Automatically Connecting Documentation to Code with Rose

Robert Pierce
Staff Technical Writer
Rational Software Corporation
rpierce@rational.com

Scott Tilley
Department of Computer Science
University of California, Riverside
stilley@cs.ucr.edu

ABSTRACT

One of the most common problems with program documentation is keeping it synchronized with the source code it purports to explain. One solution to this problem is to automate the documentation process using reverse engineering technology. Reverse engineering is an emerging branch of software engineering that focuses on recreating high-level information (such as program documentation) from low-level artifacts (such as source code). This paper describes an automated approach to maintaining the connection between documentation and code by leveraging the reverse engineering capabilities built-in to Rational Rose. The approach produces application programming interface documentation for component object model-based (COM) dynamic link libraries (DLLs), C++ source code, and Java archive files. The documentation is always accurate and up-to-date. A primary advantage of the approach is its reliance on an industry-standard tool, thereby addressing one of the main concerns with facilitating wide-spread tool adoption: commercial-level support of deployed products.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – Documentation, Reverse Engineering

General Terms

Documentation, Design, Human Factors, Languages

Keywords

Documentation, application programming interface (API), software engineering, automation, single sourcing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGDOC'02, October 20-23, 2002, Toronto, Ontario, Canada.

Copyright 2002 ACM 1-58113-543-2/02/0010…$5.00.

1. INTRODUCTION

Keeping program documentation synchronized with the source code it purports to explain is one of the thorniest issues facing the software engineering and technical writing communities today. Programmers are notoriously reluctant to invest time and effort to document their own code. Technical writers face the challenge of trying to learn the essential details of the program by examining the source code (assuming they have access to it), interviewing the developers, and relying on their own experience with similar situations from the past. The result is poor-quality documentation that fails to meet its primary requirement: helping software engineers understand previously written applications, so that subsequent maintenance and evolution is made easier and more predictable.

Consider the situation of application programming interfaces (API), which are often the vehicle for introducing new software technologies. Trying to document a set of APIs can be challenging, because the details in an API typically change throughout the development lifecycle. Often, technical writers use a manual process that is repetitive and painstaking. They have to search through code, then search for the correct classes and their properties and methods. Keeping up with changes to the interfaces becomes daunting (and costly), as numerous changes to class structures, methods, arguments, data types, and return types take place from iteration to iteration. Too often, the end result is that API documentation is done only when the API is complete. This leaves programmers without a single, reliable written reference source during the development lifecycle, and it can leave customers in the lurch for days or even weeks after they receive the product.

One solution to this problem is to make program documentation an automated process. This can be done using reverse engineering, which is an emerging branch of software engineering that focuses on recreating high-level information (such as program documentation) from low-level artifacts (such as source code) [1]. Reverse engineering directly supports program understanding by providing the developer with information gleaned from the source code – information that can serve as a surrogate for missing program documentation.

Even when documentation exists, it is often inaccurate because it no longer reflects the current state of the program. The advantage of an automated approach to program documentation is that the documentation and code are always synchronized. The problem of having misleading documentation is completely alleviated by guaranteeing that the generated documentation is a true reflection of the underlying code.

There have been several projects related to connecting program documentation to program code. For example, Hartmann *et al* describe an automated approach to program redocumentation by producing graphical views of the software application using commercial reverse engineering tools in the embedded, real-time control system domain [2]. In [4], Priestley and Utt propose integrating documentation and development processes by incorporating program documentation as another activity in the Rational Unified Process [3]. Although the methods differ, the goal is the same: a closer connection between program documentation produced by technical writers and program code produced by software engineers.

This paper describes an automated approach to maintaining the connection between documentation and code by leveraging the reverse engineering capabilities built-in to Rational Rose [5]. The approach produces application programming interface documentation for component object model-based (COM) dynamic link libraries (DLLs), C++ source code, and Java archive files. A primary advantage of the approach is that it relies on a very popular industry-standard tool, thereby addressing one of the main concerns with facilitating wide-spread tool adoption: commercial-level support of deployed products.

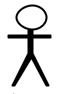
The next section of the paper provides a brief overview of Rational Rose and the Unified Modeling Language (UML). Rose has a great many software engineering capabilities beyond producing program documentation, and knowing a little about the usage of Rose in a larger development effort helps put the documentation process in context. Section 3 describes the automated documentation process, outlining the main steps the technical writer goes through to produce API documentation from code-level artifacts. Section 4 provides an example of the automated documentation process in use. Finally, Section 5 summarizes the paper and outlines possible avenues for future work.

2. RATIONAL ROSE AND THE UML

The *de facto* standard industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems in an object-oriented manner is the Unified Modeling Language™ (UML) [7]. The importance of models that use a standard nomenclature, common syntax, and agreed-upon semantics can hardly be understated. For many years the software engineering community suffered through a lack of a *linga franca* for discussing and reasoning about software designs. The so-called "three amigos" (Grady Booch, Jim Rumbaugh, and Ivar Jacobson) lead the effort to create a common language that supported modern object-oriented design principles. The result was the UML.

From the OMG Web site:

> *Modeling is the designing of software applications before coding. A model plays the analogous role in software development that blueprints and other plans (site maps, elevations, physical models) play in the building of a skyscraper. Using a model, those responsible for a software development project's success can assure themselves that business functionality is complete and correct, end-user needs are met, and program design supports requirements for scalability, robustness, security, extendibility, and other characteristics, before implementation in code renders changes difficult and expensive to make.*



The UML is a visual modeling language that relies on seemingly simplistic icons to represent complex software designs. For example, the user is represented as a simple stick figure (shown at right) – something even the most graphically-challenged engineer can draw. The use of such common graphical metaphors greatly aids the comprehension of complex UML diagrams. This is particularly important in terms of program documentation and improved communication. It means someone who is familiar with the UML (e.g., a technical writer, or a software engineer), will be able to understand the essential characteristics of the software design when it is represented as a UML diagram, even when the diagram is written by someone else. This is in contrast to the usual ad-hoc documentation techniques more traditionally used in software development.

The market-leading commercial application that most fully supports the use of UML is all phases of the software lifecycle in Rational Software Corporation's Rose. Rational

Rose is part of Rational's Enterprise Suite of software engineering tools. From Rational's product documentation:

> *By integrating the modeling and development environments using the Unified Modeling Language (UML), Rational Rose enables all team members to develop individually, communicate collaboratively and deliver better software.*

There is no doubt that Rose is primarily used as a forward development tool. It is normally used by software engineers during the early stages of the software lifecycle to record software designs, which can then be used to check against requirements, or to generate source code stubs that match the UML class diagrams.

However, Rose can also be used as a reverse engineering tool. Rose provides a single source for storing information in a variety of programming languages and deliverable formats, but it does not require a writer to have access to the source code. By storing content in a Rose model, the user can readily generate documentation using a SoDA (Software Documentation Automation) template. SoDA is a tool that extracts information from other software-engineering tools to generate accurate, comprehensive software documentation. The next section describes this process in more detail.

3. THE DOCUMENTATION PROCESS

Reverse engineering is essentially a process of extraction and abstraction. In contrast with typical software engineering, where one refines an abstract concept (e.g., requirements) into increasingly concrete terms (e.g., source code), reverse engineering goes in the opposite direction. The goal is to create accurate high-level representations of low-level information to foster better communication between team members in aid of program understanding.

3.1 Analysis and Abstraction in Rose

In the case of Rose, the extraction takes the form of automatic analysis of component object model-based (COM) dynamic link libraries (DLLs), C++ source code, and Java archive files. The abstraction process results in the creation of UML diagrams and associated annotations, which serve as program documentation. The documentation can be delivered in the following formats:

- Microsoft Word documents
- Adobe Acrobat PDF files

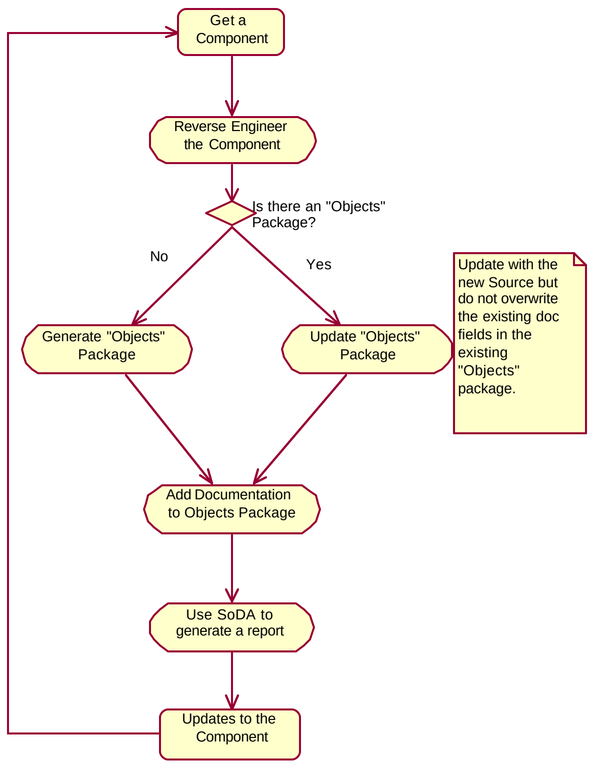

Get a Component
Reverse Engineer the Component
Is there an "Objects" Package?
No
Yes
Generate "Objects" Package
Update "Objects" Package
Update with the new Source but do not overwrite the existing doc fields in the existing "Objects" package.
Add Documentation to Objects Package
Use SoDA to generate a report
Updates to the Component

Figure 1: The API Documentation Process

- A set of API reference topics as individual HTML files, which can be linked in an XLST file to generate an HTML-based help system

The process described here focuses on the creation of API documentation. As illustrated in Figure 1, the basic process for producing API documentation is as follows:

1. Reverse engineer a component (e.g., a .dll (COM) or .jar (Java) file). The component represents the code that the developers are working on.
2. Create an Objects Package for a newly-created reverse engineered component by running a Rose script or Visual Basic application. The script copies code comments into the documentation fields in Rose.
3. Add documentation content to the Objects Package.
4. Generate a documentation report using SoDA.

The following sections detail each of these steps.

3.2 Reverse Engineer the Component

The first step in automatically generating API reference documentation is to reverse engineer a component containing the interfaces, objects, properties, and methods to be documented. For example, to reverse engineer a DLL file, one would do the following:

- Start Rose.
- Create a new file (click File > New).
- Drag the DLL from Windows Explorer and drop it onto the blank main class diagram (or onto the Logical View Package). A dialog will appear asking for Quick or Full Import.
- Select Full Import. A COM-based model will be created from the DLL file.

The result of these steps is a skeletal COM-based model that will contain the API documentation for the component.

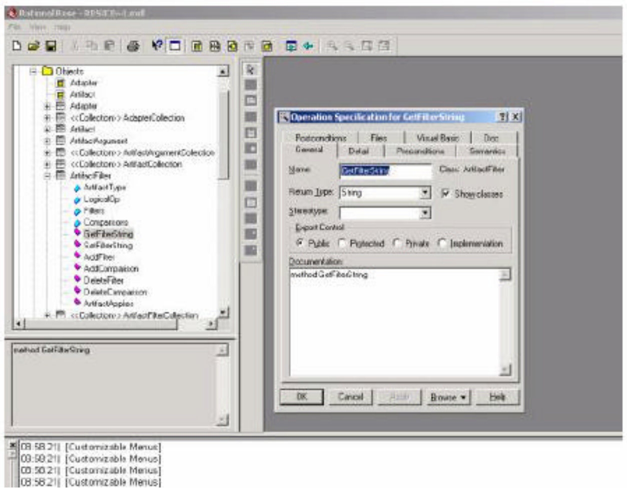
3.3 Create an Objects Package

After reverse engineering a DLL file into a COM model, it can be converted into a Visual Basic model (i.e., VB data types and interface structure). It can also be converted into a C++ model (i.e., C++ data types and interface structure). If a Java file was reverse engineered, it can be converted to a Java model (i.e., Java data types and interface structure).

An Objects Package for the reverse engineered component is created by running a Rose script or Visual Basic application. The Objects Package is where documentation content and dass diagrams are added. The script copies code comments into the documentation fields of the Objects Package in Rose.

In this context, a Rose package is a container or folder for the API documentation. Since the focus is on documenting an object-oriented API, the term "Objects Package" is used. When the component is reverse-engineered, the extracted information is stored in a sub-package of the Logical View package within the Rose model.

A copy of this sub-package is created in another sub-package called Objects. This is done so that documentation of the component's API takes place on this copy. A Rose script or the supplied *docgen* application can be used to insert the updated DLL information into the Objects Package without over-writing the generated documentation when the original DLL is reverse engineered later.



Name: GetFilterString Class: ArtifactFilter
Return Type: String [✓ Show classes]
Stereotype:
Export Control: [✓ Public] Protected Private Implementation
Documentation: method GetFilterString

method GetFilterString

```
08:58:21| [Customizable Menus]
08:58:21| [Customizable Menus]
08:58:21| [Customizable Menus]
08:58:21| [Customizable Menus]
```

Figure 2: Documenting a Method

This step can be customized with a Visual Basic application that lets the user clean up a DLL or a .jar file, and specify the type of API needed. It can also provide other features, such as generating a log file that notes all of the changes to an updated source.

3.4 Add Documentation Content

Once the Objects Package has been created, Rose can be used to add descriptions for all objects in the API. For example, content can be added to the documentation fields and class diagrams generated. Using the documentation fields for each object specification, single-source descriptions can be created for various aspects of the API (e.g., classes, attributes, operations, and parameters.)

Figure 2 shows a documentation field for the GetFilterString method of the ArtifactFilter class. Single-source documentation produced in this manner can greatly alleviate the maintenance and synchronization problems that are so common with program documentation related to code under development. Moreover, if both a Java API and a COM API are being maintained, then the common methods may be documented using a single description.

Standardized class diagrams for an API can be created by using simple drag and drop techniques provided by Rose. SoDA templates can be used to retrieve an Overview diagram for the API and class diagrams for classes in the API. The use of diagrams based on accepted standards also helps promote increased communication and code comprehension.

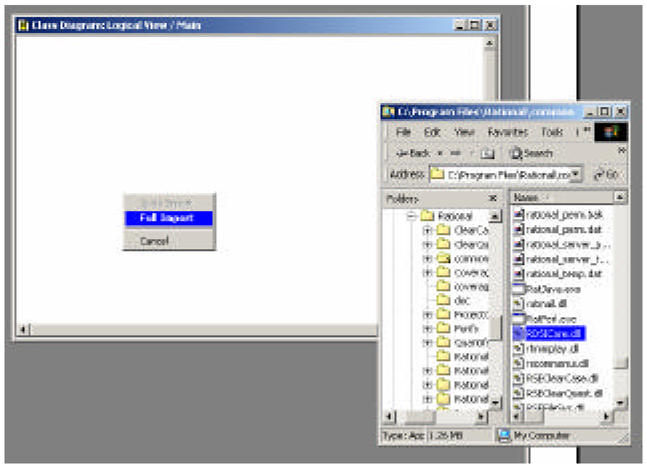


Figure 3: Reverse Engineering the Component

3.5 Generate a Report

To generate a report summarizing the API documentation, a standard SoDA template can be used. The template includes information for the following tasks:

- Generating a COM API (TEST_API_Template.doc)
- Generating a Java API
- Generating a C++ API
- Generating separate HTML files for each API topic

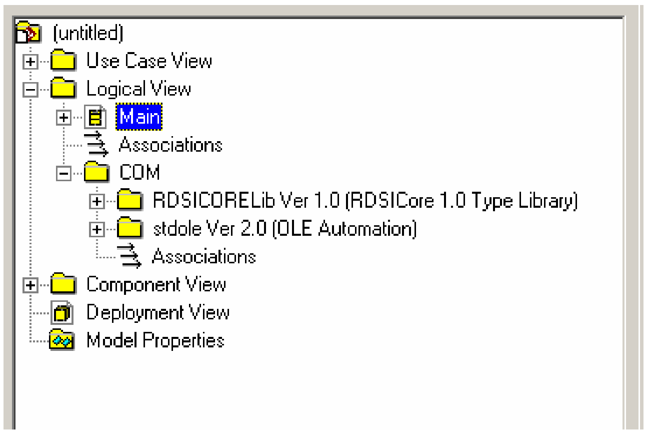
If desired, a customized template can also be used. For example, in cases where project-specific information is mandated by institutional requirements.

Generating a report is fairly straight forward. A SoDA template is used on the Objects Package to generate a Word document containing the skeleton information for all the API reference topics. Then, to generate the actual SoDA report, the following is done:

- From Rose, click Report > Soda Report
- Select the appropriate SoDA template
- Click File > Save As to store the automatically generated folders and topic files

4. A DOCUMENTATION EXAMPLE

To better illustrate the automated documentation process, an example of its use is provided. This example is a DLL that contains information of a client API callable in Visual Basic or C++. The goal in this example is to generate a reference manual for the Visual Basic programmer.



```
(untitled)
+ Use Case View
- Logical View
    + Main
      Associations
    - COM
        + RDSICORELib Ver 1.0 (RDSICore 1.0 Type Library)
        + stdole Ver 2.0 (OLE Automation)
          Associations
+ Component View
  Deployment View
  Model Properties
```

Figure 4: Documentation Sub-Packages

The documentation process begins by starting Rose to see the Main diagram in the Logical View window. The DLL component of interest is selected in Windows Explorer and drag onto the Rose workspace. A ‘Full Import’ is chosen, as illustrated in Figure 3.

After the reverse engineering process completes, a new sub-package in the Logical View folder is present. In Rose, these folders are called Packages. In this example, the new sub-package is named COM. In this case, there are additional sub-packages within the COM package. Only the packages that are relevant to the API documentation are needed; any other folders that are not needed to document the API can be deleted. In this example, there are two sub-packages, of which only one document is needed. So, the sub-package “stdole Ver 2.0 (OLE Automation)” is deleted. The sub-packages (with the to-be-deleted item still present) are shown in Figure 4.

When the sub-package is opened, it is apparent that a class diagram was automatically created by Rose during the reverse engineering process. This diagram can be used as the starting point for the API documentation, or new ones can be created from scratch. In this example, the diagram is saved as part of the Rose model SampleDLL. The class diagram is shown in Figure 5.

At this point, the Rose model could be saved and a SoDA template used to generate a report on the reverse engineered component. Detailed descriptions could be manually added to individual classes, methods and properties, which could then be included as part of the generated output. However, it is more helpful to first do some cleanup of the Rose model. This helps ensure that the data types for each method argument and return value are

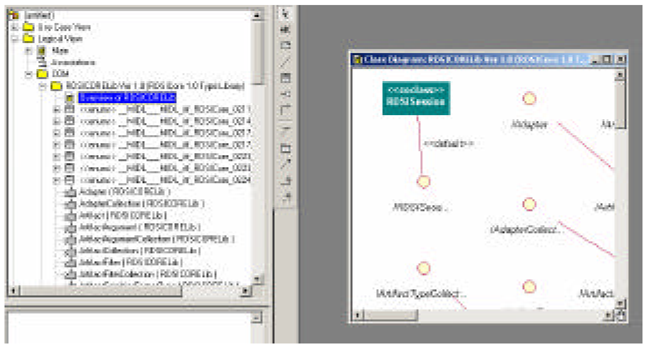


Figure 5: Generated Class Diagram

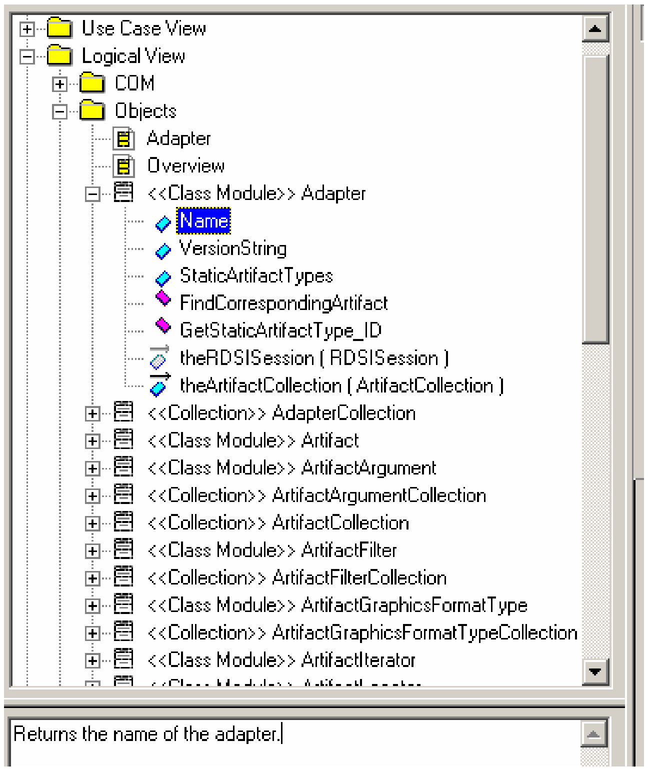
correct for the programming language in question (Visual Basic in this example).

Cleaning up the documentation in this manner can be accomplished using a script or tool that goes through the reverse engineering component and checks the return value and argument types for each method in the API documentation. Rose provides a script that automates this entire process.

The script is a simple program that converts all data types for return values and arguments to Visual Basic data types. It also converts COM interfaces and co-classes to the more straightforward object-oriented architecture of classes. Using the Rose script to convert the interfaces and co-classes to classes adds clarity to presenting the inherent structure of the objects that the API contains. Running the script results in the creation of a cleaned-up copy of the component documentation, in new packaged named Objects. The Objects package and its contents are then used by SoDA to generate a report.

For example, to add a description for a class, the user would click on the class and enter a description in the documentation field. If the class name is selected in Rose, the class properties and methods are shown in a message window below the Objects package. Each of the properties and methods can be selected in turn, to obtain or add more detailed information. In Figure 6, a detailed description for the Name property of the Adapter class is provided.

If desired, one can drill down deeper and retrieve or add information for any method that has arguments. This is down by right-clicking on the method and opening the Specification widget for the method. On the Details tab is shown the list of method arguments. Right-clicking on one of the arguments will open its Specification widget, where



```
Use Case View
Logical View
  COM
  Objects
    Adapter
    Overview
    <<Class Module>> Adapter
      Name
      VersionString
      StaticArtifactTypes
      FindCorrespondingArtifact
      GetStaticArtifactType_ID
      theRDSISession ( RDSISession )
      theArtifactCollection ( ArtifactCollection )
    <<Collection>> AdapterCollection
    <<Class Module>> Artifact
    <<Class Module>> ArtifactArgument
    <<Collection>> ArtifactArgumentCollection
    <<Collection>> ArtifactCollection
    <<Class Module>> ArtifactFilter
    <<Collection>> ArtifactFilterCollection
    <<Class Module>> ArtifactGraphicsFormatType
    <<Collection>> ArtifactGraphicsFormatTypeCollection
    <<Class Module>> ArtifactIterator

Returns the name of the adapter.
```

Figure 6: Detailed Property Documentation

extra information may be edited. This is illustrated in Figure 7.

The same information can also be documented graphically by creating class diagrams. These diagrams can then be included in the SoDA report. Typically the class diagrams would be created with the same name as the Interfaces (e.g., Adapter). More complex class diagrams can also be produced, for example showing relationships between several classes. Sample graphical documentation produced in this manner for the Adapter class is shown in Figure 8.

5. SUMMARY

This paper described an automated approach to maintaining the connection between documentation and code by leveraging the reverse engineering capabilities built-in to Rational Rose. The approach produces application programming interface documentation for component object model-based (COM) dynamic link libraries (DLLs), C++ source code, and Java archive files. An advantage of the approach is that the documentation and the code are kept synchronized through automated tool support.

Maintaining documentation for an API is an iterative process. As the software changes, so does the documentation. By using the reverse engineering features

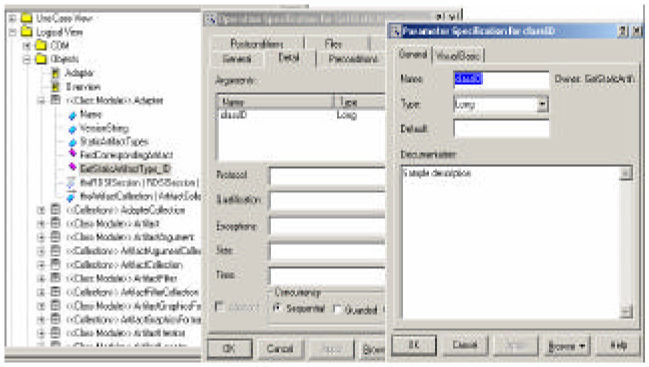


Figure 7: Detailed Method Argument Properties

in Rose, writers can work on documentation as a project develops and maintain accurate, up-to-date information. This approach not only supports development efforts with consistent and accurate documentation; it also provides the means to create clear, consistent diagrams in a format (UML) that developers are familiar with.

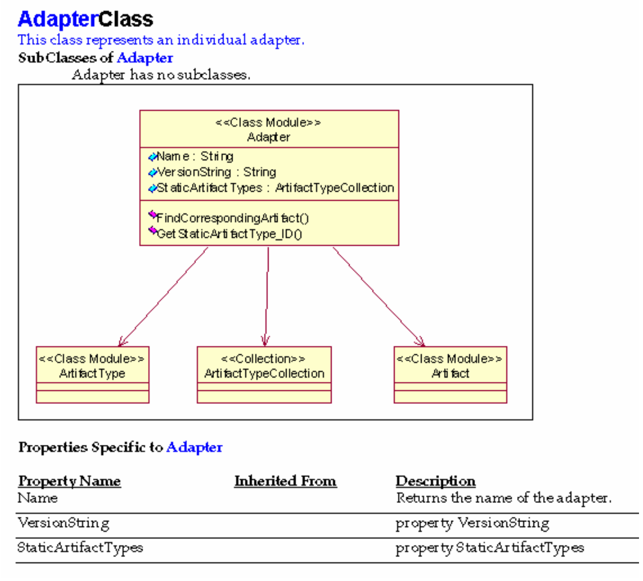
Using the process described in this paper, doing updates every few weeks as development of an API progresses can be extremely beneficial for developers, particularly if there is an audience using this documentation throughout the development lifecycle and not just at the end of a project.

To follow this automated solution, the updated component can be retrieved from the source code control system (e.g., Rational ClearCase®). Then, the component is reverse-engineered and the generated log file examined see what changes have taken place. Using this list, one can then browse through the Objects Package in the Rose model and make the necessary documentation updates. An up-to-date documentation package can then be produced using SoDA.

This process assists not just the writer but the whole development team. Developers will now have a reliable, up-to-date information source throughout the project. And customers will be assured to get complete and accurate documentation delivered right along with the product.

ACKNOWLEDGMENTS

Our thanks to Rational Software Corporation for allowing us to use tools and templates they had developed as well as encouraging the promotion of their best practices for all roles in the software development lifecycle.



AdapterClass

This class represents an individual adapter.

SubClasses of Adapter

Adapter has no subclasses.

Properties Specific to Adapter

Property Name	Inherited From	Description
Name		Returns the name of the adapter.
VersionString		property VersionString
StaticArtifactTypes		property StaticArtifactTypes

Figure 8: Graphical Class Documentation

REFERENCES

[1] Chikofsky, E.; and Cross, J. "Reverse Engineering and Design Recovery: A Taxonomy." *IEEE Software* 7(1):13-17, January 1990.

[2] Hartmann, J.; Huang, S.; and Tilley, S. "Documenting Software Systems with Views II: An Integrated Approach Based on XML." *Proceedings of the 19th Annual International Conference on Systems Documentation* (SIGDOC 2001: Santa Fe, NM; October 21-24, 2001), pp. 237-246. ACM Press: New York, NY, 2001.

[3] Kruchten, P. *The Rational Unified Process* (2nd edition). Addison-Wesley, 2000.

[4] Priestly, M.; and Utt, M. "A Unified Process for Software and Documentation Development." *Proceedings of the 18th Annual International Conference on Systems Documentation* (SIGDOC 2000: Boston, MA; September 24-27, 2001), pp. 221-238. IEEE: Piscataway, NJ, 2000.

[5] Rational Software Corporation. *Rose.* Online at www.rational.com/products/rose.

[6] Rational Software Corporation. *Soda*. Online at www.rational.com/products/soda.

[7] The Object Management Group. *The Unified Modeling Language*. Online at www.omg.org/uml.