Reverse Engineering Legacy Code with Rational Rose

by Francesco Bonfiglio
Technical Lead
Rational Software Italy

Faced with new sources of competition and new forms of partnership, many enterprises seek to integrate, streamline, or replace existing business applications and data sources. Often these efforts entail the intensive reuse of legacy code. Take a manufacturer that currently has a different, standalone quality management system (QMS) in each of its three plants. Many such companies are experiencing pressure to manage quality and other aspects of production across factories, to better serve both customers and suppliers. Whether management chooses to interconnect the existing quality systems or to standardize on a new, third-party offering, development staff will face significant challenges. Among them:

- The need to create a scalable, maintainable, and extendable solution in the shortest possible time.
- The need to work with object-oriented legacy code bases for which there may be no up-to-date architecture model, and little or no documentation.
- The need to incorporate off-the-shelf components from third parties.

Ideally, systems analysts and designers want to build new applications on a component-based foundation that leverages existing code wherever possible. But how best to model, build, and document these legacy components? One option is a laborious, manual re-engineering process, spearheaded by the programmers who know the code best. But few organizations have time for such brute-force efforts these days.

Fortunately, there is an alternative: to reverse engineer the legacy code into a visual model using Rational® Rose®, create appropriate components, and forward engineer the implementation code from there. While human
intervention is still required, reverse engineering with Rational Rose can greatly simplify and accelerate the task of delivering component-based applications that incorporate existing code. Specifically, Rational Rose can help you sort out critical modules much faster, align a visual model with business requirements and system requirements, and keep the implementation code, the model, and the requirements synchronized during the development and maintenance cycles.

In this article, I'll explain in detail how to use Rational Rose to reverse engineer existing object-oriented code, construct a UML model of the classes you need, and create robust, reusable, documented components from the model.

What Is Reverse Engineering?

Reverse engineering with Rational Rose comprises the creation of a UML model, based on the analysis of existing libraries and components. Designers can refine and enhance this model at any time during the software development lifecycle, and re-synchronize the model with the implementation code as needed (more on this below). With help from Rational Rose, teams can integrate legacy code into a new system architecture both more quickly and more accurately than is possible by hand. Provided, of course, that the code is supported by Rational Rose.

In its simplest form, reverse engineering using Rational Rose involves nothing more than the automatic generation of a class diagram from an object-oriented code base. The problem is that such a diagram is typically of little use, unless the code in question is already organized into discrete components or packages. Figure 1 illustrates what I mean. This example is not meant to be readable, but even if it were, the complexity would make it very difficult to interpret.
As you can see, code that did not begin life with a component-based structure does not often yield a picture that lends itself to easy interpretation. If the code was not well structured originally, then a class diagram does not make the relationships and dependencies among classes that much more obvious. For this reason, Rational Rose alone cannot adequately solve the problem of documenting, understanding, and repackaging most legacy code.

Instead, you need to start by working with the code step by step, in order to identify and organize the classes you actually want to reuse. Ultimately, you'll create separate class diagrams and use-case realization structures that define reusable components and can serve as a basis for generating implementation code. The process is straightforward, though it usually requires a day or more of concerted effort on the part of programming staff.

You might well ask, "Why is using Rational Rose better than starting from scratch?" Using Rational Rose is better because it simplifies and automates much of the step-by-step work you'd have to do anyhow. It documents the repackaged components automatically. And it relates your efforts to a UML model, thus leaving you with a far more maintainable and reusable end result. Today's new code is tomorrow's legacy code, after all.

The Business Case for Visual Modeling

One way Rational Rose helps automate the repackaging of legacy code is by enabling you to produce a visual model from your code, and then
automatically generate code from your model. When you change the code generated by the model, you can easily fold the changes back into the model, and vice versa. In the context of iterative development, this process of forward engineering, reverse engineering, and re-synchronization of implementation code with the model is collectively referred to as round-trip engineering.

The business reasons for building a visual model in the first place, and then keeping your code and your UML model synchronized during implementation, are convincing in my opinion:

- **Greater Efficiency.** A visual model saves time and reduces wasted effort by allowing all team members to visualize, understand, and refine business requirements for the solution and its software architecture, before coding even begins. A visual model also documents design and implementation decisions for future reference, and enables you to trace the architecture model back to both the business process model and the system requirements. Moreover, you can use the UML model to help automate the creation of test suites, design specs, and more.

- **Higher Quality.** A visual model helps the entire team visualize existing code as it is, in reality. Likewise, a visual model helps everyone visualize an application under construction as requirements mandate it should look. In effect, the model creates a blueprint that guides the construction of the application. So everyone has a clear idea of what the team is building, and how far along they are.

- **Greater Simplicity.** A visual model introduces consistency both within and across applications, for easier maintenance, greater architectural stability, easier reuse, and a better way to manage change and reduce complexity.

For example, why not manage class relationships within a UML model? Think about how difficult it can be to understand class relationships from reading code versus how easy it is to view class relationships in a Rational Rose class diagram. With Rational Rose, you simply:

1. Choose a class and expand it, as shown in Figure 2.
2. Select the elements you want to expand (see Figure 3).

3. Then you can clearly see all the relevant relationships pertaining to the class (see Figure 4).
Despite the automated help Rational Rose offers, some organizations choose to re-engineer legacy code from scratch without visual modeling. Others may wish to reverse engineer the code into UML with the help of Rational Rose, and then forward engineer it by hand.

In particular, I hear objections even from today's Rational Rose users about the value of automatic code generation ("No code generator can write better code than me!"). And that may well be true. But consider that perhaps 70 percent of object-oriented code like Java or C++ is structural code. The code generator can build that for you competently, whereas the manual task of writing all those declarations is time-consuming and subject to error. Particularly in a model-driven project, is it really worth it to model the use cases, extract all the classes, relations, operations, and attributes needed to realize them, create class diagrams, and then write the structural code by hand -- in effect, using the UML model only as a template? In my experience, Rational Rose performs those jobs quickly and reliably, while still enabling you to revise the generated code as required.

**Laying the Foundation**

It is imperative, in my experience, that any large-scale reverse engineering effort begin with an analysis session. The purpose of the analysis is to help analysts and designers understand what parts of the legacy system should be reverse engineered -- because you never have to reverse engineer *everything*. If they're using the Rational Unified Process® (RUP®), some organizations will
choose to precede an analysis of the code with business modeling; others will start with a formal requirements definition process. Either way, once you establish a clear understanding of what the system needs to do, you can begin analyzing the legacy system in order to develop use-case models in UML. These models not only specify what your system should do, but also guide your reverse engineering activities.

I usually plan on an analysis period of several days or less, depending on the size and complexity of the code base involved. The people it is most important to include are the hands-on developers and architects who know the code and its functionality best. For example, it is extremely important to have on hand someone who understands all the interconnection mechanisms, service mechanisms, relationships among modules, and so on.

Once the right people are assembled, you can begin identifying the classes to be reverse engineered. When upgrading existing systems, these might be subsystems that are highly dynamic, components that contribute to high failure rates, a legacy Web application, or code that has proven difficult to maintain. For new applications, these might simply be the legacy subsystems that can meet the system requirements.

**Moving in Reverse**

Having identified the key processes to be reused or adapted, you are ready to begin the reverse engineering process in earnest. Table 1 provides a quick overview of the steps involved.

At the conclusion of these steps, you will have created a package for each use case that shows its realization. Within each package, a class diagram identifies all the classes needed to perform the actions described in the use case. Each package will also contain a sequence diagram that identifies what messages are exchanged between objects in those classes, in order to implement those actions.

In effect, you will have created the foundation of a Design Model as described in the RUP. Readers who are familiar with the RUP may already have noticed that the reverse engineering process I just described looks somewhat like the
RUP Analysis and Design workflow, run in reverse.

You will also have created a comprehensive, robust visual representation of your system architecture, one that enables you to navigate from classes to processes and back. This model can be reused, maintained, and expanded along with the code, simply by re-synchronizing as needed.

Table 1: Steps for Reverse Engineering

<table>
<thead>
<tr>
<th>Step</th>
<th>Summary</th>
<th>Description</th>
<th>Roles Involved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Describe what the system will do.</td>
<td>Create a use-case model representing the principal processes within the legacy application that will be reused to meet the system requirements. Figure 5 illustrates a use-case model.</td>
<td>Business Analysts, System Analysts, Developers, Users</td>
</tr>
<tr>
<td>2.</td>
<td>Begin identifying the classes you need.</td>
<td>Interview developers to identify the classes within the legacy code that perform or support the actions described in the use cases. List the identified classes in a text note.</td>
<td>System Analysts, Designers, Developers</td>
</tr>
<tr>
<td>3.</td>
<td>Choose the correct settings.</td>
<td>Configure Rational Rose to optimally support the specific reverse engineering activity you want to perform. Settings will vary by programming language, IDE, and other factors. In particular, all elements should be referenced using path-map variables, as opposed to absolute paths. This is important because files might be installed in different locations on different systems. All external references therefore need to be relative. Figure 6 illustrates a path-map definition.</td>
<td>Developers, Designers</td>
</tr>
<tr>
<td></td>
<td>Reverse engineer the code.</td>
<td>Run Rational Rose against the appropriate subset of legacy code to reverse engineer the classes selected in Step 2 into the UML model. The result is a class diagram, as shown in Figure 7.</td>
<td>Developers, Designers</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>5.</td>
<td>Start organizing the classes into a logical structure.</td>
<td>In the Rational Rose Logical View, create a use-case realization structure (see Figure 8) that contains a package for each use case identified in Step 1. In each package, create a class diagram that identifies the classes that realize that specific use case.</td>
<td>Developers, Designers</td>
</tr>
<tr>
<td>6.</td>
<td>Identify and add more classes.</td>
<td>Beginning with the class diagram associated with each use case, work with those who know the code to add the main classes that realize the use case. Use the Query Expand capability (as shown in Figure 4) to get a complete picture of the additional classes involved. Then double-check the results with your team. Figure 9 illustrates an expanded class diagram.</td>
<td>Developers, Designers</td>
</tr>
<tr>
<td>7.</td>
<td>Create sequence diagrams.</td>
<td>Again, working closely with experienced developers, consider the behavior described in each use case. First identify the main flow of events; then identify one or more alternate flows. For each flow scenario, create a sequence diagram (see Figure 10) within the use-case package. Then populate the diagram with instances of the classes you identify as resolving the use case. Do this, at a minimum, for the main</td>
<td>Developers, Designers</td>
</tr>
</tbody>
</table>
Construct the flow of messages between objects.

Now work closely with the developers to specify the flow of messages between the objects that realize a specific flow scenario within a specific use case. Figure 11 shows a sample flow scenario. When you create a message in Rational Rose, you can right-click on it to see all the operations for that object. This greatly simplifies the process.

Developers, Designers

## Conclusion

As you can see, reverse engineering remains mostly a human-powered process. There is no Harry Potter magic involved, although the capabilities of Rational Rose simplify things dramatically. I've used this same, basic approach on some very large projects, and across many different technologies (VB, ASP, JSP, C++, etc.). Customers who adopt the process outlined above are almost invariably happy with the results.

Often, businesses that have positive experiences with reverse engineering begin to use UML and round-trip engineering on new projects. Thanks to these capabilities, there's no need to abandon the model and move into "uncharted territory" when the implementation diverges from the original model. Once you've created a model and a framework, it's a simple matter to reverse engineer the implementation code back into the model, within the framework. In fact, by simplifying the inclusion of ongoing changes into an updated design, reverse engineering can often become part of the day-to-day process of software development, not just a way to deal with legacy code.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!