class. Therefore, the Codegen also generates a method interface which can be implemented and plugged into the actual class. Please see the generated classes to learn how this can be done.

4.4. Object Initializations

Additionally, an initializer can be defined for each type, for example to provide initial values. Please see the generated classes to learn how this can be done.

4.5. Configuration Parameters

Codegen can be configured with a few configuration parameters. In the command line version of Codegen, the changes need to be implemented in a configuration XML file. In the Maven plugin version, the parameters are defined in the POM XML file. Please also see the provided examples for more complete examples of the configuration process. The examples for the command line version are in the 'configexamples' folder and the 'maven-integration-sample' Maven project provides examples for configuring the Maven integration.

4.5.1. Model Directories

The model parameters define the directories where Codegen is searching for information models that are used in the code generation. The files must conform to the OPC UA NodeSet2 XML format.

In the command line version, the models are defined in the following way:

```
<models>
  <modelFolder>${app.home}/models</modelFolder>
</models>
```

And in the Maven plugin:

```
<models>
  <fileset>
    <directory>${project.basedir}/models</directory>
    <includes>
      <include>**/*.xml</include>
    </includes>
  </fileset>
</models>
```

4.5.2. Namespace Mapping Parameters

The namespace mapping parameters define how the OPC UA namespaces that are defined in the information models are mapped to their respective Java packages. See chapter [Mapping NamespaceUris to Java Packages](#) for an explanation of the concept. Each mapping consists of the following parameters:

Parameter	Description
uri	The NamespaceUri associated with the information model used as the input of the code generation
packageName	The name of the Java package where the Java code output for the information model is placed

Parameter	Description
prefix	Optional The prefix used in the name of the ServerInformationModel and Ids classes, e.g., prefix value 'Di' outputs the class DiServerInformationModel

In the command line version, the namespace mapping parameters are defined in the following way:

```
<namespaceMappings>
  <namespaceMapping uri="http://ua.prosysopc.com/SampleTypes" packageName="example.packagename"
  prefix="Sample"/>
</namespaceMappings>
```

And in the Maven plugin:

```
<namespaceMappings>
  <namespaceMapping>
    <uri>http://ua.prosysopc.com/SampleTypes</uri>
    <packageName>example.packagename</packageName>
    <prefix>Sample</prefix>
  </namespaceMapping>
</namespaceMappings>
```

4.5.3. Generation Target Parameters

The concept of the generation target is explained in more detail in the chapter [Generation Targets](#).

Parameter	Description
targets	Defines what is the output of the code generation
uri	The NamespaceUri associated with the information model used as the input of the code generation
output	The folder where the code generation output is placed

In the command line version, the generation target parameters are defined in the following way:

```
<generates>
  <generate targets="all_code">
    <uri>http://ua.prosysopc.com/SampleTypes</uri>
    <output>${app.home}/sampletypes/output_code</output>
  </generate>
</generates>
```

And in the Maven plugin:

```
<generates>
  <generate>
    <targets>all_code</targets>
    <uris>
      <uri>http://ua.prosysopc.com/SampleTypes</uri>
    </uris>
    <output>${codegen.output.code}</output>
  </generate>
</generates>
```

5. Using Generated Classes in Applications

5.1. Client-side Applications

For instructions on using the generated classes in client-side applications developed with the Prosys OPC UA Java SDK, please refer to the Prosys OPC UA Java SDK Client Tutorial in the 'tutorial' folder of the distribution package.

5.2. Server-side Applications

For instructions on using the generated classes in server-side applications developed with the Prosys OPC UA Java SDK, please refer to the Prosys OPC UA Java SDK Server Tutorial in the 'tutorial' folder of the distribution package.