Using IBM Rational Publishing Engine to generate compliance documents, Part 4: Generate diagram-based documents from IBM Rational Software Architect Design Manager

Dr. Einar W. Karlsen

March 11, 2014

Part of a series covering tips, tricks, and best practices for using IBM Rational Publishing Engine to generate compliance-relevant reports, this article shows how to generate IBM Rational Software Architect Design Manager design documents using a diagram-based reporting approach.

More articles on document generation

For other articles on this topic, please see the four-part series by Einar Karlsen on Generating compliance documentation using Rational Publishing Engine. Topics include: overview and workflow (part 1); document design and layout (part 2); reporting over multiple data sources (part 3); and testing, review, and deployment (part 4).

This article shows how to use IBM® Rational® Publishing Engine to generate design documents with IBM® Rational® Software Architect Design Manager using a diagram-based reporting approach.

Model-based reporting was discussed in Part 3 of this series (see sidebar, "Model-based reporting"). In diagram-based reporting, reporting doesn't take its origin in a complete model as such. Rather, it starts by extracting diagrams and then retrieves information about the elements on the diagram, independent of where these elements are declared — that is, whether they're in the same model as the diagram or in other models. One example of diagram-based reporting is shown in Figure 1:
Diagram-based reporting or model-based reporting

The diagram-based style of reporting is convenient when you use it to generate reports for Enterprise Architecture Frameworks such as The Open Group Architecture Framework (TOGAF) or the NATO Architecture Framework (NAF) because these frameworks are view-oriented. To comply with NAF, for example, you must create reports containing different views of the application, which can be defined using UML diagrams. This diagram-based style of reporting provides more flexibility than a traditional model-based reporting approach: in fact, the internal structure of a diagram-based model is not (necessarily) obvious from the final report, whereas it would be completely visible in a model-based approach.

So the requirements to model guidelines for diagram-based reporting are less rigorous than for model-based reporting, but this doesn't mean that model guidelines can (or should) be ignored. They're still fundamental to produce high-quality models when you're working in a team. For example, you still should have guidelines that define the rules for naming diagrams and UML elements, or for the font size and style used in UML element documentations, and so on, so you can generate a report that looks proficient.
A case study

Model-based reporting

In model-based reporting, a model is traversed recursively to some depth, and diagrams as well as model elements found in the packages form the basis for generating the report to some defined level of detail.

Such a reporting style usually requires well-defined model guidelines, because the model's structure is directly reflected in the document. The table of content mirrors the package structure, package names are used as a basis for defining section titles, and details are extracted based on the information available for the UML elements located in the package.

Some packages in the model may convey information that clients would typically not like exposing in the report. Users of this reporting style therefore typically invent tricks such as preceding internal package names with an underscore or using attributes to define whether a package should be visible for reporting.

Diagram-based reporting imposes additional challenges and techniques. Requiring that a report extract one diagram is impractical in the context of large models. Diagram-based reporting therefore demands an approach where reporting can be controlled in a flexible way, using parameters so that a document can be split into a limited number of subdocuments, which are finally collected (using the concept of Microsoft Word master and affiliated documents, for example).

In the remainder of this article I'll introduce the solution for diagram-based reporting using a real case study with the following reporting requirements:

- Report generation for a specific subdocument can start at any package in a model (or the model itself).
- Defining whether all diagrams in the package should be extracted or only a diagram with a specific name must be possible by using parameters.
- Reporting recursively on the diagrams of any child package to a specific depth of recursion should be possible.
- Reporting should be able to show element information in more or less detail (depending on user configuration).
- Limiting the UML elements for which information is shown according to their type should be possible.
- The initial section number for the topmost package must be flexible — it can be a chapter, a subsection, a sub-subsection, and so on.

For diagram-based reporting to work with these requirements, you must define the parameters in the corresponding template, which can be configured when you define the Rational Publishing Engine document specification.

Overall design

Fulfilling these requirements might be simpler if you split the solution into two levels rather than define one large, complex template to do everything:
• **Main template** (also called a **crawler**) that traverses a model, processes packages and diagrams, and controls the structure of the report. It's the crawler that decides which information gets into the report and where.
• **UML type-specific snippets** that are externally referenced and render the details of the UML element types involved in reporting. There will be at least one UML snippet for each relevant UML type for reporting.

Figure 2 shows the overall design. The approach makes extensive use of parameters to control what is generated into the report. To get this approach to work, you must be able to pass parameters from the crawler to the snippets — for example, the URL of the UML element to render.

**Figure 2. Design of the solution**

You can define various kinds of crawlers, each of which extracts information from a model differently. However, different crawlers may share the same set of snippets. Changes in a snippet immediately propagate to all crawlers that use that snippet, which makes the approach flexible and more easily adoptable to changes in the requirements.

The result, when generated, is a Microsoft Word document. Word documents from different generations can then be combined into a larger document using the concept of Word master and affiliated document. This way you can also combine generated output from the Rational Publishing Engine with handwritten information such as title page, change notes, table of content, introductions, conclusions, and so on.

**Parameterization of document specification**

A document specification for diagram-based reporting could typically take the form shown in Figure 3. Notice that the data source is scoped to a UML model.
For diagram-based reporting to work, parameters must be provided that uniquely identify the top-level package in scope for reporting. This identification is done with the variables `PackageName`, which denotes the simple name of the top-level package, and `PackageId`, which is the unique package ID as it can be found in Rational Design Manager. For example, `PackageId` could be used if the package names are not unique within a model. The variable `DiagramName` can be used to narrow the scope to a particular diagram in the package.

`RecursionLevel` defines the depth of recursion starting from the top-level package. Its default is 0, meaning that no recursion takes place. The variable `SectionLevel` defines the section style for the top-level package (H1, H2, H3, and so on) in terms of an integer number. The variable `UMLTypes` defines the types of elements for which information should be extracted in terms of a comma- or semicolon-separated list. Its default value is “All,” meaning that information is extracted for all objects found on a diagram. Finally, the variable `RenderElementDetails` controls the UML elements on the diagram if just the name, stereotype, and description are supposed to be rendered (“false”) or if more information (such as class operations, attributes, and so on) should be reported, as well (“true”).

**Diagram elements**

One key requirement in diagram-based reporting is that the report must extract information about the elements on a diagram in scope for reporting. Technically this boils down to iterating over all
object references found on a diagram. The key template fragment for this process is shown in Figure 4:

**Figure 4. Extracting diagram elements**

For each diagram found (top container, query $37$), the template prints the diagram name, description, and image, followed by a figure title. The Rational Design Manager schema for diagrams offers a query (bottom container, query $40$) named references/Object to extract the objects referenced on the diagram. This query retrieves all object references, such as URLs of the objects found on the diagram.

For a use case diagram, extracting information about the use cases and actors on the diagram makes sense, whereas, for example, associations may be less relevant. The iteration has therefore been supplemented by a condition that checks if the concrete types of the current object references are in the set of UML types listed in the external variable UMLTypes. The condition is also true if UMLTypes has been set to All. After the object reference is retrieved, you can retrieve the detailed information, which depends on the type of the element at hand.

**Crawlers and snippets**

One limitation of diagram-based reporting in Rational Design Manager is that the information about the object references found on the diagram is limited to the URL and the concrete type of the object. Model-based reporting provides a bit more — for example, basic information such as the name, description, and applied stereotypes, which is sufficient to render overview information about an element.

For diagram-based reporting, you must retrieve this information using a dynamic data source connection on an element-by-element basis. Depending on the concrete type of UML object,
different kinds of data sources and queries must be used. Therefore, you might want to try an approach that defines UML type-specific snippets that are externally referenced by the crawler and render the details of the UML element types involved in reporting. Figure 5 shows the externally referenced templates in green; these are basically templates in their own right. The advantages of using this approach are that changes to the template are immediately visible in all referencing crawlers, which makes maintenance easier and adoption to changing requirements more flexible.

**Figure 5. Dispatcher for UML element types**

Figure 5 also shows a central part of the main template. This template investigates the concrete type of the UML element and dispatches the control to the template snippet responsible for generating documentation for the element type. The URL of the object is initially assigned to the internal variable \_ElementURL and the concrete class of the object to the variable \_Element_Type. The variable \_Element_Type (rather than the query directly) defines the conditions that control which snippet is used to generate information about the object. The next section demonstrates how to produce documentation for a UML class.

**Class snippet**

The snippet responsible for defining the documentation for a UML class starts with a dynamic data source connection using the schema **DM Class** (see Figure 6). At runtime, this results in a request to the Design Manager REST service to fetch the details for the class. The value of the variable \_ElementURL is used to define the URI property of the data source.
Figure 6. Snippet for UML class

Next comes a query, $2$, which iterates over all classes returned (there is exactly one class). The name of the class and the list of the applied stereotype applications (query $14$) are rendered in the same paragraph followed by the Description (query $41$) in the next paragraph. The details of the class — such as the properties and operations — are rendered only if RenderElementDetails has been set to true. The value of this variable is passed from the main template to the snippet and set by the user in the document specification (see Figure 3).

Parameter passing

Some crucial technicalities are involved in parameter passing. The variable _ElementURL, which holds the URL of the referenced object on the diagram, is also declared in the snippet as an external variable and in the crawler as an internal variable. Consequently, parameter passing is enabled from the main template to the snippet, and the value of the variable can be used in the snippet to configure the dynamic data source DM Class. However, the variable isn't visible to the end user in the document specification.

Other variables passed to the snippet are RenderElementDetails, UMLTypes, and _DataSource. _DataSource defines the name of the inherited data source that the snippets data source connection uses to define parameters such as user credentials (user name, password). The dynamic data source is configured like this: Set the URI property to the value expression _ElementURL, and set the inherited data source property to _DataSource.

Note

Rational Publishing Engine 1.2.0.1 requires a workaround to make parameter passing between the crawler and the snippet work properly. You can maintain parameters in a separate template and then import this template in the snippet and the crawler to make the variables known to both templates. Then import the snippet in the crawler, selecting "Dynamic Referencing" as import type.
in the import wizard, and use the commands in the Import Template Wizard to map variables in the snippet to reuse template variables, as shown in Figure 7. The variables in the crawler and in the snippets are now bound together.

Figure 7. Parameter passing

![Import Template Wizard](image)

Layering

Rational Software Architect supports layering. You can group elements on a diagram into categories, or layers, that may be either visible or hidden. By doing so, you can control to a very fine degree what kind of details of an object — such as a class — should be shown on a specific diagram. A class might, for example, provide operations of relevance to different domains or themes. By using layers, you can create diagrams that deliberately hide specific themes.

Of course, reporting should be limited to render only the visible elements in the diagram. Figure 8 shows the solution to achieve this:
An internal variable called `_DiagramVisibleRefs` is used to hold the URLs of all elements in the visible layers. Rather than printing out all object references on the diagram (query $67$), define a condition just to consider those objects whose URL can be found in `_DiagramVisibleRefs` (see the condition for filtering layered objects).

Some technicalities are involved in doing that, however. `_DiagramVisibleRefs` is initialized to @. Next follows query $74$ that iterates over all `layers` in the diagram and extracts the visible layers using a filter. Query $75$ iterates over all `views`, followed by query $76$ that iterates over all `elements` of the view. The variable `_DiagramVisibleRefs` is then assigned as follows:

```javascript
_DiagramVisibleRefs  = _DiagramVisibleRefs + resourceID + "@";
```

A condition is used to test if the `resourceID` of the element on the diagram can be found in the variable `_DiagramVisibleRefs`. Only then is the corresponding snippet for the UML element called to render information for the element. Note that rather than using a string array, you can also use dynamic arrays (maps) combined with a simple lookup function to determine if an element is visible or not.

**When elements aren't on a layer**

This solution has one side effect: it renders information only for elements that are in a visible layer. However, there may be elements on a diagram that aren't on a layer at all, so the current template ignores them.

You can change this by adopting the model guideline that all elements on a layered diagram must be part of a layer. Alternatively, you can create a variant of the template that checks for hidden elements rather than visible elements — for example, it iterates over all hidden layers and collects the resource IDs for these object references. The condition shown in Figure 8
should then be changed accordingly so that it renders an element only if the resource ID is \textit{not} found in the variable. Finally, it would then make sense to change the name of the variable to _DiagramHiddenRefs to make the template more readable.

However, this small example demonstrates one essential aspect of generating highly proficient reports: (compliance to) model guidelines and well-working reporting templates go hand in hand. Without the former, you can't achieve the latter.

\textbf{Crawler variants}

You can develop specialized crawlers for different reporting needs — for example, one that implements the diagram-based reporting paradigm and another that implements the traditional model-based paradigm. The crawler that was implemented for the diagram-based paradigm was defined to take its root in a UML model for one reason: you can define a large number of document specifications for one and the same model with little extra work. To do so, use the Data Source Configuration Wizard to define an initial document specification for the model URL and set some of the parameters such as Package Name (and possibly Package ID). All the other document specifications can then be copied from this initial document specification, and all you have to do is change the package name and package ID (and possibly some other parameter).

\textbf{Disadvantage of developing specialized crawlers}

One disadvantage of this approach is that it requires an extensive use of internal parameters to find the top-level package in the model, to compute when a package is in scope for reporting, and to compute the style of the current section. Figure 9 shows an expression of the computations involved in determining if a package is in scope for reporting (inside or outside the designated top-level package) and what the corresponding style of the section header for the package should then be:
Rational Publishing Engine 1.2.1 has a much-improved Data Source Configuration Wizard, which makes configuring data sources easier. If you set the default project and workspace once, you can just use the wizard to navigate from there to the ULM model in scope for reporting.

An alternative to a report starting in the context of a model is to report directly on packages and use the wizard to select the top-level package directly. This method has two advantages. First, the logic in the crawler template is much simpler (most of the computations in Figure 9 aren't needed). Second, the template is faster, because less information needs to be retrieved for loading (large) models, and no execution time is spent traversing the model to find the top-level package.

**Deployment**

Diagram-based reporting is more likely to result in a larger set of document specifications than model-based reporting, because the scope for each generation is typically smaller (just a subset of the model rather that the entire model). Therefore, developing command scripts that generate all the documentation specifications making up a report is convenient. Figure 10 shows how to do this:
Figure 10. Batch execution

Alternatively, you can define document specifications with multiple references to the template, and then configure each template reference differently.

With the current implementation of Rational Publishing Engine, generating a template imposes a certain runtime overhead because of the time needed to open and check the snippets for consistency with the crawler. Before you send a template to production, import the snippets physically into the corresponding crawler and then use this variant of the template for report generation (but keep the original template for future revisions and modifications).

Assessing benefits and drawbacks

The diagram-based approach, which takes its origin in a set of diagrams in a model and then reports on the UML object references found in the diagram, generates reports in the context of Enterprise Architecture Frameworks, which demands that specific views be produced for a domain. The advantages of a diagram-based approach include flexibility in picking and choosing what to render in a report.

But you pay a price for this flexibility — at least with the current implementation. Since an object reference on a diagram has limited properties available (URL and concrete type), you have limited out-of-the-box capabilities for sorting elements on the diagram (for example, by name). Sorting can be done, but it requires a Javascript to look up and cache model element information before it is sorted and rendered.

In addition, diagram-based reporting has a runtime overhead compared to model-based reporting, because only object references are available. Model-based reporting also offers name, description, stereotypes, and so on directly, without retrieving details of an element by calling out to the REST service. Finally, redundancy may occur in a consolidated document: if an element has been referenced in 10 diagrams in the document, it is likely to be reported 10 times, unless specific — and potentially complex — actions are taken. These could include keeping a map of documented elements combined with using bookmarks and hyperlinks should an element already have been printed.
Conclusion

Users of Rational Publishing Engine traditionally employ a model-based approach to reporting, where reporting starts at some top-level package in the model and traverses recursively through the packages and renders diagrams and package elements. This approach has also been adopted to provide templates for IBM® Rational® Unified Process and the V-Model, for example, in earlier reporting solutions such as IBM® Rational® SoDA. However, the diagram-based approach offers greater flexibility.

Both the traditional model-based and the diagram-based reporting paradigm can be used for generating a consolidated document. You just need to consider carefully from the beginning which would be the best choice for a given reporting task; that is, a given section of the document. In fact, you can use both to generate a report — for example, a diagram-based crawler for generating architectural views and a more traditional model-based crawler for generating data model dictionaries.

Acknowledgement

I would like to thank Kevin Cornel for providing valuable tips and tricks for Design Manager Reporting over the last couple of years, and for giving me permission to introduce the techniques for reporting on layers, which are due to him.
Related topics

• Learn more about Rational Publishing Engine features and benefits:
  • Check the overview, get installation help, and read the documentation in the information center.
  • Watch the demo to see how you can use Rational Publishing Engine, including how to configure templates.
• "Tutorial: Design Management Reports," (Kevin Cornell, Adim Delibas, developerWorks [registration required]).
• For additional tips and tricks, visit Rational Publishing Engine Actual.
• Gather more information for the Reporting Area Web Publisher.
• To learn more about Rational Software Architect:
  • Start at the Rational Software Architect family of products page and be sure to check the developerWorks page.
  • Also, explore the product overview, the information center for installation and use instructions, and the Rational Software Architect wiki for more resources.
  • Check Jazz.net to find out more about Rational Software Architect Design Manager.
• For more help with templates and schemas:
  • For more tips, tutorials and sample templates, visit the Rational Publishing Engine Community wiki and check the Rational Publishing Engine Template library.
  • If part of what you want to include in your reports is in Microsoft Office documents, here's where you can find help: Go to either 2007 Office System: XML Schema Reference to download XSD schemas for Microsoft Office products. Navigate to Microsoft Office Project 2007 XML Data Interchange Schema to download the XSD schema required to report on Microsoft Project data. Read the Microsoft article titled Create an XML data file and XML schema file from worksheet data to learn more about how to create appropriate schemas for reporting Excel data.
  • Improve your skills. Check the Rational training and certification catalog, which includes many types of courses on a wide range of topics. You can take some of them anywhere, anytime, and many of the Getting Started ones are free.
• Download any of these for a trial at no charge:
  • Rational Software Architect
  • Rational Software Architect for WebSphere Software
  • Rational Software Architect Design Manager

© Copyright IBM Corporation 2014
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)