RepoDoc User Guide

www.archimetes.com

Version 1.2.22544.2075, 2017-07-28

Table of Contents

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)\Archimetes\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.

Generating documents

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 a document.

> RepoDoc comes with several pre-installed templates. These templates are stored in the `C:\Program Files (x86)\Archimetes\RepoDoc\DocumentTemplates\Input` directory or you can download the templates from our website. The document templates have `rdt` 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.rdt` 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.
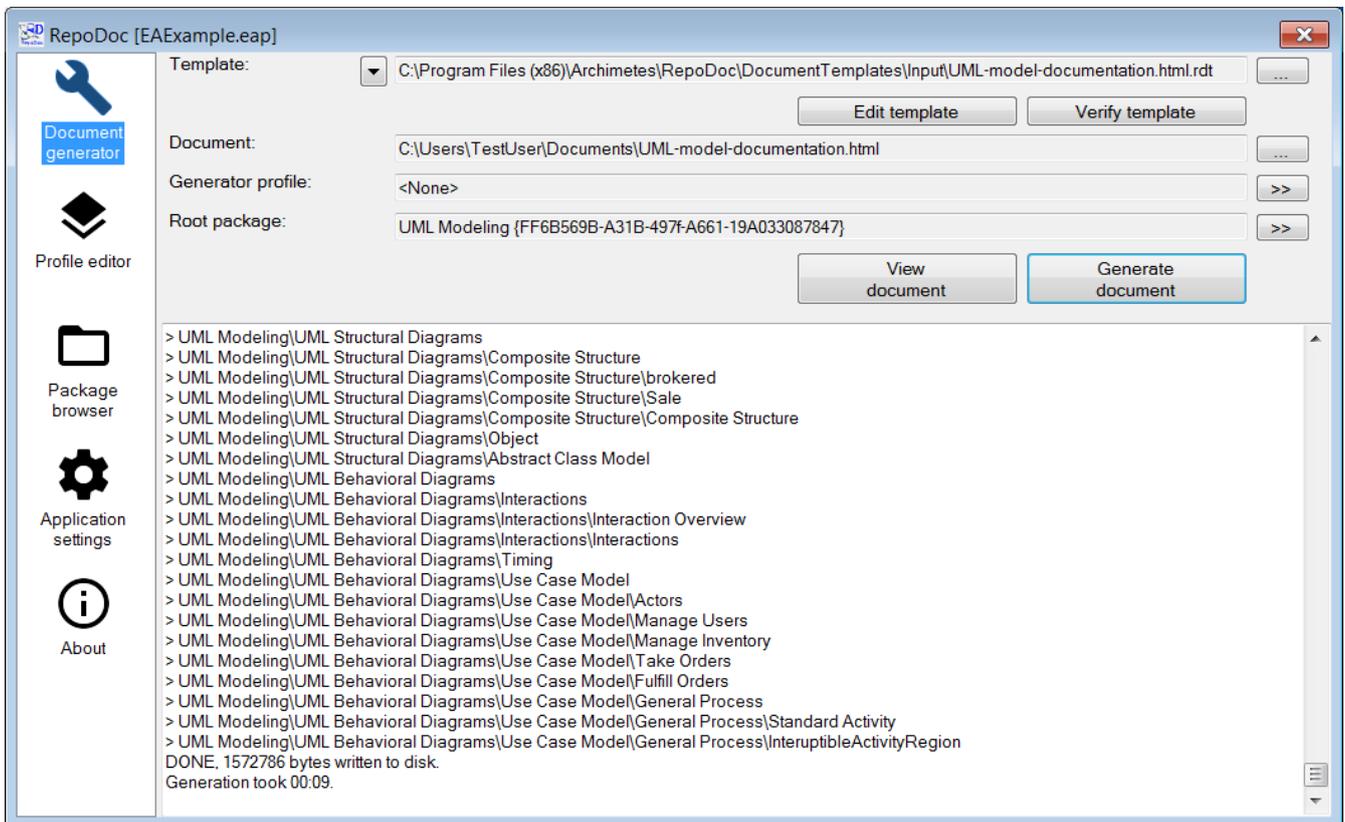
The generated document is stored in the `Documents` directory (the path may differ based on your username) and should look like the one pictured below.
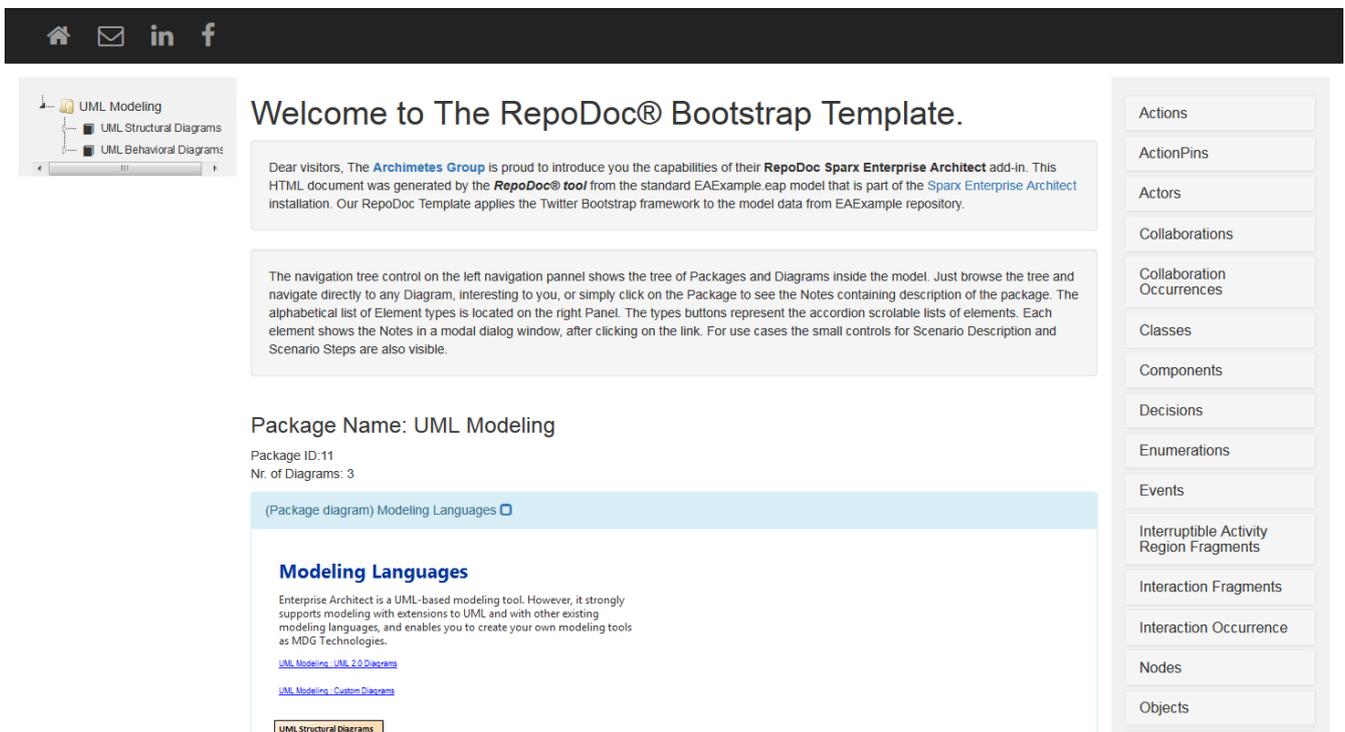
Additional functionality of the document generator includes:

- `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 (please see the Application settings for further details).

- 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.

Editing templates

RepoDoc templates are just plain text files and you can use any text editor to edit them. To make the authoring and reading of the templates easier, syntax highlighting and keyword autocompletion for RepoDoc templates is currently implemented for Notepad editor. Follow these instructions to install syntax highlighting for RepoDoc templates in Notepad:

- start Notepad++,

- select *Language → Define Your Language → Import* and import the file RepoDocTemplateUDL.xml. You'll find the file in c:\Program Files (x86)\Archimetes\RepoDoc\NppExtensions directory,

- restart Notepad++ and after it a new Language "RepoDocTemplate" should appear inside the Language menu,

- open any RepoDoc template, it should colour the syntax as on the screenshot above.

Follow these instructions to install keyword autocompletion for RepoDoc templates in Notepad++:

- stop Notepad++ if it's running,

- copy RepoDocTemplate.xml c:\Program Files (x86)\Archimetes\RepoDoc\NppExtensions into <Notepad++ root folder>\plugins\API\ directory,

- start Notepad++ and open any RepoDoc template. The output should look like the one pictured below.

> Change the path to your favorite editor in the *Application settings*. Your editor will be used to open the templates when clicking *Edit template* button in the *Document generator* form.

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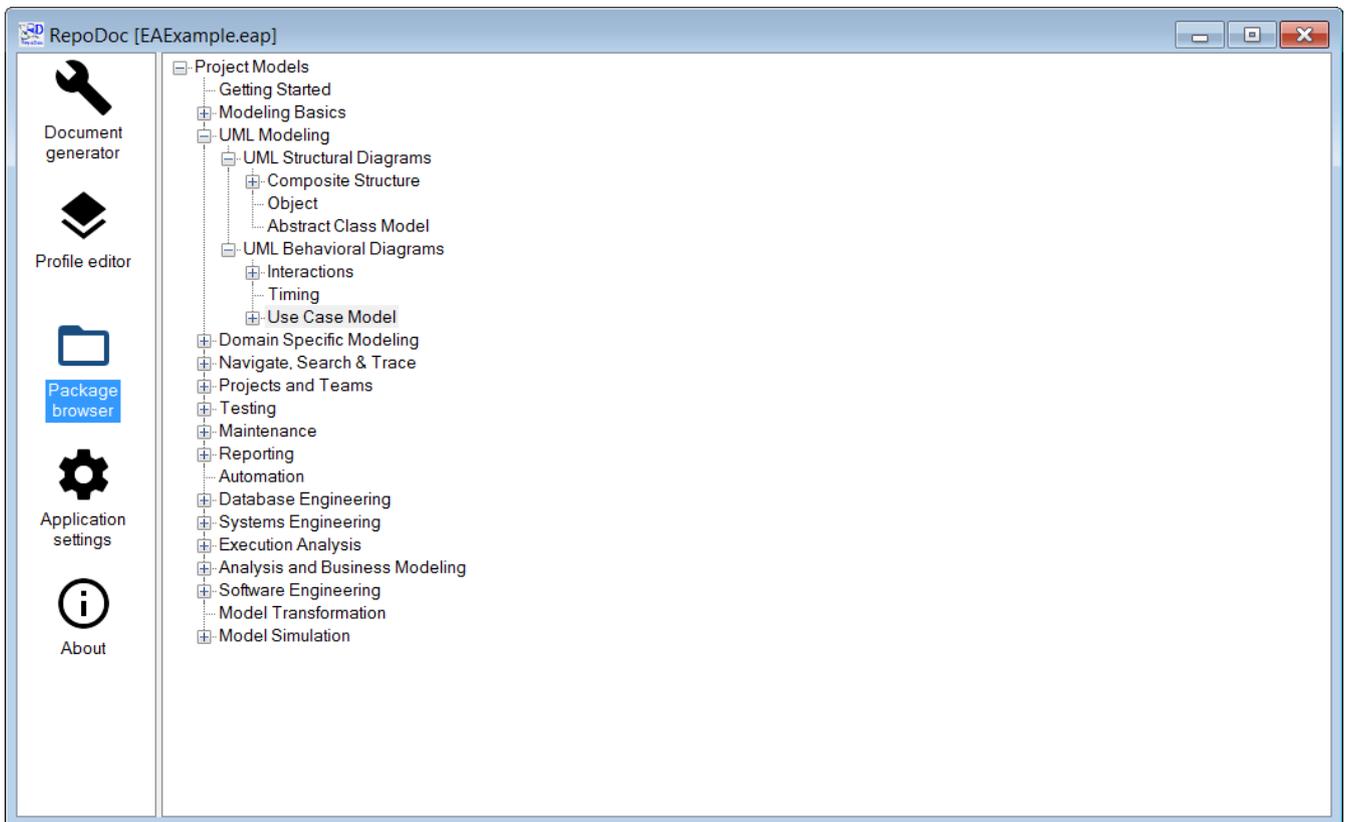
Select a package you wish to start with, right click on it and choose *Set as new root package*.

License key

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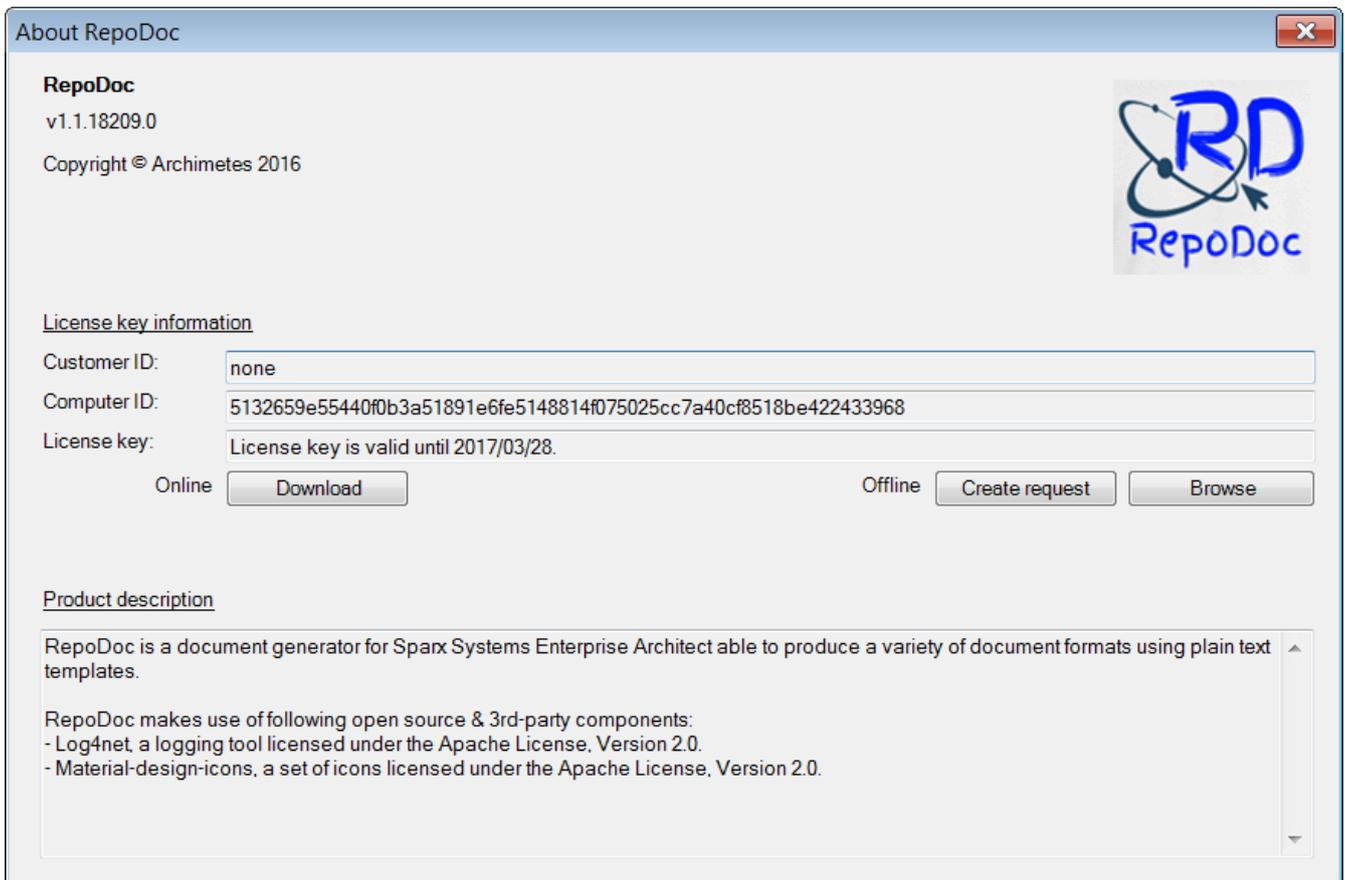
Time limited license key is available for free and can be simply obtained by clicking the `Download` license key button. Please contact info@archimetes.com for further information.

Creating templates

RepoDoc comes with several pre-installed document templates. You can modify these templates or create new templates easily in any text editor. The RepoDoc **template is just a 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**

For this purpose, the template is divided into one or more **sections** (which repository items to look for) whereby each section may contain (among some text) one or more **variables** (what information about these items to put into the document).

> The RepoDoc makes use of the Enterprise Architect class model and being familiar with it is an advantage when writing templates.

A small template example follows:

This is our first RepoDoc template example, containing one Package section and one Element section.

```
[Package]
Package name is "$Package.Name"
[Element $Element.Type=="Class"]
Element name is "$Element.Name"
[/Element]
[/Package]
```

This template tells RepoDoc to do following:

1. Put the name of each package you encounter in the repository into the document.
2. Put the name of each element (that is type of *Class*) in each encountered package into the document.

> Sections are written with an opening and a closing tag, with the content in between. Variables are marked with the $ (dollar) sign.

When iterating repository, RepoDoc begins always with the root package you've selected and then continues with all of its child packages. Long story short, taking the `EAExample.eap` model (that's shipped with Enterprise Architect) and selecting *XML Schema* as the root package

we get following document:

This is our first RepoDoc generated document based on our first template example.

```
Package name is "XML Schema"
Package name is "Schema"
Element name is "Account"
Element name is "LineItem"
Element name is "Order"
Element name is "ShoppingBasket"
Element name is "StockItem"
Element name is "Transaction"
Package name is "Schema_A"
Element name is "fName"
Element name is "lName"
Package name is "Schema_B"
Element name is "mName"
Element name is "Person"
```

The document contains the names of the packages and elements under the *XML Schema* package (including the root package itself). As you can see, the content of `Package` section has been evaluated (put into the document) for each package and the same applies to the `Element` section. This evaluation logic is the same for (almost) all sections RepoDoc works with. The content of the section is evaluated as many times, as there are corresponding items in the repository (in our example these items are the packages and elements). You can see the results of the evaluation procedure in the document below. It contains the packages (as in the Model) and the elements in each encountered package are also named. The *OrderStatus* element, being an *Enumeration*, is not in the document as we requested only *Class* type elements via a filter attribute (don't worry, we'll get to filter attributes later).

Please observe that the lines containing section tags have not been put into the document. RepoDoc automatically removes lines consisting purely of section tags and whitespace characters. Following templates:

```
[Package]
Package name is "$Package.Name"
                [Element $Element.Type=="Class"]
Element name is "$Element.Name"
   [/Element]
  [/Package]
```

```
[Package]Package name is "$Package.Name"
[Element $Element.Type=="Class"]
Element name is "$Element.Name"
[/Element][/Package]
```

produce therefore the same output as our first template example.

By now, you have learned that **using sections, you can control, what parts of the repository will**

be documented because section content is evaluated and put into the document for each corresponding item (package, element, diagram etc.) found in the repository.

Summarizing the above, you can document attribute names for each element. Looking at the sample `EAExample.eap` model

you can see that many classes have some attributes defined. You can document those by adding an `Attribute` section to the template:

This is our second RepoDoc template example, containing one Package section, one Element section and one Attribute section.

```
[Package]
Package name is "$Package.Name"
[Element $Element.Type=="Class"]
Element name is "$Element.Name"
[Attribute]
Attribute name is "$Attribute.Name"
[/Attribute]
[/Element]
[/Package]
```
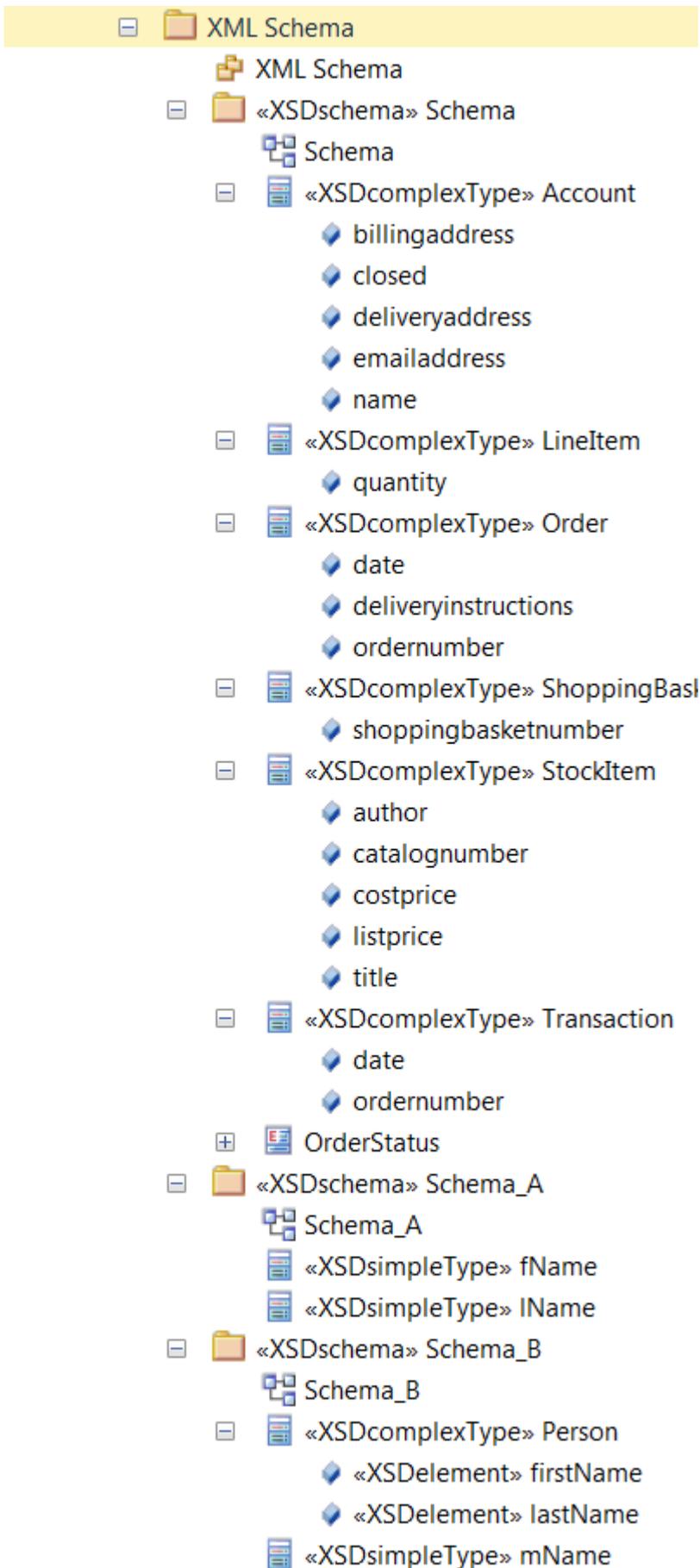
Using this template you get a slightly larger document.

This is our second RepoDoc generated document based on our second template example.

```
Package name is "XML Schema"
Package name is "Schema"
Element name is "Account"
Attribute name is "billingaddress"
Attribute name is "closed"
Attribute name is "deliveryaddress"
Attribute name is "emailaddress"
Attribute name is "name"
Element name is "LineItem"
Attribute name is "quantity"
Element name is "Order"
Attribute name is "date"
Attribute name is "deliveryinstructions"
Attribute name is "ordernumber"
Element name is "ShoppingBasket"
Attribute name is "shoppingbasketnumber"
Element name is "StockItem"
Attribute name is "author"
Attribute name is "catalognumber"
Attribute name is "costprice"
Attribute name is "listprice"
Attribute name is "title"
Element name is "Transaction"
Attribute name is "date"
Attribute name is "ordernumber"
Package name is "Schema_A"
Element name is "fName"
Element name is "lName"
Package name is "Schema_B"
Element name is "mName"
Element name is "Person"
Attribute name is "firstName"
Attribute name is "lastName"
```

As of now we have identified all **three basic parts of a template** that are summarized below.

Sections

Sections are part of the template written with an opening tag and a closing tag, with the content in between. A section tag is composed of the name of the section, surrounded by square brackets. A closing tag also has a slash after the opening bracket, to distinguish it from the opening tag. Each section may contain either ordinary text, variables or one or more subsections or combination of the former (of course there are some rules but we'll get to them later). There are some special purpose sections, but typically the section names correspond to the classes in the Enterprise Architect class model. Following the class model and relationships defined for the classes, it's clear that there must be some similar hierarchy of sections and rules exist on how the sections may be nested within each other. For example a package may contain one or more elements or diagrams,

an element may have attributes or operations (which may in turn have parameters) and so on. Since there are many classes defined by the Enterprise Architect so there are many sections you may use. Have a look at Template syntax reference for all currently supported sections.

Sections with a corresponding class are evaluated for each corresponding *item* found in the repository. There are however some specific points to be addressed as how these *items* are found and what they can represent.

Packages

`Package` section is evaluated for each package found in the repository starting with the root package defined by the user. If a package has child packages, they are evaluated before evaluating following package on the same level. User may choose to exclude some packages from the final document by specifying one or more filter attributes for the `Package` section.

Elements

`Element` section is evaluated for each element contained in or associated with the package. At first, this is straightforward, but there's more information in the model as the Project browser shows. Technically speaking, the element section contains:

1. The associated element object (or so called package element) from the Package class.
2. Element collection defined in the Package class.

Modifying our first example, we will document all elements (please note the filter attribute has been removed) with their types:

This is 3rd RepoDoc template example, containing one Package section and one Element section without the filter attribute.

```
[Package]
Package name is "$Package.Name"
[Element]
Element name is "$Element.Name" and element type is "$Element.Type"
[/Element]
[/Package]
```

we get following document:

```
Package name is "XML Schema"
Element name is "XML Schema" and element type is "Package"
Package name is "Schema"
Element name is "Schema" and element type is "Package"
Element name is "" and element type is "Note"
Element name is "Account" and element type is "Class"
Element name is "LineItem" and element type is "Class"
Element name is "Order" and element type is "Class"
Element name is "OrderStatus" and element type is "Enumeration"
Element name is "ShoppingBasket" and element type is "Class"
Element name is "StockItem" and element type is "Class"
Element name is "Transaction" and element type is "Class"
Package name is "Schema_A"
Element name is "Schema_A" and element type is "Package"
Element name is "fName" and element type is "Class"
Element name is "lName" and element type is "Class"
Package name is "Schema_B"
Element name is "Schema_B" and element type is "Package"
Element name is "" and element type is "Note"
Element name is "mName" and element type is "Class"
Element name is "Person" and element type is "Class"
```

The result is now slightly different and shows not only how the sections work, but also how the information about elements within Enterprise Architect is stored:

1. The document contains the associated element for each package. It has the same name as the package itself, but has a distinct type (see the **Type** property of the Element class). The package element is always documented before other elements contained in the package on the same level.

2. The document contains elements with empty name of type "Note". This is because Notes in the diagrams are internally represented as nameless elements and Enterprise Architect does not show them directly in the Project browser.

> Simply use filter attributes to filter out the package element or other elements you do not need, like those representing Notes in the diagrams.

Elements containing elements (child elements)

Some elements may have child elements representing sub components, ports or interfaces. There is no special section for child elements in RepoDoc instead the `Element` section evaluates not only the element contained in the package but also all of its child elements before evaluating next element (in the package). Looking at the model below.

You can see that three elements in the Components package have one child element (in this case it's a Port). Starting from the highlighted *Components* package and using the third template you get following document:

```
Package name is "Components"
Element name is "Components" and element type is "Package"
Element name is "Charging Unit - ADP2291" and element type is "Class"
Element name is "charger" and element type is "Port"
Element name is "Codec with Amplifier - TLV320AIC3107" and element type is "Class"
Element name is "Li-Ion Battery Monitoring System - AD7230" and element type is
"Class"
Element name is "chargeMonitor" and element type is "Port"
Element name is "Memory - MT42L32M64D2KH-25" and element type is "Class"
Element name is "Panasonic Li-Ion CGR18650AF" and element type is "Class"
Element name is "chargeIndicator" and element type is "Port"
Element name is "Processor - TMS320VC5507" and element type is "Class"
Element name is "RS232" and element type is "Class"
Element name is "Touch-screen" and element type is "Class"
Element name is "USB - PL-2528" and element type is "Class"
```

You can see that the *charger* child element has been evaluated before *Codec with Amplifier* element, because *charger* is child element of *Charging Unit* element. The same applies for child elements *chargeMonitor* and *chargeIndicator*.

Diagrams

Diagrams can be documented using the `Diagram` section. Diagrams may be contained in a package or in an element and the same applies to the `Diagram` section.

Diagram images

Diagram images may be obtained in two ways:

- Using the `$Diagram.ImagePng` variable which evaluates to a base64 encoded diagram image in PNG format.

- Using an empty `Diagram` section. Normally empty sections (containing only opening and closing tag) do nothing, but for `Diagram` section, this is different. An empty `Diagram` section (`[Diagram][/Diagram]`) causes the image (in PNG format) to be written directly to a file. The file is stored in the same directory as the document and its name follows this naming convention: `Diagram_<DiagramID>.png`.

Diagram objects

Each diagram typically has one or more objects. These diagram objects may be documented using the `DiagramObject` section. The `DiagramObject` section is evaluated for each diagram object in the diagram. The variables defined in the `DiagramObject` section (and its subsections) are (almost) identical to the variables defined for the `Element` section. This is because each diagram object is in fact just an element modelled somewhere in the repository. Additionally the object contains position information. Example below documents diagram object names in all diagrams and in all packages (starting from the root package).

This is RepoDoc template example containing one Package section and one Diagram section.

```
[Package]Package name is "$Package.Name"
[Diagram][DiagramObject]DiagramObject (i.e. Element) name is "$DiagramObject.Name" and
DiagramObject type is "$DiagramObject.Type"
[/DiagramObject][/Diagram][/Package]
```

Other repository items

Other repository items like attributes, operations, operation parameters etc. have their corresponding section and variables. For full reference see Template syntax reference for all possible sections in RepoDoc and Enterprise Architect Object Model.

Variables

Variables identify properties of repository items that will be documented (i.e. became part of the document). They are marked with the $ (dollar) sign and correspond (in most of the cases) to the properties (sometimes called **Attributes** in the official Enterprise Architect documentation) of the repository classes that the user wishes to document. Their name is composed of the Class name and Property name delimited with . (dot) i.e. `<ClassName>.<Property>`. Since each section corresponds to a specific class it's only logical that the variable is defined and its value is known only within its section or within the subsection(s) of the section for which it is defined. Long story short, have a look at following examples.

This is a valid template.

```
[Package]List of elements for package: $Package.Name
[Element]$Element.Name
[/Element][/Package]
```

This is also a valid template.

```
[Package]List of elements:
[Element]Element $Element.Name is within package $Package.Name
[/Element][/Package]
```

This template is not valid. Variable `$Element.Name` is misplaced because `$Element.Name` is only defined within the `Element` sections (or its subsection(s)).

```
[Package]List of elements for package: $Package.Name
[Element][/Element]
$Element.Name
[/Package]
```

The properties for each class (or variables for section if you like) may be found in the Enterprise Architect documentation e.g. Element class properties. Have a look at Template syntax reference for a quick overview of all possible variables.

Text

Text is literally everything being not a variable or a section tag.

Advanced topics

Notes format

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

- `$<ClassName>.Notes` returns notes in plain text (without formatting). The colour or font format information is lost, however the lists are preserved using indents and newlines.

- `$<ClassName>.NotesHtml` returns notes in html format with all the formatting expressed using html tags.

Filter attributes

Filter attributes are means to exclude some items (packages, elements, diagrams, etc.) from the final document. By default, RepoDoc puts content of each section that has a corresponding item in the repository into the document. In some cases, it may be useful to exclude some items from the repository and not document them. These may include items with some special stereotype(s) or name(s). The names of filter attributes have the same format as variables and they are specified in the section opening tag, together with a comparison operator and a value. The same restrictions apply for attributes as for variables regarding their placement.

This RepoDoc template puts all elements with type Class into the document.

```
[Package][Element $Element.Type=="Class"]$Element.Name
[/Element][/Package]
```

This RepoDoc template puts only packages with at least one element into the document.

```
[Package $Package.ElementCount!="0"]$Package.Name
[/Package]
```

This RepoDoc template puts only packages with at least one element and one diagram into the document.

```
[Package $Package.ElementCount!="0" $Package.DiagramCount!="0"]$Package.Name
[/Package]
```

If you specify multiple filter attributes they must be separated by a single space. Multiple attributes are always evaluated using AND logical operators i.e. all must be true, otherwise the item will be filtered out.

RepoDoc supports currently four comparison operators:

- `==` acts like a string comparison operator and evaluates to "true", if the specified attribute is equal to the value specified by the user in between quotes,

- `!=` acts like a negated string comparison,

- `=~` acts like a regex matcher and evaluates to "true" if the specified attribute matches the regular expression specified by the user in between quotes,

- `!~` acts like a negated regex matcher.

Using reserved characters

RepoDoc uses some reserved characters that have special meaning during template processing and document generation, namely:

- characters `[` and `]` and `/` mark the opening and closing section tags,

- character `$` (dollar) is used as a variable marker,

- character `"` (double quotes) is used to denote the filter attribute value.

There are some rules to follow if you want to use these characters in your templates and suppress their original meaning.

In ordinary text:

- characters `[`, `]`, and `$` must not stand alone and may be escaped as `[[`, `]]` and `$$`.

- character `"` may stand alone and does not need to be escaped.

In filter attribute values:

- character `$` must not stand alone and may be escaped as `$$`.

- characters `[`, `]` may stand alone and do not need to be escaped.

- character `"` must not be used.

Convenience variables

RepoDoc follows Enterprise Architect class model and therefore many variables are easy to understand for someone being familiar with Enterprise Architect. Beside these variables RepoDoc provides some "convenience" variables, that are not directly defined in the class model. Some of these are described below:

- `$<ClassName>.<AnotherClassName>Count` variables return number of items in child collections. Typically the content of these collections is documented using sections e.g. elements in a package are documented using the `Element` section. Nevertheless, in some cases it may be useful to document the number of items in those collections, e.g. to document the number of attributes for each class or number of methods in a component. For this purpose the item count in collections (see the class model for full extent) are accessible using `$<ClassName>.<AnotherClassName>Count` variables i.e. `$Package.ElementCount`, `$Element.OperationCount` etc.

- `$<ClassName>.IndexOf` returns the index of the item in parent's collection of items of the same type. For example, if the item is an Element, then IndexOf returns either the index of the element in the Elements collection from the parent package or the the index from the collection

of child elements of the parent element if the currently evaluated element is a sub/embedded element.

- $<ClassName>.CountOf returns the number of items in parent's collection of items of the same type. In most of the cases, this value is directly related to the structure of the repository as seen in the Project browser. For example, if the item is an Element, then CountOf returns either the number of elements in the parent package or the number of child elements of the parent element if the currently evaluated element is a sub/embedded element.

- $DG.PackagePath global variable that returns a backslash delimited list of package names starting with the root package and ending with the currently evaluated package.

- $DG.PackageDepth global variable that returns the depth for the currently evaluated package in the tree of packages. The root package has always a depth equal to zero.

Multiple repository iterations

RepoDoc allows you to iterate all items in the repository more than once using a single template. For example, you may wish first to output a list of packages in the repository and then a list of elements for each package. The template syntax allows you to put a section more than once into the template as long as the section relationships are obeyed. The following template makes the described above possible:

```
List of packages:
[Package]
Package name is "$Package.Name"
[/Package]

List of elements for each package:
[Package]
List of elements for package "$Package.Name":
[Element]
$Element.Name
[/Element]
[/Package]
```

Using sections in the manner described above you'll be able to document different parts of repository with a single template. Following template uses filter attributes to output the requirements and then the use cases in two different packages into a single document.

```
List of requirements:
[Package $Package.Name=="Requirements"]
Requirement name is "$Package.Name"
... Put your element or diagram section here ...
[/Package]

List of use cases:
[Package $Package.Name=="Use cases"]
Use case name is "$Package.Name"
... Put your element or diagram section here ...
[/Package]
```

Custom queries

While the sections corresponding to Enterprise Architect classes and their variables should provide all information necessary for the document, there may be cases when information beyond these sections is required. For such cases RepoDoc offers the RDQuery section which offers a Statement attribute for the user's custom SQL query. The following example demonstrates the usage of custom queries.

```
[RDQuery $RDQuery.Statement="SELECT package_id, name FROM t_package"]
[RDQueryRow]
PackageID=$RDQueryRow.Column1
Name=$RDQueryRow.Column2
[/RDQueryRow]
[/RDQuery]
```

The query results are accessible in the RDQueryRow subsection using the Column variables. This subsection is evaluated for each row returned by the query. The example demonstrates just the concept, in this case it would be easier to access the information using simply the Package section.

> ❗ Use the sections corresponding to repository items like Package, Element or Diagram whenever possible. Use RDQuery only in cases when you need some extra information not covered by RepoDoc's sections.

Generator profiles

Generator profile changes the way the repository is processed and/or contains a document post-processing command. It solves easily problems that arise when some repository items contain content that is problematic from the output format point of view. For example a package with name <MyPackage> breaks formatting when your 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 dedicated document generator profile with defined escape sequences. The profile editor lets you choose and edit a document generator profiles.
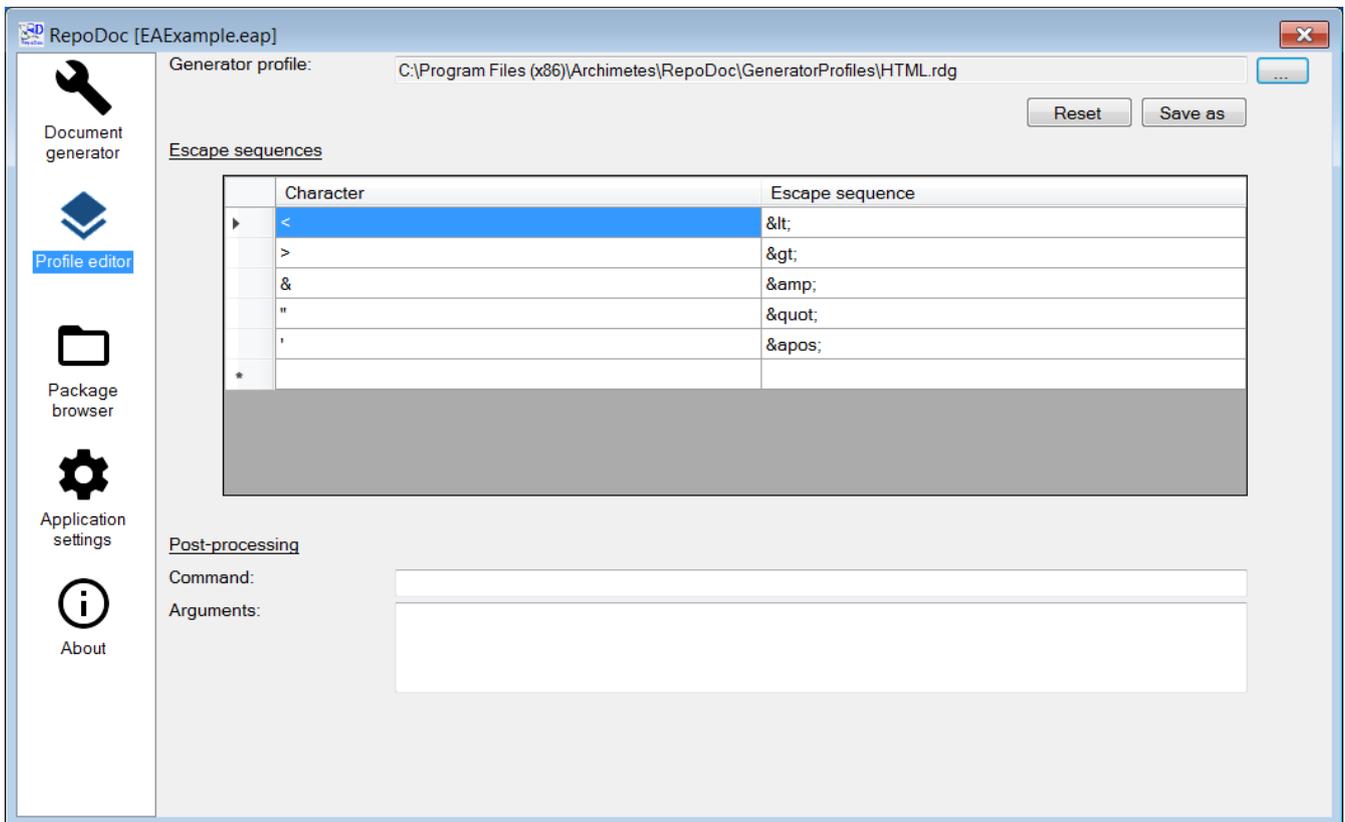
> RepoDoc comes with pre-installed profiles, but you are free to create new profiles to meet your needs. You'll find sample profiles in the installation directory `C:\Program Files (x86)\Archimetes\RepoDoc\GeneratorProfiles`. The generator profiles have `rdg` file extension.

Escape sequences

Each profile may contain a list of characters and a corresponding escape sequence for each character. During document generation RepoDoc uses this list to escape the characters in `Name`, `Alias` and `Notes` variables.

> 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.

> Use the global variables like `$DG.TemplateFileName` or `$DG.DocumentFileName` to simplify your post-processing commands.

Template syntax reference

Global variables

Global variables may be used at any place in the template regardless of the section.

Table 1. Table of global variables

Variable name	Variable description
`$DG.DocumentDirPath`	Document file directory path. If the full document path is `C:\MyDir\MyDocument.html` then the variable evaluates to `C:\MyDir`).
`$DG.DocumentFileName`	Document file name. If the full document path is `C:\MyDir\MyDocument.html` then the variable evaluates to `MyDocument.html`).
`$DG.DocumentFileNameWE`	Document file name without extension. If the full document path is `C:\MyDir\MyDocument.html` then the variable evaluates to `MyDocument`).
`$DG.DocumentFilePath`	Document file path e.g. `C:\MyDir\MyDocument.html`.
`$DG.PackageDepth`	Depth of the currently evaluated package related to the root package. The root package has always depth equal to 0, its child package has depth equal to 1 and so on.
`$DG.PackageGUIDPath`	Backslash delimited list of package GUIDs starting with the root package and ending with the currently evaluated package.
`$DG.PackagePath`	Backslash delimited list of package names starting with the root package and ending with the currently evaluated package.
`$DG.RootPackageGUID`	GUID of the root package.
`$DG.RootPackageName`	Name of the root package.
`$DG.TemplateDirPath`	Template file directory path. If the full template path is `C:\MyDir\MyTemplate.html.rdt` then the variable evaluates to `C:\MyDir`).
`$DG.TemplateFileName`	Template file name. If the full template path is `C:\MyDir\MyTemplate.html.rdt` then the variable evaluates to `MyTemplate.html.rdt`).
`$DG.TemplateFileNameWE`	Template file name without extension. If the full template path is `C:\MyDir\MyTemplate.html.rdt` then the variable evaluates to `MyTemplate.html`).
`$DG.TemplateFilePath`	Template file path e.g. `C:\MyDir\MyTemplate.rdt`.

Sections

Following sections can be used in the template: `Package`, `Element`, `Diagram`, `PresentationDiagram`, `DiagramObject`, `Attribute`, `Operation`, `Parameter`, `Scenario`, `ScenarioStep`, `Constraint`, `Connector`, `ConnectorEnd`, `Tag`, `Model`, `RDQuery`, `RDQueryRow`, `RDPut`, `RDRem`.

Most of the sections are named after the corresponding classes that exist in the Enterprise Architect class model. Notable exceptions are the `RD` prefixed sections, that do not have a corresponding class. Variables represent the properties of repository items that can be documented and are marked with the `$` (dollar) sign. Almost all variables correspond to the properties defined in the Enterprise Architect class model. For these, the descriptions are taken from the official documentation for Enterprise Architect and are provided here for convenience.

Section relationships follow mostly the Enterprise Architect class model i.e. the `Package` section may contain an `Element` or a `Diagram` section and so on. All possible parent-child relationships are pictured below whereby the dark highlighted sections may be used as top level sections.

RepoDoc - Template section relationships
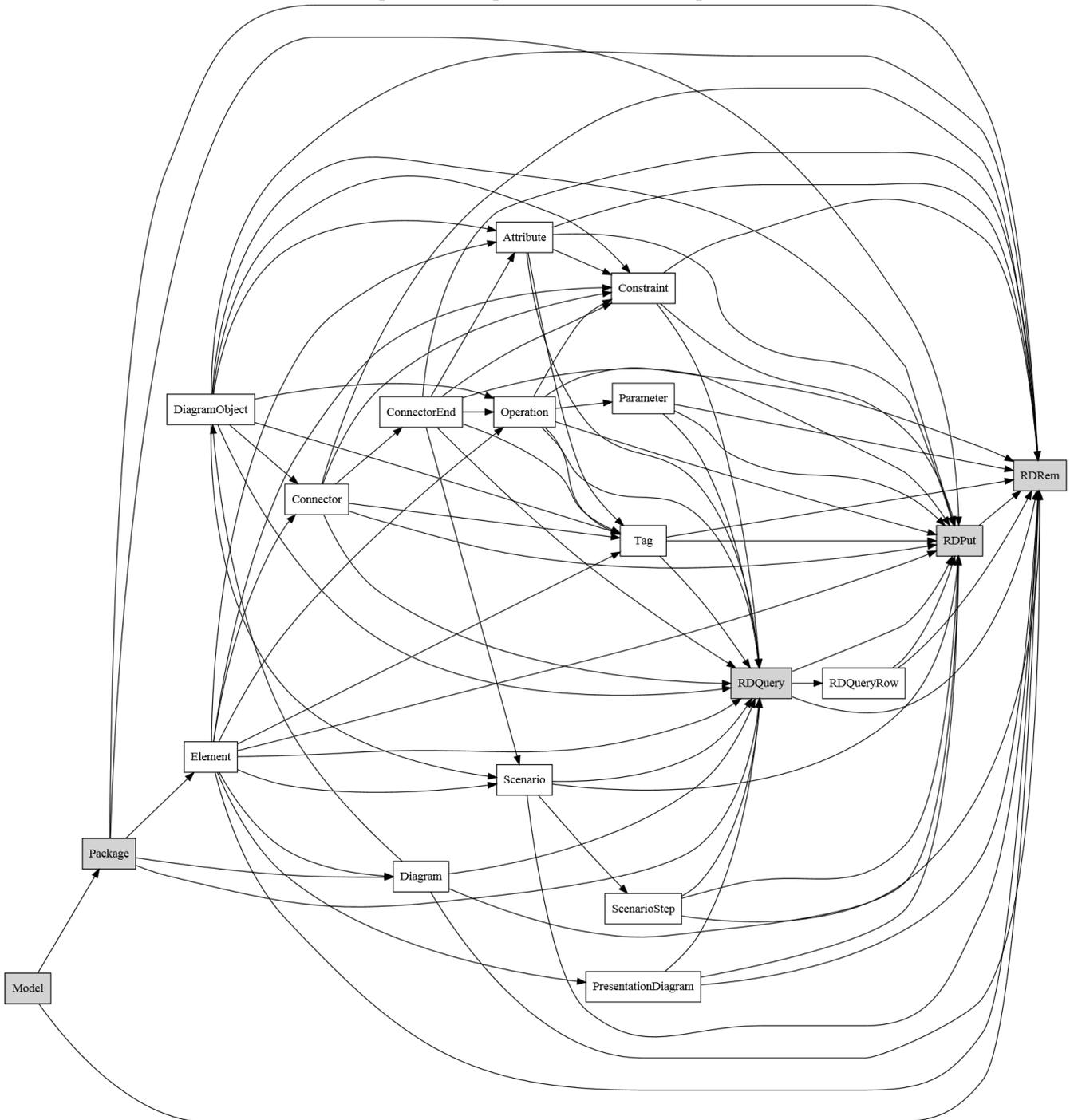
Package

This section is evaluated for each package found in the repository starting with the root package defined by the user. If the package has child packages, they are evaluated before evaluating next package on the same level.

Table 2. Table of variables for section Package

Variable name	Variable description
$Package.Alias	Package alias.
$Package.CountOf	Number of items in parent's collection of items of the same type.

Variable name	Variable description
`$Package.CreatedDate`	The date the package was created.
`$Package.DiagramCount`	Number of child diagrams of the package.
`$Package.ElementCount`	Number of child elements of the package.
`$Package.ElementID`	ID of the package element (evaluates to empty string if the package represents a model).
`$Package.GUID`	Package GUID.
`$Package.ID`	Package ID.
`$Package.IndexOf`	Index of the item in parent's collection of items of the same type.
`$Package.IsControlled`	Indicates if the package has been marked as Controlled.
`$Package.IsModel`	Indicates if the package is a model.
`$Package.IsNamespace`	Indicates that the package is a Namespace root.
`$Package.IsProtected`	Indicates if the package has been marked as Protected.
`$Package.ModifiedDate`	The date the package was modified.
`$Package.Name`	The name of the package.
`$Package.Notes`	Notes in plain text format (i.e. without formatting).
`$Package.NotesHtml`	Notes in HTML format.
`$Package.Owner`	The package owner when using controlled packages.
`$Package.PackageCount`	Number of child packages of the package.
`$Package.ParentID`	The ID of the parent package. Zero indicates that the evaluated package is a model and has no parent.
`$Package.Version`	The version of the package.

Element

This section is evaluated for each element associated with or contained in the package. Any child elements of an element are evaluated before evaluating next element on the same level.

Table 3. Table of variables for section Element

Variable name	Variable description
`$Element.Abstract`	Indicates if the element is Abstract (1) or Concrete (0).

Variable name	Variable description
$Element.ActionFlags	A structure to hold flags concerned with Action semantics.
$Element.Alias	An alias for the element.
$Element.AttributeCount	Number of attributes defined for the element.
$Element.Author	The element author.
$Element.ClassifierID	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).
$Element.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.
$Element.ConnectorCount	Number of connectors defined for the element.
$Element.ConstraintCount	Number of constraints defined for the element.
$Element.CountOf	Number of items in parent's collection of items of the same type.
$Element.CreatedDate	The date the element was created.
$Element.DiagramCount	Number of child diagrams of the element.
$Element.ElementCount	Number of child elements of the element.
$Element.GUID	Element GUID.
$Element.ID	Element ID.
$Element.IndexOf	Index of the item in parent's collection of items of the same type.
$Element.IsActive	Boolean value indicating whether the element is active or not.
$Element.IsLeaf	Boolean value indicating whether the element is in leaf node or not.
$Element.IsRoot	
$Element.IsSpec	Boolean value indicating whether the element is a specification or not.
$Element.Language	The code generation type; for example, Java, C++, C#, VBNet, Visual Basic, Delphi.
$Element.ModifiedDate	The date the element was modified.
$Element.Multiplicity	Multiplicity value for the element.
$Element.Name	The name of the element.

Variable name	Variable description
$Element.Notes	Notes in plain text format (i.e. without formatting).
$Element.NotesHtml	Notes in HTML format.
$Element.OperationCount	Number of operations defined for the element.
$Element.PackageID	ID of the package containing the element.
$Element.ParentID	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).
$Element.Persistence	The persistence associated with this element; can be Persistent or Transient.
$Element.Phase	The phase the element is scheduled to be constructed in; any string value.
$Element.PresentationDiagramCount	Number of presentation diagrams (diagrams displaying the element) of the element.
$Element.RunState	The element's runstate list as a string.
$Element.ScenarioCount	Number of scenarios defined for the element.
$Element.Status	The status of the element, such as Proposed or Approved.
$Element.Stereotype	The primary stereotype of the element.
$Element.TagCount	Number of tags defined for the element.
$Element.Type	The element type (such as Class, Component).
$Element.Version	The version of the element.
$Element.Visibility	The Scope of the element within the current Package. Valid values are: Public, Private, Protected or Package.

Diagram

Depending on the parent section, this section is evaluated for each diagram contained in the evaluated package or element.

Table 4. Table of variables for section Diagram

Variable name	Variable description
$Diagram.CountOf	Number of items in parent's collection of items of the same type.
$Diagram.DiagramObjectCount	Number of diagram objects displayed on the diagram.

Variable name	Variable description
`$Diagram.GUID`	Diagram GUID.
`$Diagram.ID`	Diagram ID.
`$Diagram.ImagePng`	A base64 encoded diagram image in PNG format.
`$Diagram.IndexOf`	Index of the item in parent's collection of items of the same type.
`$Diagram.Name`	The name of the diagram.
`$Diagram.Notes`	Notes in plain text format (i.e. without formatting).
`$Diagram.NotesHtml`	Notes in HTML format.
`$Diagram.PackageID`	The ID of the Package that the diagram belongs to.
`$Diagram.ParentID`	ID of the element the diagram is child of. Contains 0 if the diagram is child of the package.
`$Diagram.Type`	The diagram type for example Activity or Logical.

PresentationDiagram

This section is evaluated for each diagram that displays the currently evaluated element. It offers the same variables as the `Diagram` section.

Table 5. Table of variables for section PresentationDiagram

Variable name	Variable description
`$PresentationDiagram.CountOf`	Number of items in parent's collection of items of the same type.
`$PresentationDiagram.DiagramObjectCount`	Number of diagram objects displayed on the diagram.
`$PresentationDiagram.GUID`	Diagram GUID.
`$PresentationDiagram.ID`	Diagram ID.
`$PresentationDiagram.ImagePng`	A base64 encoded diagram image in PNG format.
`$PresentationDiagram.IndexOf`	Index of the item in parent's collection of items of the same type.
`$PresentationDiagram.Name`	The name of the diagram.
`$PresentationDiagram.Notes`	Notes in plain text format (i.e. without formatting).
`$PresentationDiagram.NotesHtml`	Notes in HTML format.

Variable name	Variable description
`$PresentationDiagram.PackageID`	The ID of the Package that the diagram belongs to.
`$PresentationDiagram.ParentID`	ID of the element the diagram is child of. Contains 0 if the diagram is child of the package.
`$PresentationDiagram.Type`	The diagram type for example Activity or Logical.

DiagramObject

This section is evaluated for each diagram object displayed on the currently evaluated diagram. It offers the same variables as the `Element` section plus the coordinates related variables.

Table 6. Table of variables for section DiagramObject

Variable name	Variable description
`$DiagramObject.Abstract`	Indicates if the element is Abstract (1) or Concrete (0).
`$DiagramObject.ActionFlags`	A structure to hold flags concerned with Action semantics.
`$DiagramObject.Alias`	An alias for the element.
`$DiagramObject.AttributeCount`	Number of attributes defined for the element.
`$DiagramObject.Author`	The element author.
`$DiagramObject.Bottom`	The bottom edge position of the diagram object in the image.
`$DiagramObject.ClassifierID`	The ElementID of a Classifier associated with the element; that is, the base type. Only valid for instance type elements (such as Object or Sequence).
`$DiagramObject.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.
`$DiagramObject.ConnectorCount`	Number of connectors defined for the element.
`$DiagramObject.ConstraintCount`	Number of constraints defined for the element.
`$DiagramObject.CountOf`	Number of items in parent's collection of items of the same type.
`$DiagramObject.CreatedDate`	The date the element was created.
`$DiagramObject.DiagramCount`	Number of child diagrams of the element.
`$DiagramObject.DiagramID`	The ID of the associated diagram where the element is displayed.

Variable name	Variable description
`$DiagramObject.ElementCount`	Number of child elements of the element.
`$DiagramObject.GUID`	Element GUID.
`$DiagramObject.ID`	Element ID.
`$DiagramObject.IndexOf`	Index of the item in parent's collection of items of the same type.
`$DiagramObject.IsActive`	Boolean value indicating whether the element is active or not.
`$DiagramObject.IsLeaf`	Boolean value indicating whether the element is in leaf node or not.
`$DiagramObject.IsRoot`	
`$DiagramObject.IsSpec`	Boolean value indicating whether the element is a specification or not.
`$DiagramObject.Language`	The code generation type; for example, Java, C++, C#, VBNet, Visual Basic, Delphi.
`$DiagramObject.Left`	The left edge position of the diagram object in the image.
`$DiagramObject.ModifiedDate`	The date the element was modified.
`$DiagramObject.Multiplicity`	Multiplicity value for the element.
`$DiagramObject.Name`	The name of the element.
`$DiagramObject.Notes`	Notes in plain text format (i.e. without formatting).
`$DiagramObject.NotesHtml`	Notes in HTML format.
`$DiagramObject.OperationCount`	Number of operations defined for the element.
`$DiagramObject.PackageID`	ID of the package containing the element.
`$DiagramObject.ParentID`	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).
`$DiagramObject.Persistence`	The persistence associated with this element; can be Persistent or Transient.
`$DiagramObject.Phase`	The phase the element is scheduled to be constructed in; any string value.
`$DiagramObject.PresentationDiagramCount`	Number of presentation diagrams (diagrams displaying the element) of the element.
`$DiagramObject.Right`	The right edge position of the diagram object in the image.
`$DiagramObject.RunState`	The element's runstate list as a string.

Variable name	Variable description
`$DiagramObject.ScenarioCount`	Number of scenarios defined for the element.
`$DiagramObject.Status`	The status of the element, such as Proposed or Approved.
`$DiagramObject.Stereotype`	The primary stereotype of the element.
`$DiagramObject.TagCount`	Number of tags defined for the element.
`$DiagramObject.Top`	The top edge position of the diagram object in the image.
`$DiagramObject.Type`	The element type (such as Class, Component).
`$DiagramObject.Version`	The version of the element.
`$DiagramObject.Visibility`	The Scope of the element within the current Package. Valid values are: Public, Private, Protected or Package.

Attribute

Depending on the parent section, this section is evaluated for each element, diagram object or connector end attribute.

Table 7. Table of variables for section Attribute

Variable name	Variable description
`$Attribute.AllowDuplicates`	Indicates if duplicates are allowed in the collection. If the attribute represents a database column this, when set, represents the Not Null option.
`$Attribute.ConstraintCount`	Number of constraints defined for the attribute.
`$Attribute.Container`	The container type.
`$Attribute.Containment`	The type of containment - Not Specified, By Reference or By Value.
`$Attribute.CountOf`	Number of items in parent's collection of items of the same type.
`$Attribute.Default`	The initial value assigned to this attribute.
`$Attribute.GUID`	Attribute GUID.
`$Attribute.ID`	Attribute ID.
`$Attribute.IndexOf`	Index of the item in parent's collection of items of the same type.
`$Attribute.IsCollection`	Indicates if the current feature is a collection or not. If the attribute represents a database column this, when set, represents a Foreign Key.

Variable name	Variable description
`$Attribute.IsConst`	A flag indicating if the attribute is Const or not.
`$Attribute.IsDerived`	Indicates if the attribute is derived (that is, a calculated value).
`$Attribute.IsOrdered`	Indicates if a collection is ordered or not. If the attribute represents a database column this, when set, represents a Primary Key.
`$Attribute.IsStatic`	Indicates if the attribute is a static feature or not. If the attribute represents a database column this, when set, represents the Unique option.
`$Attribute.Length`	The attribute length, where applicable.
`$Attribute.LowerBound`	A value for the collection lower boundary.
`$Attribute.Name`	The name of the attribute.
`$Attribute.Notes`	Notes in plain text format (i.e. without formatting).
`$Attribute.NotesHtml`	Notes in HTML format.
`$Attribute.ParentID`	Returns the ElementID of the element that this attribute is a part of.
`$Attribute.Precision`	The precision value.
`$Attribute.Scale`	The scale value.
`$Attribute.Stereotype`	The stereotype for the attribute.
`$Attribute.TagCount`	Number of tags defined for the attribute.
`$Attribute.Type`	The attribute type (by name; also see ClassifierID).
`$Attribute.UpperBound`	A value for the collection upper boundary.
`$Attribute.Visibility`	Identifies the scope of the attribute - Private, Protected, Public or Package.

Operation

Depending on the parent section, this section is evaluated for each element, diagram object or connector end operation.

Table 8. Table of variables for section Operation

Variable name	Variable description
`$Operation.Abstract`	A flag indicating if the operation is abstract (1) or not (0).
`$Operation.Behavior`	Some further explanatory behavior notes (for example, pseudocode).

Variable name	Variable description
$Operation.ClassifierID	The Classifier ID that applies to the ReturnType.
$Operation.Code	An optional field to hold the operation code (used for the Initial Code field).
$Operation.Concurrency	Indicates the concurrency type of the method.
$Operation.ConstraintCount	Number of constraints defined for the operation.
$Operation.CountOf	Number of items in parent's collection of items of the same type.
$Operation.GUID	Operation GUID.
$Operation.ID	Operation ID.
$Operation.IndexOf	Index of the item in parent's collection of items of the same type.
$Operation.IsConst	A flag indicating that the operation is Const.
$Operation.IsLeaf	A flag to indicate if the operation is Leaf (cannot be overridden).
$Operation.IsPure	A flag indicating that the operation is defined as Pure in C++.
$Operation.IsQuery	A flag to indicate if the operation is a query (that is, does not alter Class variables).
$Operation.IsRoot	A flag to indicate if the operation is Root.
$Operation.IsStatic	A flag to indicate a static operation.
$Operation.IsSynchronized	A flag indicating a Synchronized operation call.
$Operation.Name	The name of the operation.
$Operation.Notes	Notes in plain text format (i.e. without formatting).
$Operation.NotesHtml	Notes in HTML format.
$Operation.ParameterCount	Number of parameters defined for the operation.
$Operation.ParentID	Returns the ElementID of the element that the operation belongs to.
$Operation.ReturnIsArray	A flag to indicate that the return value is an array.
$Operation.ReturnType	The return type for the operation; this can be a primitive data type or a Class or Interface type.
$Operation.StateFlags	Some flags as applied to operations in State elements.
$Operation.Stereotype	The operation stereotype.

Variable name	Variable description
$Operation.TagCount	Number of tags defined for the operation.
$Operation.Visibility	The operation scope - Public, Protected, Private or Package.

Parameter

This section is evaluated for each operation parameter.

Table 9. Table of variables for section Parameter

Variable name	Variable description
$Parameter.AllowDuplicates	Indicates if duplicates are allowed in the collection.
$Parameter.ClassifierID	A ClassifierID for the parameter, if known.
$Parameter.CountOf	Number of items in parent's collection of items of the same type.
$Parameter.Default	A default value for this parameter.
$Parameter.GUID	Parameter GUID.
$Parameter.IndexOf	Index of the item in parent's collection of items of the same type.
$Parameter.IsConst	A flag indicating that the parameter is Const (cannot be altered).
$Parameter.IsOrdered	Indicates if a collection is ordered or not.
$Parameter.Kind	The parameter kind - in, inout, out, or return.
$Parameter.LowerBound	A value for the collection lower boundary.
$Parameter.Name	The name of the parameter.
$Parameter.Notes	Notes in plain text format (i.e. without formatting).
$Parameter.NotesHtml	Notes in HTML format.
$Parameter.OperationID	Operation ID of the operation that this parameter belongs to.
$Parameter.Type	The parameter type; can be a primitive type or a defined classifier.
$Parameter.UpperBound	A value for the collection upper boundary.

Scenario

Depending on the parent section, this section is evaluated for each element, diagram object or connector end scenario.

Table 10. Table of variables for section Scenario

Variable name	Variable description
`$Scenario.BasicPathBranchLevel`	Branch level in the basic path scenario if this scenario is an alternative/exception scenario. Returns empty string if this scenario is a basic path scenario.
`$Scenario.BasicPathMergeLevel`	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.
`$Scenario.BasicPathName`	Name of the basic path scenario if this scenario is an alternative/exception scenario. Returns the name of itself if this is scenario is a basic path scenario.
`$Scenario.CountOf`	Number of items in parent's collection of items of the same type.
`$Scenario.GUID`	Scenario GUID.
`$Scenario.IndexOf`	Index of the item in parent's collection of items of the same type.
`$Scenario.Name`	Scenario name.
`$Scenario.Notes`	Notes in plain text format (i.e. without formatting).
`$Scenario.NotesHtml`	Notes in HTML format.
`$Scenario.ParentID`	Element ID of the element this scenario belongs to.
`$Scenario.StepCount`	Number of steps defined for the scenario.
`$Scenario.Type`	The scenario type (for example, Basic Path).

ScenarioStep

This section is evaluated for each scenario step defined in the evaluated scenario.

Table 11. Table of variables for section ScenarioStep

Variable name	Variable description
`$ScenarioStep.Action`	Identifies the action specified for the step.
`$ScenarioStep.CountOf`	Number of items in parent's collection of items of the same type.
`$ScenarioStep.GUID`	Scenario step GUID.

Variable name	Variable description
$ScenarioStep.IndexOf	Index of the item in parent's collection of items of the same type.
$ScenarioStep.Level	The number of the step as shown in the scenario editor.
$ScenarioStep.Results	Any results that are given from the step.
$ScenarioStep.ScenarioGUID	GUID of the scenario that the step is a part of.
$ScenarioStep.State	A description of the state the system enters when the step is executed.
$ScenarioStep.Type	Identifies whether the step is being performed by a user or the system.
$ScenarioStep.Uses	The input and requirements that are relevant to the step.

Constraint

This section is evaluated for each constraint defined in the evaluated repository item.

Table 12. Table of variables for section Constraint

Variable name	Variable description
$Constraint.AdditionalInfo	Contains Status in case of Element constraint and Type in case of Operation constraint. Is empty for Attribute constraints.
$Constraint.CountOf	Number of items in parent's collection of items of the same type.
$Constraint.IndexOf	Index of the item in parent's collection of items of the same type.
$Constraint.Name	The constraint name.
$Constraint.Notes	Notes in plain text format (i.e. without formatting).
$Constraint.NotesHtml	Notes in HTML format.
$Constraint.ParentID	ID of the parent this constraint belongs to.
$Constraint.Type	Constraint type.

Connector

Depending on the parent section, this section is evaluated for each element or diagram object connector. The section contains also variables for the source and destination connector ends.

Table 13. Table of variables for section Connector

Variable name	Variable description
`$Connector.Alias`	Connector alias.
`$Connector.ConstraintCount`	Number of constraints defined for the connector.
`$Connector.CountOf`	Number of items in parent's collection of items of the same type.
`$Connector.DestID`	The ElementID of the element at the target end of this connector.
`$Connector.Direction`	The connector direction, which can be set to one of the following: Unspecified, Bi-Directional, Source → Destination, Destination → Source.
`$Connector.GUID`	Connector GUID.
`$Connector.ID`	Connector ID.
`$Connector.IndexOf`	Index of the item in parent's collection of items of the same type.
`$Connector.IsLeaf`	A flag indicating that the connector is a leaf.
`$Connector.IsRoot`	A flag indicating that the connector is a root.
`$Connector.IsSpecification`	A flag indicating that the connector is a specification.
`$Connector.Name`	The name of the connector.
`$Connector.Notes`	Notes in plain text format (i.e. without formatting).
`$Connector.NotesHtml`	Notes in HTML format.
`$Connector.SourceID`	The ElementID of the element at the source end of this connector.
`$Connector.Stereotype`	The stereotype for the connector.
`$Connector.TagCount`	Number of tags defined for the connector.
`$Connector.Type`	The connector type; valid types are held in the t_connectortypes table.

ConnectorEnd

This section is evaluated for each connector end belonging to the currently evaluated connector. It offers the same variables as the `Element` section plus the connector end related variables.

Table 14. Table of variables for section ConnectorEnd

Variable name	Variable description
`$ConnectorEnd.Abstract`	Indicates if the element is Abstract (1) or Concrete (0).

Variable name	Variable description
$ConnectorEnd.ActionFlags	A structure to hold flags concerned with Action semantics.
$ConnectorEnd.Aggregation	The type of Aggregation as it applies to the connector end; valid values are: 0 = None, 1 = Shared, 2 = Composite.
$ConnectorEnd.Alias	An alias for the element.
$ConnectorEnd.AllowDuplicates	For multiplicities greater than 1, indicates that duplicate entries are possible.
$ConnectorEnd.AttributeCount	Number of attributes defined for the element.
$ConnectorEnd.Author	The element author.
$ConnectorEnd.Cardinality	The cardinality associated with the particular connector end.
$ConnectorEnd.ClassifierID	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).
$ConnectorEnd.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.
$ConnectorEnd.ConnectorCount	Number of connectors defined for the element.
$ConnectorEnd.Constraint	A constraint that can be applied to the particular connector end.
$ConnectorEnd.ConstraintCount	Number of constraints defined for the element.
$ConnectorEnd.Containment	The containment type applied to the particular connector end.
$ConnectorEnd.CountOf	Number of items in parent's collection of items of the same type.
$ConnectorEnd.CreatedDate	The date the element was created.
$ConnectorEnd.Derived	Indicates that the value of the particular connector end is derived.
$ConnectorEnd.DerivedUnion	Indicates the value of the role derived from the union of all roles that subset this.
$ConnectorEnd.DiagramCount	Number of child diagrams of the element.
$ConnectorEnd.ElementCount	Number of child elements of the element.
$ConnectorEnd.EndStereotype	Gets the stereotype for the particular connector end.

Variable name	Variable description
$ConnectorEnd.EndVisibility	The Scope associated with the particular connector end - Public, Private, Protected or Package.
$ConnectorEnd.GUID	Element GUID.
$ConnectorEnd.ID	Element ID.
$ConnectorEnd.IndexOf	Index of the item in parent's collection of items of the same type.
$ConnectorEnd.IsActive	Boolean value indicating whether the element is active or not.
$ConnectorEnd.IsChangeable	Flag indicating whether the particular end is changeable or not - frozen, addOnly or none.
$ConnectorEnd.IsLeaf	Boolean value indicating whether the element is in leaf node or not.
$ConnectorEnd.IsNavigable	A flag indicating that the particular connector end is navigable from the other end.
$ConnectorEnd.IsRoot	
$ConnectorEnd.IsSpec	Boolean value indicating whether the element is a specification or not.
$ConnectorEnd.Language	The code generation type; for example, Java, C++, C#, VBNet, Visual Basic, Delphi.
$ConnectorEnd.ModifiedDate	The date the element was modified.
$ConnectorEnd.Multiplicity	Multiplicity value for the element.
$ConnectorEnd.Name	The name of the element.
$ConnectorEnd.Navigability	Indicates whether the role of an association is navigable from the opposite classifier - Navigable, Non-Navigable or Unspecified.
$ConnectorEnd.Notes	Notes in plain text format (i.e. without formatting).
$ConnectorEnd.NotesHtml	Notes in HTML format.
$ConnectorEnd.OperationCount	Number of operations defined for the element.
$ConnectorEnd.Ordering	Ordering for the particular connector end.
$ConnectorEnd.Owned	Indicates that the Association end corresponds to an attribute on the opposite end of the Association.
$ConnectorEnd.PackageID	ID of the package containing the element.

Variable name	Variable description
$ConnectorEnd.ParentID	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).
$ConnectorEnd.Persistence	The persistence associated with this element; can be Persistent or Transient.
$ConnectorEnd.Phase	The phase the element is scheduled to be constructed in; any string value.
$ConnectorEnd.PresentationDiagramCount	Number of presentation diagrams (diagrams displaying the element) of the element.
$ConnectorEnd.Qualifier	A qualifier that can apply to the particular connector end.
$ConnectorEnd.Role	The particular connector end role.
$ConnectorEnd.RoleNote	Notes associated with the role of the particular connector end.
$ConnectorEnd.RoleType	The role type applied to the particular end of the connector.
$ConnectorEnd.RunState	The element's runstate list as a string.
$ConnectorEnd.ScenarioCount	Number of scenarios defined for the element.
$ConnectorEnd.Status	The status of the element, such as Proposed or Approved.
$ConnectorEnd.Stereotype	The primary stereotype of the element.
$ConnectorEnd.TagCount	Number of tags defined for the element.
$ConnectorEnd.Type	The element type (such as Class, Component).
$ConnectorEnd.Version	The version of the element.
$ConnectorEnd.Visibility	The Scope of the element within the current Package. Valid values are: Public, Private, Protected or Package.

Tag

This section is evaluated for each element, diagram object, attribute, operation, connector or connector end tag. Note that for diagram objects and connector ends, the tagged values of the underlying element are evaluated.

Table 15. Table of variables for section Tag

Variable name	Variable description
$Tag.CountOf	Number of items in parent's collection of items of the same type.

Variable name	Variable description
`$Tag.GUID`	Tag GUID.
`$Tag.IndexOf`	Index of the item in parent's collection of items of the same type.
`$Tag.Name`	The name of the tag.
`$Tag.Notes`	Notes in plain text format (i.e. without formatting).
`$Tag.NotesHtml`	Notes in HTML format.
`$Tag.ParentID`	ID of the parent this tag belongs to.
`$Tag.Value`	The value assigned to this tag.

Model

This section causes the iteration in the `Package` section to start at the model level. Using this section, the user can jump out of the scope of the root package and start gathering information from the model in which the root package exists.

RDQuery

This section represents a custom query from the user. Use this section to query any data from the repository using an SQL command.

Table 16. Table of variables for section RDQuery

Variable name	Variable description
`$RDQuery.ColumnCount`	Number of columns in the returned data.
`$RDQuery.RowCount`	Number of rows returned by the query.
`$RDQuery.Statement`	SQL SELECT statement.

RDQueryRow

This section is evaluated for each row in the returned custom query.

Table 17. Table of variables for section RDQueryRow

Variable name	Variable description
`$RDQueryRow.Column1`	Value of the 1st column in the returned row.
`$RDQueryRow.Column10`	Value of the 10th column in the returned row.
`$RDQueryRow.Column11`	Value of the 11th column in the returned row.
`$RDQueryRow.Column12`	Value of the 12th column in the returned row.
`$RDQueryRow.Column13`	Value of the 13th column in the returned row.

Variable name	Variable description
`$RDQueryRow.Column14`	Value of the 14th column in the returned row.
`$RDQueryRow.Column15`	Value of the 15th column in the returned row.
`$RDQueryRow.Column16`	Value of the 16th column in the returned row.
`$RDQueryRow.Column17`	Value of the 17th column in the returned row.
`$RDQueryRow.Column18`	Value of the 18th column in the returned row.
`$RDQueryRow.Column19`	Value of the 19th column in the returned row.
`$RDQueryRow.Column2`	Value of the 2nd column in the returned row.
`$RDQueryRow.Column20`	Value of the 20th column in the returned row.
`$RDQueryRow.Column3`	Value of the 3rd column in the returned row.
`$RDQueryRow.Column4`	Value of the 4th column in the returned row.
`$RDQueryRow.Column5`	Value of the 5th column in the returned row.
`$RDQueryRow.Column6`	Value of the 6th column in the returned row.
`$RDQueryRow.Column7`	Value of the 7th column in the returned row.
`$RDQueryRow.Column8`	Value of the 8th column in the returned row.
`$RDQueryRow.Column9`	Value of the 9th column in the returned row.

RDPut

This section has no corresponding class and it's content is evaluated as is. Use this section with some filter attributes within another section to achieve a finer control over the output.

RDRem

This section serves as remark and its content is not evaluated.