

RepoDoc User Guide

www.archimetes.com

Version 2.0.7136.2210, 2017-10-25

Table of Contents

About	1
Installing and running.....	2
Add-in mode	2
Standalone mode	2
Working with application.....	4
Document generator	4
Package browser	6
Application settings.....	6
About dialog	7
License key	8
Understanding templates	9
Objects.....	9
Tags	9
Filters	9
Template examples	11
Iterating packages	13
Iterating elements	14
Obtaining diagram images	16
Redirecting output.....	16
Notes format	17
Generator profiles	17
Escape sequences	18
Post-processing.....	18
Custom queries.....	19
Miscellaneous objects and filters.....	19
Reference	21
Package.....	21
Element.....	22
Diagram	24
DiagramObject	25
Attribute.....	27
Constraint	28
Tag.....	28
Operation	29
Parameter	30
Scenario	30
ScenarioStep	31
Connector.....	32

About

RepoDoc is a powerful document generator for Enterprise Architect able to produce a variety of document formats using templates written in any text editor. These include HTML, LaTeX, Markdown or AsciiDoc documents, but also CSV, XML or JSON files, GraphViz graphs, SVG diagrams and even source codes in different languages. With RepoDoc you can also generate PDF documents easily using the built in post-processing feature.

Installing and running

Download [RepoDoc installer](#) and follow the installation steps on the screen. Please note that RepoDoc has following requirements for running:

- Microsoft Windows 7 or later, (32/64 bit),
- Microsoft .NET Framework 4.5 or later,
- Enterprise Architect v1305 or later installed,



RepoDoc works with following repository types:

- MySQL
- PostgreSQL
- MS SQL Server
- Firebird (database repository or local *.feap filed base repositories)
- Oracle
- JET (local *.eap file based repositories)

For initial setup and configuration of connection to your repository, please follow the Sparx Systems Help.

Once you have finished the installation you can use RepoDoc either as an Enterprise Architect add-in or as a standalone application.

Add-in mode

Start the add-in from the ribbon by navigating to the *Extensions* → *RepoDoc* → *Control panel* option or

using the project browser by right-clicking on a package in the package browser and selecting *Extensions* → *RepoDoc* → *Control panel* option.



The Add-in mode is not supported for EALite edition of Enterprise Architect.

Standalone mode

Standalone application can be started from the command line. When you have your Enterprise Architect installed and configured, you can run RepoDoc from Windows command line with the following command:

```
C:\Program Files (x86)\Archimedes\RepoDoc\RepoDoc.exe [ConnectionString]
```

Start the application with the connection string (or the full path to your .eap or .feap file) to connect

to your repository.

Working with application

This section describes the main application parts with emphasis on the document generator. The generator profiles are discussed in the [Template examples](#) section.

Document generator

RepoDoc follows the same principles as the default document generator included in the Enterprise Architect and as such needs two kinds of inputs to generate a document:

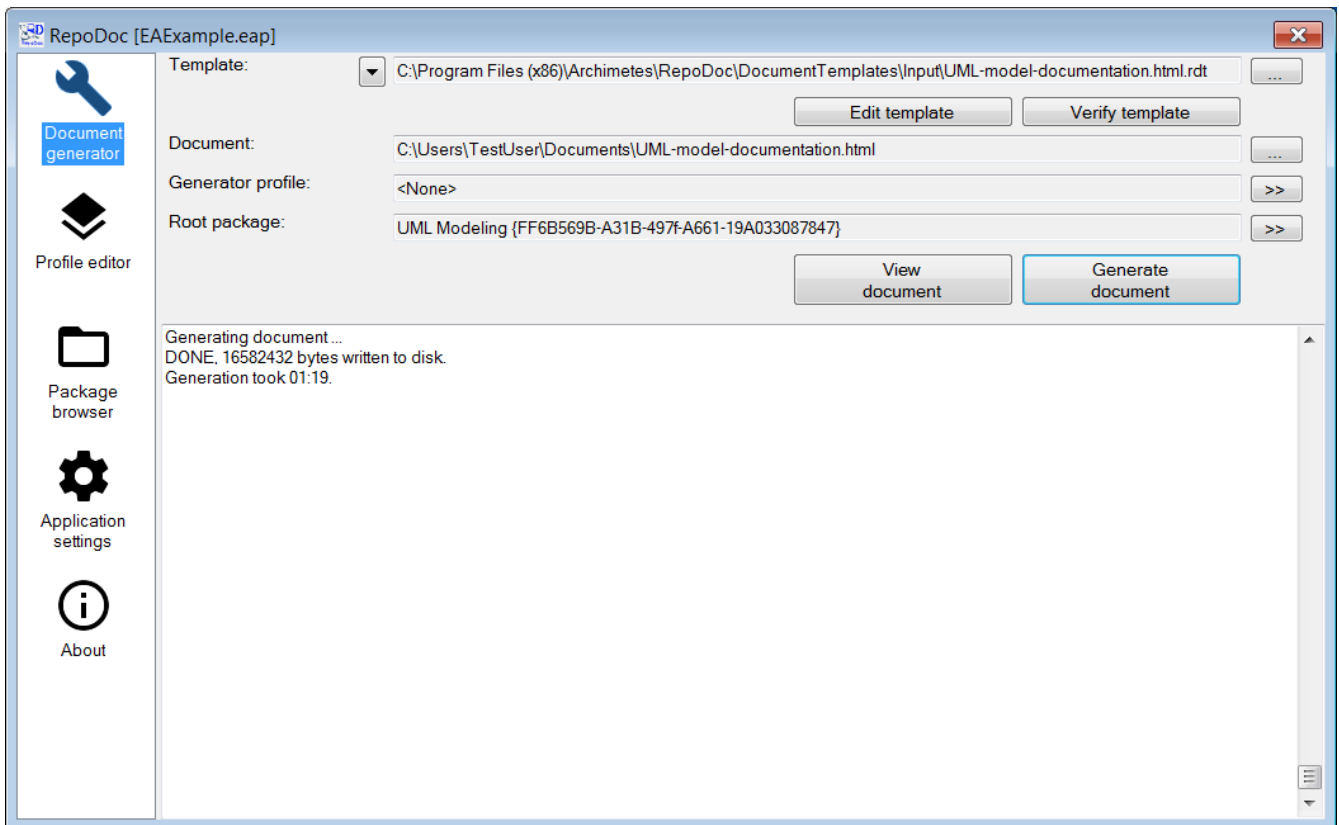
1. Starting point in the repository determining the part of your model you wish to document i.e. a root package.
2. Document template that tells RepoDoc what to take out from the repository and where to put it into the document.



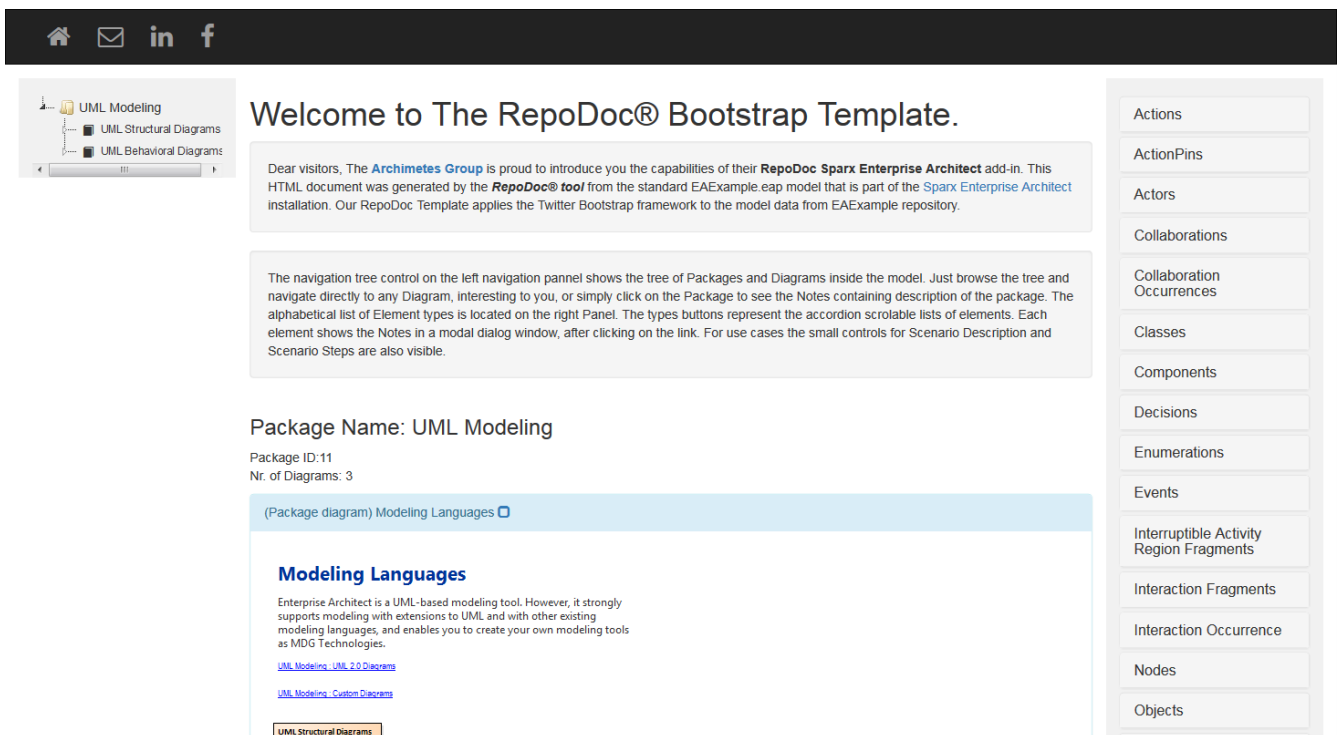
RepoDoc comes with several pre-installed templates. These templates are stored in the `C:\Program Files (x86)\Archimedes\RepoDoc\DocumentTemplates\Input` directory or you can [download the templates](#) from our website. The document templates have `liquid` file extension.

To demonstrate the document generation we'll use the standard `EAExample.eap` model that is shipped with every Enterprise Architect and is typically stored in the `c:\Program Files (x86)\Sparx Systems\EA` directory. To generate a document, based on this model, please follow these steps:

1. Open the `EAExample.eap` model in Enterprise Architect and select the `UML Modeling` package in the Project browser.
2. Right-click on the selected package and choose *Extensions → RepoDoc → Control Panel*. RepoDoc starts and presents itself with the *Document generator* form.
3. Select the *About* dialog and click the `Download` license key button if your are using RepoDoc for the first time. Then switch back to the *Document generator*.
4. Click the `...` button in the first row and select the `UML-model-documentation.html.liquid` template from the dialog.
5. Click the `Generate document` button. RepoDoc generates a HTML documentation for the `UML Modeling` package and outputs information similar to the one pictured below.



The generated document is stored in the **Documents** directory (the path may differ based on your username) and should look like the one pictured below.



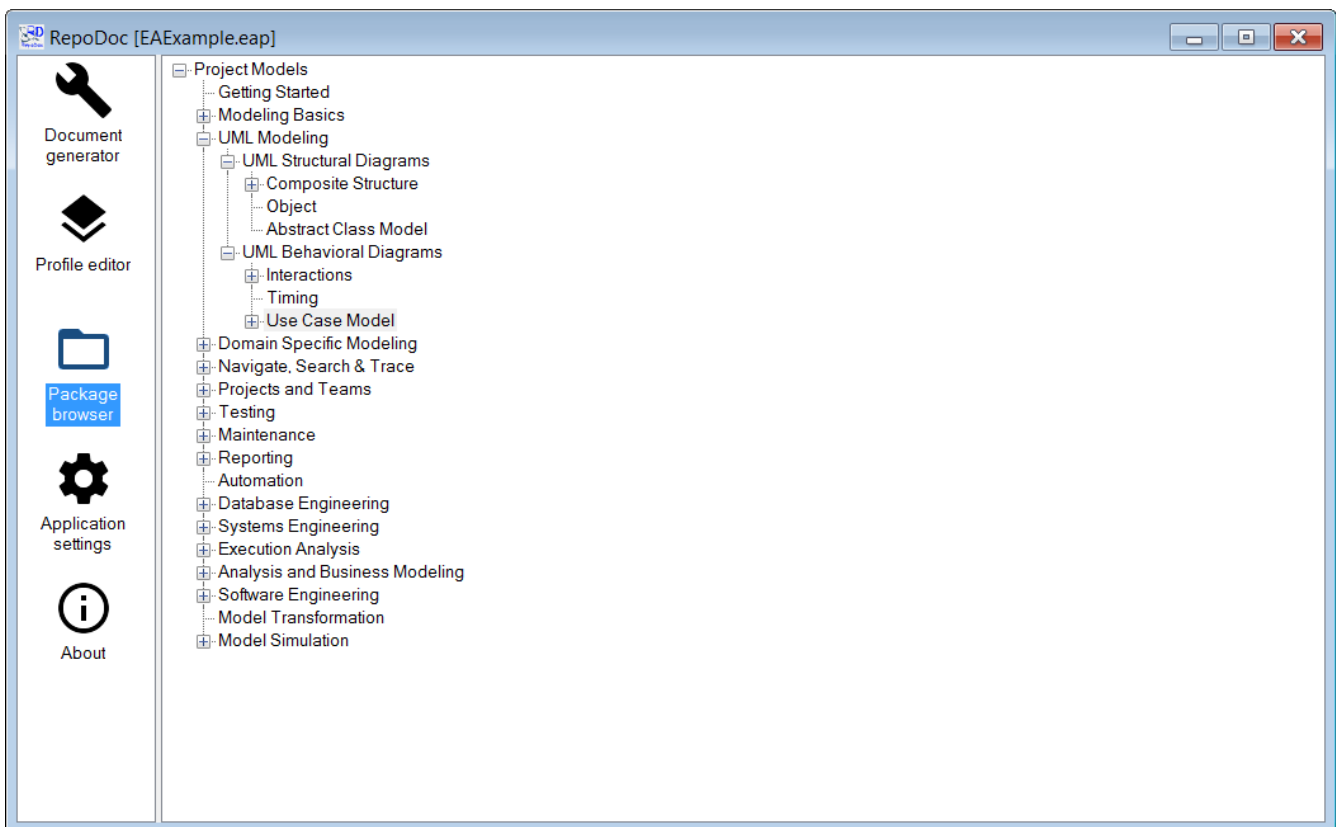
Additional functionality of the document generator includes:

- **Verify template** button starts template verification without generating a document. In this case the connection to the repository is not necessary.
- **Edit template** button opens the selected template in a text editor defined by the user. You can change the path to your favorite editor in the *Application settings*.

- Generator profile button >> navigates to the generator profile editor which allows you to modify the way the repository is processed or to set a document post-processing command.
- Package browser button >> navigates to the Package browser which allows you to choose a different package to document without closing the RepoDoc. The name of the root package is displayed in the textbox together with the package GUID.
- View document button opens the generated document in the associated application.

Package browser

The package browser lets you choose a root package from the model. Please note that the browser is visible only when you invoked RepoDoc with an opened project in the Enterprise Architect or with a `ConnectionString` argument (in case you are using it in standalone mode).

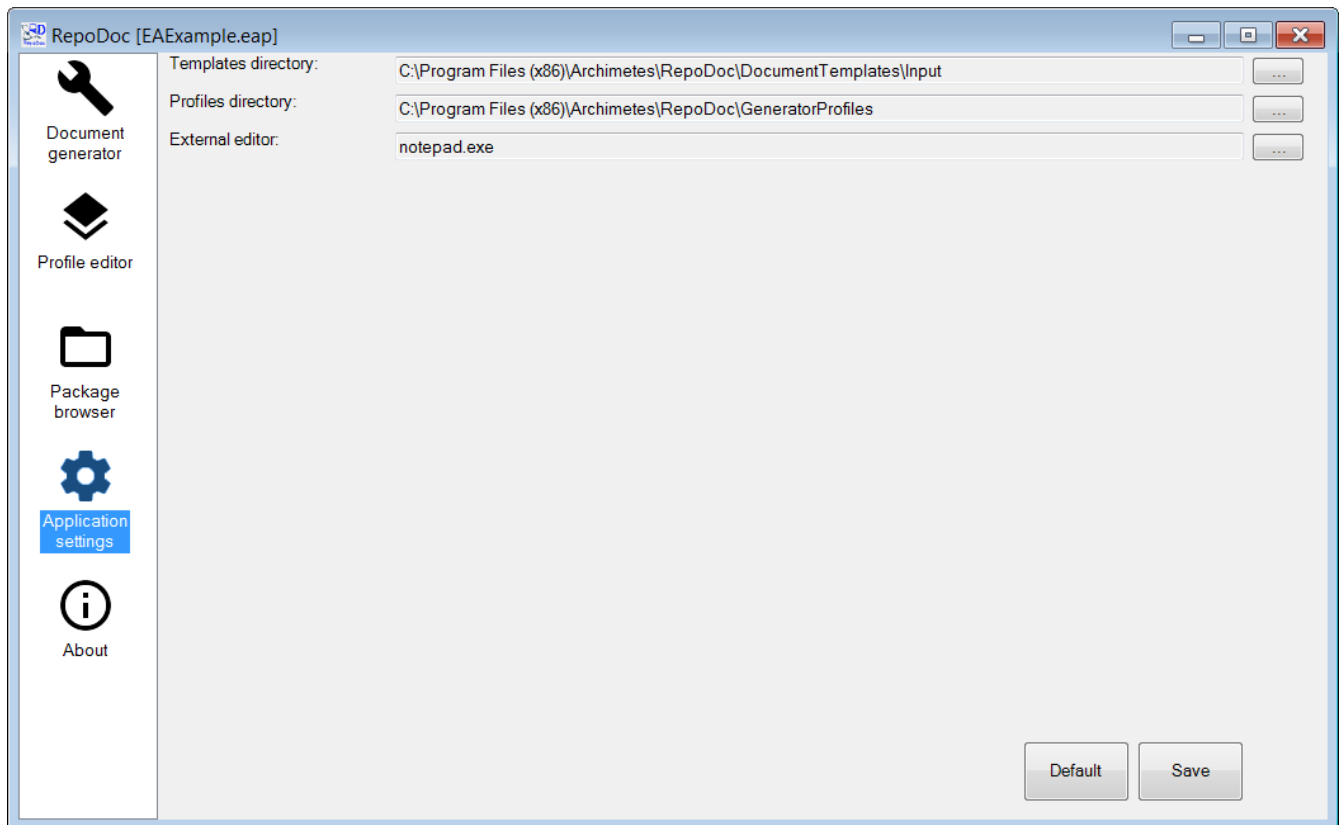


Select a package you wish to start with, right click on it and choose *Set as new root package*.

Application settings

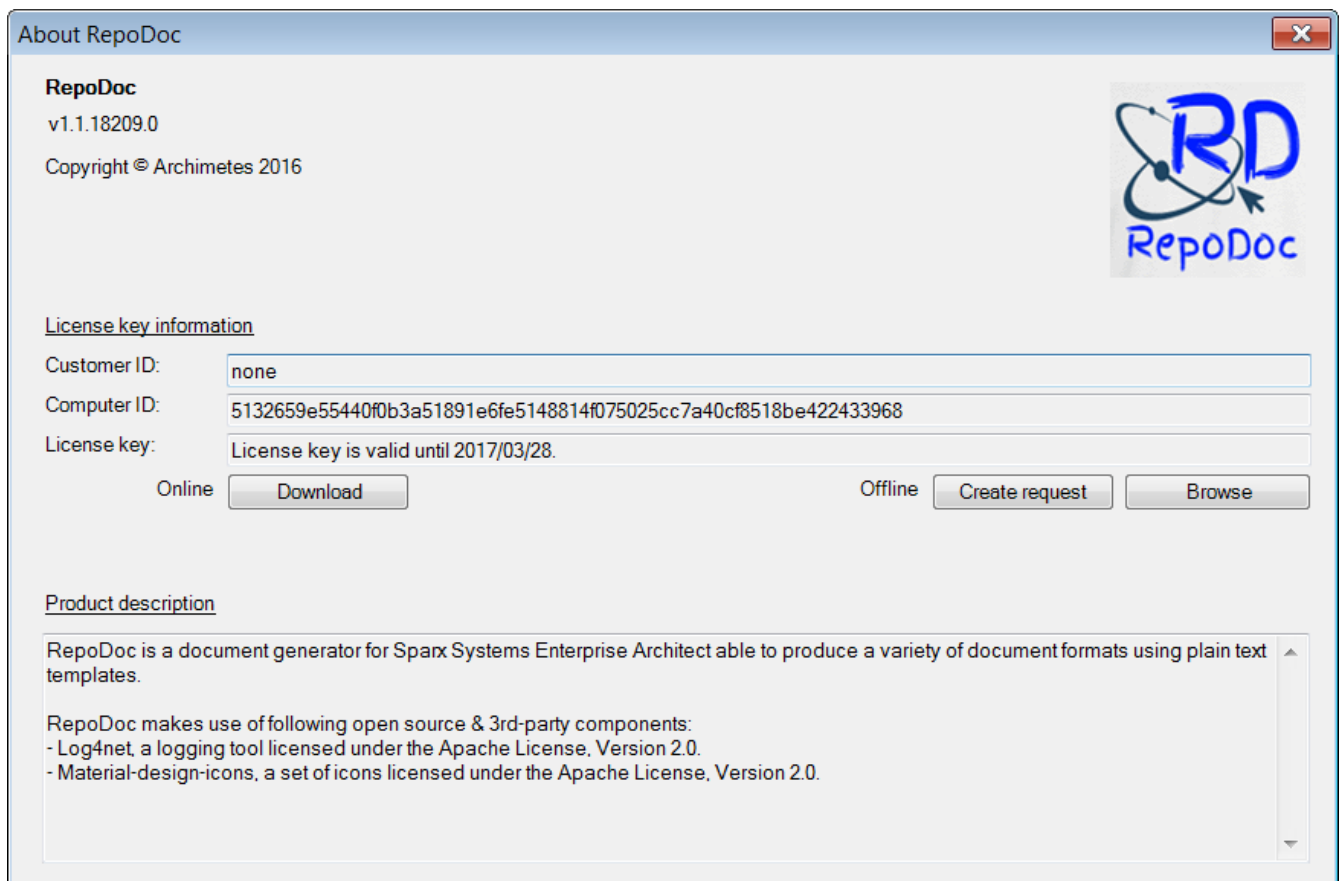
This form contains application settings and allows you to set:

- Folder where you prefer to store your templates. This folder will be then preset when selecting template in the *Document generator* form.
- Folder where you prefer to store your generator profiles. This folder will be then preset when selecting generator profile in the *Profile editor* form.
- Path to the text editor that will be used to open templates when clicking *Edit template* button in the *Document generator* form.



About dialog

The *About* dialog shows product and license key information.



License key

RepoDoc needs a valid license key for document generation. Time limited license key is available for free and can be simply obtained by clicking the **Download** license key button. Please contact info@archimetes.com for further information.

Understanding templates

Every RepoDoc template is a plain text file with following instructions:

1. which repository items to look for (packages, elements, diagrams etc.)
2. what information about these items (name, notes, author, stereotype) to put into the document.

RepoDoc does not use its own template syntax, but instead makes use of [Liquid template language](#). Liquid templates consist of objects, tags, and filters and are briefly described below. For more information, please refer to the [Liquid template language documentation](#).

Objects

Objects represent variables and are denoted by double curly braces: `{{` and `}}`. Objects can be defined by the user or are provided "as is" by RepoDoc. One of the provided objects is the `root_package` object. It represents the package selected by the user in the Project browser and serves usually as the starting point for repository iteration. We'll use this object in the following examples.

This small template below puts the name of the root package into the document.

```
{{ root_package.name }}
```



The RepoDoc makes use of the [Enterprise Architect class model](#) and being familiar with it is an advantage when writing templates.



The RepoDoc uses the so called [snake case](#) for object names. For example `Notes` becomes `notes`, `ParentID` becomes `parent_id` etc. Have a look at [Reference](#) section for a quick overview of all possible objects.

Tags

Tags create the logic and control flow for templates. They are denoted by curly braces and percent signs: `{%` and `%}`. The template below puts the name of root package into the document, if it contains the REQ string.

```
{% if root_package.name contains "REQ" %}  
root_package.notes  
{% endif %}
```

Filters

Filters change the output of a Liquid object. They are used within an output and are separated by a `|`.

```
{{ root_package.name | upcase }}
```

The template above changes the name of the root package to uppercase and puts the result into the document.



Liquid offers a rich set of standard filters to manipulate strings, numbers and arrays. RepoDoc offers some additional filters for accessing and/or manipulating repository items. These additional filters are described in the [Template examples](#) section.












































Template examples

This section contains template examples and focuses on main RepoDoc features. For standard Liquid features please consult the [Liquid template language documentation](#).



For further examples have a look at the RepoDoc pre-installed templates. These templates are stored in the `C:\Program Files (x86)\Archimedes\RepoDoc\DocumentTemplates\Input` directory or you can [download the templates](#) from our website.

The examples below have been created using the `EAExample.eap` model that is shipped with Enterprise Architect. The *WSDL* package from the *Domain Specific Modeling* (see picture below) is used as the root package (`root_package` object).

- ▲  Project Models
 - ▷  Getting Started
 - ▷  Modeling Basics
 - ▷  UML Modeling
 - ▲  Domain Specific Modeling
 -  Domain Based Models
 - ▷  NIEM
 - ▷  BPMN
 - ▷  XML Schema
 - ▲  WSDL
 -  WSDL
 - ▲  «WSDLnamespace» WSDLPackage1
 -  Overview
 - ▲  «XSDschema» Types
 -  Types
 -  «XSDcomplexType» InputParameters
 - ▷  «XSDcomplexType» OutputParameters
 - ▲  Messages
 -  Messages
 - ▷  «WSDLmessage» SampleInput
 - ▷  «WSDLmessage» SampleOutput
 - ▲  PortTypes
 -  PortTypes
 - ▷  «WSDLportType» SamplePortTypeHTTP
 - ▷  «WSDLportType» SamplePortTypeSOAP
 - ▲  Bindings
 -  Bindings
 - ▷  «WSDLbinding» SampleBindingHTTP
 - ▷  «WSDLbinding» SampleBindingSOAP
 - ▲  Services
 -  SampleService
 -  «WSDLservice» SampleService
 - ▲  «WSDL» SampleWSDLFile1
 -  SampleService: SampleService
 - ▷  «WSDLnamespace» WSDLPackage2
 - ▷  «WSDLnamespace» WSDLPackage3
 - ▷  «XSDschema» Schema
 - ▷  GML
 - ▷  ArcGIS
 - ▷  Win32 UI
 - ▷  Entity Relationship
 - ▷  Example UMLProfile
 - ▷  Navigate, Search & Trace

Iterating packages

This basic template outputs the names of root package child packages.

Iterating packages (Input)

```
{% for package in root_package.packages %}
  {{ package.name }}
{% endfor %}
```

Iterating packages (Output)

```
WSDLPackage1

WSDLPackage2

WSDLPackage3

Schema
```

Notice the additional blank lines in the output. These occur because there is a line break right after the `for` tag. In Liquid, you can include a hyphen in your tag `{{- , -}}`, `{%- , and -%}` to strip whitespace from the left or right side of a tag. This is called [whitespace control](#).

Iterating packages with whitespace control (Input)

```
{% for package in root_package.packages -%}
  {{ package.name }}
{% endfor %}
```

Iterating packages with whitespace control (Output)

```
WSDLPackage1
WSDLPackage2
WSDLPackage3
Schema
```

In the example above, only the child packages of the root package are iterated. You can use the `get_packages` filter to get an array of all child packages. The filter works recursively and returns an array of all packages under the given package.

Iterating packages using a filter (Input)

```
{% assign packages = root_package | get_packages -%}
{% for package in packages -%}
  {{ package.name }}
{% endfor %}
```


Iterating packages using a filter (Output)

```
WSDLPackage1
Types
Messages
PortTypes
Bindings
Services
WSDLPackage2
Messages
PortTypes
Bindings
Services
WSDLPackage3
Types
Messages
PortTypes
Bindings
Services
Schema
```



You can specify an optional boolean parameter for the `get_packages` filter.

Example: `{% assign packages = some_package | get_packages: true %}`

If it's true, then the `some_package` is included in the returned array for convenience.

Iterating elements

Elements can be iterated either using the package `elements` property or using the `get_elements` filter.

The template below uses the `elements` property that contains an array of child elements of a package. To shorten the output we'll stop iterating when we reach package `WSDLPackage2`.

Iterating elements (Input)

```
{% assign packages = root_package | get_packages -%}
{% for package in packages -%}
{% if package.name == "WSDLPackage2" -%}{% break %}{% endif -%}
Package name: {{ package.name }}
{% for element in package.elements -%}
{% if element.type != "Note" -%}
  Element name: {{ element.name }}
{% endif -%}
{% endfor -%}
{% endfor %}
```

Iterating elements (Output)

```
Package name: WSDLPackage1
  Element name: SampleWSDLFile1
Package name: Types
  Element name: InputParameters
  Element name: OutputParameters
Package name: Messages
  Element name: SampleInput
  Element name: SampleOutput
Package name: PortTypes
  Element name: SamplePortTypeHTTP
  Element name: SamplePortTypeSOAP
Package name: Bindings
  Element name: SampleBindingHTTP
  Element name: SampleBindingSOAP
Package name: Services
  Element name: SampleService
```

Some elements may have their own child elements representing sub components, ports or interfaces. These can be accessed either by using again the `elements` property, this time for each element or by using the `get_elements` filter. The filter works recursively and returns an array of child elements for the given repository item. It can be used with both packages and elements.

Iterating elements using a filter (Input)

```
{% assign packages = root_package | get_packages -%}
{% for package in packages -%}
{% if package.name == "WSDLPackage2" -%}{% break %}{% endif -%}
Package name: {{ package.name }}
{% assign elements = package | get_elements -%}
{% for element in elements -%}
{% if element.type != "Note" -%}
  Element name: {{ element.name }}
{% endif -%}
{% endfor -%}
{% endfor %}
```

Iterating elements using a filter (Output)

```
Package name: WSDLPackage1
  Element name: SampleWSDLFile1
  Element name: SampleService
Package name: Types
  Element name: InputParameters
  Element name: OutputParameters
Package name: Messages
  Element name: SampleInput
  Element name: SampleOutput
Package name: PortTypes
  Element name: SamplePortTypeHTTP
  Element name: SamplePortTypeSOAP
Package name: Bindings
  Element name: SampleBindingHTTP
  Element name: SampleBindingSOAP
Package name: Services
  Element name: SampleService
```

You can see that the component *SampleWSDLFile1* is now followed by the *SampleService* interface.



You can specify an optional boolean parameter for the `get_elements` filter.

Example: `{% assign elements = some_package | get_elements: true %}`

If it's true then the package element will be included in the returned array.

The so called package element can (also) be accessed by simply using the package `element` property.

Example: `{{ some_package.element }}`.

Obtaining diagram images

Diagram images are accessible via the diagram `image_png` property. This property returns a base64 encoded diagram image in the PNG format. The image can be taken and used "as is" (for example in HTML code) or written to a file using output redirection filters.

HTML template fragment with diagram image.

```

```

Redirecting output

Sometimes it is necessary to redirect output to a file or files. For this purpose, RepoDoc offers following filters:

- `input | write_all_text: filename` filter writes the input to the specified file. The filter assumes that the input is plain text.
- `input | write_all_bytes: filename` filter writes the input to the specified file. The filter assumes that the input is binary.



The file is stored in the same directory as the document.

The `base64_encode` and `base64_decode` filters may be used in conjunction with the `write_all_...` filters to modify the input before writing the data to a file. In the following example each image is first base64 decoded and then written to a file whereby the name of the file contains diagram GUID to ensure that the filename is unique.

Saving diagram images to files.

```
{% assign packages = root_package | get_packages: true %}
{% for package in packages %}
{% for diagram in package.diagrams %}
{% assign filename = "Diagram_" | append: diagram.guid | replace: "{", "" | replace:
"}", "" | append: ".png" %}
{{ diagram.image_png | base64_decode | write_all_bytes: filename }}
{% endfor %}
{% endfor %}
```

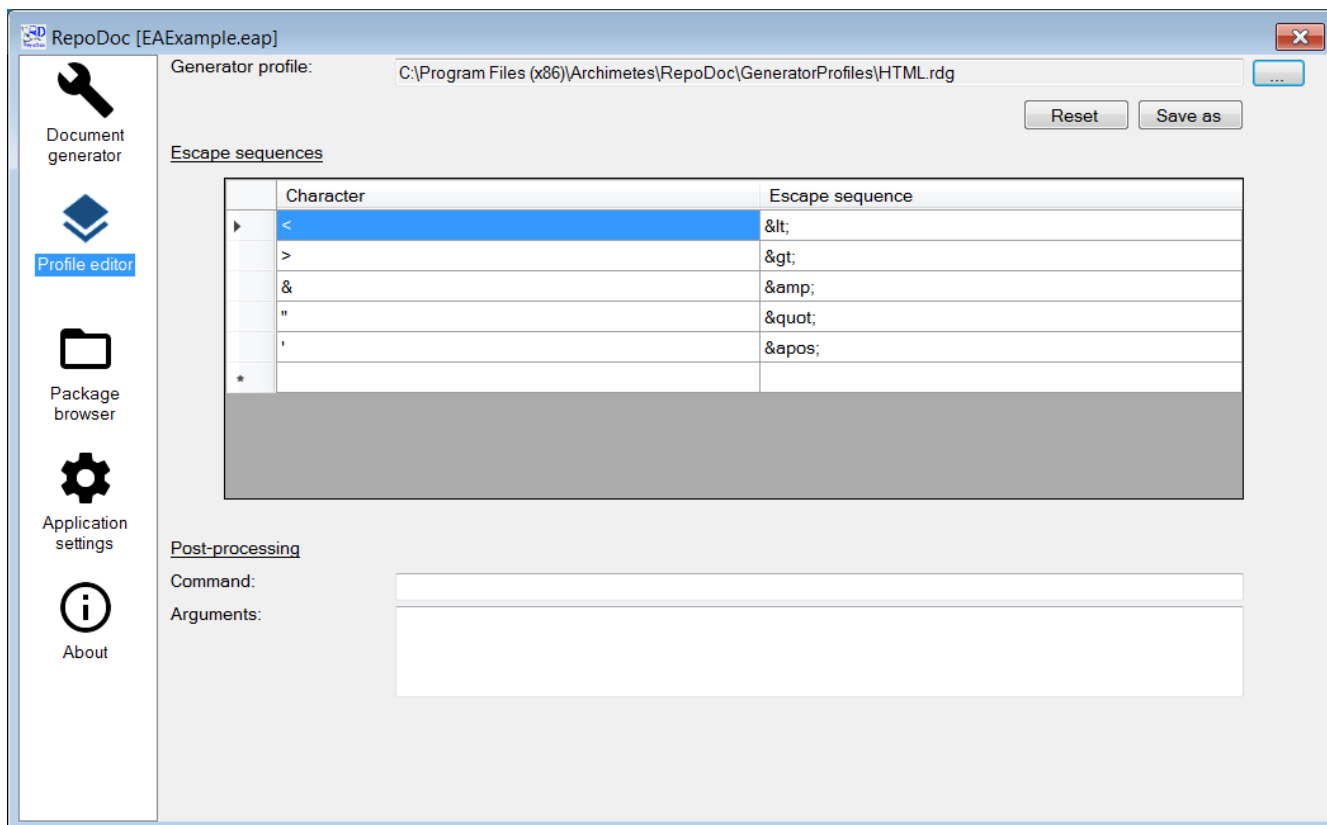
Notes format

Notes for various items may contain format information, like font or color information. RepoDoc provides two objects when dealing with notes and formatting.

- `{{ some_item.notes }}` returns notes in plain text (without formatting). The color or font format information is lost, however the lists are preserved using indents and newlines.
- `{{ some_item.notes_html }}` returns notes in html format with all the formatting expressed using html tags.

Generator profiles

Generator profiles can be used to specify escape sequences for certain characters and/or post-processing command for the generated document. The profile editor lets you create and edit a document generator profile.



RepoDoc comes with several pre-installed profiles, but you are free to create new profiles to meet your needs. You'll find the sample profiles in the installation directory `C:\Program Files (x86)\Archimedes\RepoDoc\GeneratorProfiles`. The generator profiles have `rdg` file extension.

Escape sequences

Some repository items may contain content that is problematic from the output format point of view. For example a package with name `<MyPackage>` breaks formatting when the template is intended for generating a HTML document. Clearly the characters `<` and `>` in the name need to be replaced (in other words escaped) with correct HTML entities `<` and `>`. This can be done automatically during document generation by using a dedicated document generator profile with defined escape sequences. Each profile may contain a list of characters and a corresponding escape sequence for each character. To apply the escaping during document generation simply use the `esc` filter.

HTML template fragment with escaped package name.

```
<h1>{{ package.name | esc }}</h1>
```



Enter `0x09`, `0x0a` or `0x0d` in character column to define escape sequences for the tab, line feed and carriage return characters.

Post-processing

Post-processing command in the profile allows you to define a command and its arguments that

will be started upon successful document generation. This may be any command like archivation, transformation into a different format or simply an upload of the document to your company document store.

Custom queries

Iterating the `root_package` (or `models`) object should provide all information necessary for your document. However, there may be cases when information is required beyond what is covered by RepoDoc. For such cases, RepoDoc offers the `sql_query` filter to perform the user's custom SQL query. The following example demonstrates the usage of this filter.

Getting a list of packages using a custom SQL query.

```
{%- assign statement = "SELECT package_id, name FROM t_package" -%}  
{%- assign table = statement | sql_query -%}  
{%- for row in table.rows -%}  
PackageID={{ row.column_values[0] }}  
Name={{ row.column_values[1] }}  
{% endfor %}
```

The query results are accessible via the `table` object. It offers an array of `rows` and `column_values` for each row. There is also a `column_names` object available (not used in the example above). The example demonstrates just the concept, in this case it would be easier to access the information simply by iterating the `root_package`.



Iterate packages, elements and diagrams to retrieve the needed information whenever possible. Use custom queries only in cases when you need some extra information not covered by RepoDoc.

Miscellaneous objects and filters

RepoDoc provides several useful objects and filters not mentioned in previous sections.

Objects:

- `models` object contains an array of repository models. Use this object if you need to iterate the repository regardless of the selected root package. Example: `{{ models[0].name }}`.
- `template_path` object contains the full path to the currently processed template.
- `document_path` object contains the full path to the document name.

Filters:

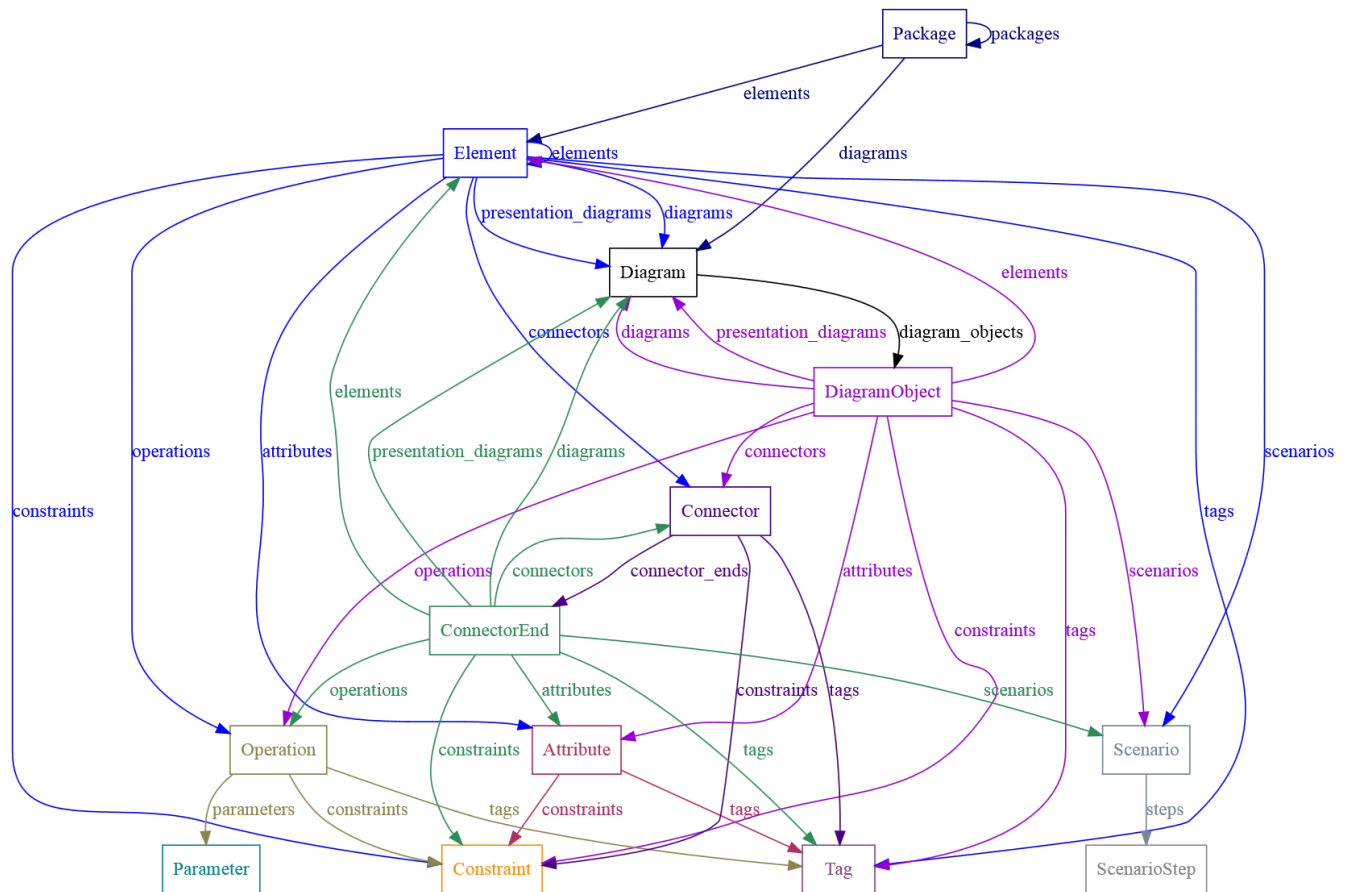
- `get_file_name` filter returns the file name and extension of the specified path string.
- `get_file_name_without_extension` filter returns the file name of the specified path string without the extension.
- `get_directory_name` filter returns the directory information for the specified path string.

- `get_parents` filter returns an array of repository items starting with the top most known parent and ending with the parent of the provided input item.

Reference

This section provides detailed information on all the available objects and classes covered by RepoDoc. The Package class can be considered a top level one and its instance is available via the `root_package` object or the `models` array.

The class names and their relationships correspond to the classes defined in the [Enterprise Architect class model](#). All covered parent-child relationships are pictured below (e.g. a package object may contain one or more element objects accessible using the `elements` property and so on). More over diagram objects and connector ends can be treated like elements (they inherit all element properties).



The properties for each class are listed below.



Property names and their descriptions are taken mostly from the [official Enterprise Architect class model documentation](#) and are provided here for convenience.

Package

Property name	Property description
<code>alias</code>	Package alias.
<code>created_date</code>	The date the package was created.

Property name	Property description
diagrams	List of child diagrams of the package.
element	The associated element object (so called package element). Use this property to get common information such as stereotype, complexity, author, constraints, tagged values and scenarios. Contains null, if the package is a model.
elements	List of child elements of the package.
guid	Package GUID.
id	Package ID.
is_controlled	Indicates if the package has been marked as Controlled.
is_model	Indicates if the package is a model.
is_namespace	Indicates that the package is a Namespace root.
is_protected	Indicates if the package has been marked as Protected.
modified_date	The date the package was modified.
name	The name of the package.
notes	Notes in plain text format (i.e. without formatting).
notes_html	Notes in HTML format.
owner	The package owner when using controlled packages.
packages	List of child packages of the package.
parent_id	The ID of the parent package. Zero indicates that the evaluated package is a model and has no parent.
version	The version of the package.

Element

Property name	Property description
abstract	Indicates if the element is Abstract (1) or Concrete (0).
action_flags	A structure to hold flags concerned with Action semantics.
alias	An alias for the element.

Property name	Property description
attributes	List of attributes defined for the element.
author	The element author.
classifier_id	The ElementID of a Classifier associated with the element; that is, the base type. Only valid for instance type elements (such as Object or Sequence).
complexity	A complexity value indicating how complex the element is; used for metric reporting and estimation. Valid values are: 1 for Easy, 2 for Medium, 3 for Difficult.
connectors	List of connectors defined for the element.
constraints	List of constraints defined for the element.
created_date	The date the element was created.
diagrams	List of child diagrams of the element.
elements	List of child elements of the element.
guid	Element GUID.
id	Element ID.
is_active	Boolean value indicating whether the element is active or not.
is_leaf	Boolean value indicating whether the element is in leaf node or not.
is_root	
is_spec	Boolean value indicating whether the element is a specification or not.
language	The code generation type; for example, Java, C++, C#, VBNet, Visual Basic, Delphi.
modified_date	The date the element was modified.
multiplicity	Multiplicity value for the element.
name	The name of the element.
notes	Notes in plain text format (i.e. without formatting).
notes_html	Notes in HTML format.
operations	List of operations defined for the element.
package_id	ID of the package containing the element.

Property name	Property description
parent_id	ID of the element this element is child of. If it's nonzero then this element is a child element (sub element or an embedded element like port or interface).
persistence	The persistence associated with this element; can be Persistent or Transient.
phase	The phase the element is scheduled to be constructed in; any string value.
presentation_diagrams	List of presentation diagrams (diagrams displaying the element).
run_state	The element's runstate list as a string.
scenarios	List of scenarios defined for the element.
status	The status of the element, such as Proposed or Approved.
stereotype	The primary stereotype of the element.
tags	List of tags defined for the element.
type	The element type (such as Class, Component).
version	The version of the element.
visibility	The Scope of the element within the current Package. Valid values are: Public, Private, Protected or Package.

Diagram

Property name	Property description
diagram_objects	List of diagram objects displayed on the diagram.
guid	Diagram GUID.
id	Diagram ID.
image_png	A base64 encoded diagram image in PNG format.
name	The name of the diagram.
notes	Notes in plain text format (i.e. without formatting).
notes_html	Notes in HTML format.
package_id	The ID of the Package that the diagram belongs to.

Property name	Property description
parent_id	ID of the element the diagram is child of. Contains 0 if the diagram is child of the package.
type	The diagram type for example Activity or Logical.

DiagramObject

Property name	Property description
abstract	Indicates if the element is Abstract (1) or Concrete (0).
action_flags	A structure to hold flags concerned with Action semantics.
alias	An alias for the element.
attributes	List of attributes defined for the element.
author	The element author.
bottom	The bottom edge position of the diagram object in the image.
classifier_id	The ElementID of a Classifier associated with the element; that is, the base type. Only valid for instance type elements (such as Object or Sequence).
complexity	A complexity value indicating how complex the element is; used for metric reporting and estimation. Valid values are: 1 for Easy, 2 for Medium, 3 for Difficult.
connectors	List of connectors defined for the element.
constraints	List of constraints defined for the element.
created_date	The date the element was created.
diagram_id	The ID of the associated diagram where the element is displayed.
diagrams	List of child diagrams of the element.
elements	List of child elements of the element.
guid	Element GUID.
id	Element ID.
is_active	Boolean value indicating whether the element is active or not.

Property name	Property description
is_leaf	Boolean value indicating whether the element is in leaf node or not.
is_root	
is_spec	Boolean value indicating whether the element is a specification or not.
language	The code generation type; for example, Java, C++, C#, VBNet, Visual Basic, Delphi.
left	The left edge position of the diagram object in the image.
modified_date	The date the element was modified.
multiplicity	Multiplicity value for the element.
name	The name of the element.
notes	Notes in plain text format (i.e. without formatting).
notes_html	Notes in HTML format.
operations	List of operations defined for the element.
package_id	ID of the package containing the element.
parent_id	ID of the element this element is child of. If it's nonzero then this element is a child element (sub element or an embedded element like port or interface).
persistence	The persistence associated with this element; can be Persistent or Transient.
phase	The phase the element is scheduled to be constructed in; any string value.
presentation_diagrams	List of presentation diagrams (diagrams displaying the element).
right	The right edge position of the diagram object in the image.
run_state	The element's runstate list as a string.
scenarios	List of scenarios defined for the element.
status	The status of the element, such as Proposed or Approved.
stereotype	The primary stereotype of the element.
tags	List of tags defined for the element.
top	The top edge position of the diagram object in the image.

Property name	Property description
type	The element type (such as Class, Component).
version	The version of the element.
visibility	The Scope of the element within the current Package. Valid values are: Public, Private, Protected or Package.

Attribute

Property name	Property description
allow_duplicates	Indicates if duplicates are allowed in the collection. If the attribute represents a database column this, when set, represents the Not Null option.
constraints	List of constraints defined for the attribute.
container	The container type.
containment	The type of containment - Not Specified, By Reference or By Value.
default	The initial value assigned to this attribute.
guid	Attribute GUID.
id	Attribute ID.
is_collection	Indicates if the current feature is a collection or not. If the attribute represents a database column this, when set, represents a Foreign Key.
is_const	A flag indicating if the attribute is Const or not.
is_derived	Indicates if the attribute is derived (that is, a calculated value).
is_ordered	Indicates if a collection is ordered or not. If the attribute represents a database column this, when set, represents a Primary Key.
is_static	Indicates if the attribute is a static feature or not. If the attribute represents a database column this, when set, represents the Unique option.
length	The attribute length, where applicable.
lower_bound	A value for the collection lower boundary.
name	The name of the attribute.
notes	Notes in plain text format (i.e. without formatting).

Property name	Property description
notes_html	Notes in HTML format.
parent_id	Returns the ElementID of the element that this attribute is a part of.
precision	The precision value.
scale	The scale value.
stereotype	The stereotype for the attribute.
tags	List of tags defined for the attribute.
type	The attribute type (by name; also see ClassifierID).
upper_bound	A value for the collection upper boundary.
visibility	Identifies the scope of the attribute - Private, Protected, Public or Package.

Constraint

Property name	Property description
additional_info	Contains Status in case of Element constraint and Type in case of Operation constraint. Is empty for Attribute constraints.
name	The constraint name.
notes	Notes in plain text format (i.e. without formatting).
notes_html	Notes in HTML format.
parent_id	ID of the parent this constraint belongs to.
type	Constraint type.

Tag

Property name	Property description
guid	Tag GUID.
name	The name of the tag.
notes	Notes in plain text format (i.e. without formatting).
notes_html	Notes in HTML format.
parent_id	ID of the parent this tag belongs to.

Property name	Property description
value	The value assigned to this tag.

Operation

Property name	Property description
abstract	A flag indicating if the operation is abstract (1) or not (0).
behavior	Some further explanatory behavior notes (for example, pseudocode).
classifier_id	The Classifier ID that applies to the ReturnType.
code	An optional field to hold the operation code (used for the Initial Code field).
concurrency	Indicates the concurrency type of the method.
constraints	List of constraints defined for the operation.
guid	Operation GUID.
id	Operation ID.
is_const	A flag indicating that the operation is Const.
is_leaf	A flag to indicate if the operation is Leaf (cannot be overridden).
is_pure	A flag indicating that the operation is defined as Pure in C++.
is_query	A flag to indicate if the operation is a query (that is, does not alter Class variables).
is_root	A flag to indicate if the operation is Root.
is_static	A flag to indicate a static operation.
is_synchronized	A flag indicating a Synchronized operation call.
name	The name of the operation.
notes	Notes in plain text format (i.e. without formatting).
notes_html	Notes in HTML format.
parameters	List of parameters defined for the operation.
parent_id	Returns the ElementID of the element that the operation belongs to.
return_is_array	A flag to indicate that the return value is an array.

Property name	Property description
return_type	The return type for the operation; this can be a primitive data type or a Class or Interface type.
state_flags	Some flags as applied to operations in State elements.
stereotype	The operation stereotype.
tags	List of tags defined for the operation.
visibility	The operation scope - Public, Protected, Private or Package.

Parameter

Property name	Property description
allow_duplicates	Indicates if duplicates are allowed in the collection.
classifier_id	A ClassifierID for the parameter, if known.
default	A default value for this parameter.
guid	Parameter GUID.
is_const	A flag indicating that the parameter is Const (cannot be altered).
is_ordered	Indicates if a collection is ordered or not.
kind	The parameter kind - in, inout, out, or return.
lower_bound	A value for the collection lower boundary.
name	The name of the parameter.
notes	Notes in plain text format (i.e. without formatting).
notes_html	Notes in HTML format.
operation_id	Operation ID of the operation that this parameter belongs to.
type	The parameter type; can be a primitive type or a defined classifier.
upper_bound	A value for the collection upper boundary.

Scenario

Property name	Property description
<code>basic_path_branch_level</code>	Branch level in the basic path scenario if this scenario is an alternative/exception scenario. Returns empty string if this scenario is a basic path scenario.
<code>basic_path_merge_level</code>	Merge level in the basic path scenario if this scenario is an alternative/exception scenario. Returns the string "End" if this is an exception scenario or an alternative that does not join its basic path. Returns empty string if this scenario is a basic path scenario.
<code>basic_path_name</code>	Name of the basic path scenario if this scenario is an alternative/exception scenario. Returns the name of itself if this is scenario is a basic path scenario.
<code>guid</code>	Scenario GUID.
<code>name</code>	Scenario name.
<code>notes</code>	Notes in plain text format (i.e. without formatting).
<code>notes_html</code>	Notes in HTML format.
<code>parent_id</code>	Element ID of the element this scenario belongs to.
<code>steps</code>	List of steps defined for the scenario.
<code>type</code>	The scenario type (for example, Basic Path).

ScenarioStep

Property name	Property description
<code>action</code>	Identifies the action specified for the step.
<code>guid</code>	Scenario step GUID.
<code>level</code>	The number of the step as shown in the scenario editor.
<code>results</code>	Any results that are given from the step.
<code>scenario_guid</code>	GUID of the scenario that the step is a part of.
<code>state</code>	A description of the state the system enters when the step is executed.
<code>type</code>	Identifies whether the step is being performed by a user or the system.

Property name	Property description
uses	The input and requirements that are relevant to the step.

Connector

Property name	Property description
alias	Connector alias.
connector_ends	List of connector ends defined for the connector.
constraints	List of constraints defined for the connector.
dest_id	The ElementID of the element at the target end of this connector.
direction	The connector direction, which can be set to one of the following: Unspecified, Bi-Directional, Source → Destination, Destination → Source.
guid	Connector GUID.
id	Connector ID.
is_leaf	A flag indicating that the connector is a leaf.
is_root	A flag indicating that the connector is a root.
is_specification	A flag indicating that the connector is a specification.
name	The name of the connector.
notes	Notes in plain text format (i.e. without formatting).
notes_html	Notes in HTML format.
source_id	The ElementID of the element at the source end of this connector.
stereotype	The stereotype for the connector.
tags	List of tags defined for the connector.
type	The connector type; valid types are held in the t_connectortypes table.

ConnectorEnd

Property name	Property description
abstract	Indicates if the element is Abstract (1) or Concrete (0).

Property name	Property description
<code>action_flags</code>	A structure to hold flags concerned with Action semantics.
<code>aggregation</code>	The type of Aggregation as it applies to the connector end; valid values are: 0 = None, 1 = Shared, 2 = Composite.
<code>alias</code>	An alias for the element.
<code>allow_duplicates</code>	For multiplicities greater than 1, indicates that duplicate entries are possible.
<code>attributes</code>	List of attributes defined for the element.
<code>author</code>	The element author.
<code>cardinality</code>	The cardinality associated with the particular connector end.
<code>classifier_id</code>	The ElementID of a Classifier associated with the element; that is, the base type. Only valid for instance type elements (such as Object or Sequence).
<code>complexity</code>	A complexity value indicating how complex the element is; used for metric reporting and estimation. Valid values are: 1 for Easy, 2 for Medium, 3 for Difficult.
<code>connectors</code>	List of connectors defined for the element.
<code>constraint</code>	A constraint that can be applied to the particular connector end.
<code>constraints</code>	List of constraints defined for the element.
<code>containment</code>	The containment type applied to the particular connector end.
<code>created_date</code>	The date the element was created.
<code>derived</code>	Indicates that the value of the particular connector end is derived.
<code>derived_union</code>	Indicates the value of the role derived from the union of all roles that subset this.
<code>diagrams</code>	List of child diagrams of the element.
<code>elements</code>	List of child elements of the element.
<code>end_stereotype</code>	Gets the stereotype for the particular connector end.
<code>end_visibility</code>	The Scope associated with the particular connector end - Public, Private, Protected or Package.

Property name	Property description
guid	Element GUID.
id	Element ID.
is_active	Boolean value indicating whether the element is active or not.
is_changeable	Flag indicating whether the particular end is changeable or not - frozen, addOnly or none.
is_leaf	Boolean value indicating whether the element is in leaf node or not.
is_navigable	A flag indicating that the particular connector end is navigable from the other end.
is_root	
is_spec	Boolean value indicating whether the element is a specification or not.
language	The code generation type; for example, Java, C++, C#, VBNet, Visual Basic, Delphi.
modified_date	The date the element was modified.
multiplicity	Multiplicity value for the element.
name	The name of the element.
navigability	Indicates whether the role of an association is navigable from the opposite classifier - Navigable, Non-Navigable or Unspecified.
notes	Notes in plain text format (i.e. without formatting).
notes_html	Notes in HTML format.
operations	List of operations defined for the element.
ordering	Ordering for the particular connector end.
owned	Indicates that the Association end corresponds to an attribute on the opposite end of the Association.
package_id	ID of the package containing the element.
parent_id	ID of the element this element is child of. If it's nonzero then this element is a child element (sub element or an embedded element like port or interface).
persistence	The persistence associated with this element; can be Persistent or Transient.

Property name	Property description
phase	The phase the element is scheduled to be constructed in; any string value.
presentation_diagrams	List of presentation diagrams (diagrams displaying the element).
qualifier	A qualifier that can apply to the particular connector end.
role	The particular connector end role.
role_note	Notes associated with the role of the particular connector end.
role_type	The role type applied to the particular end of the connector.
run_state	The element's runstate list as a string.
scenarios	List of scenarios defined for the element.
status	The status of the element, such as Proposed or Approved.
stereotype	The primary stereotype of the element.
tags	List of tags defined for the element.
type	The element type (such as Class, Component).
version	The version of the element.
visibility	The Scope of the element within the current Package. Valid values are: Public, Private, Protected or Package.