Successful code reuse with code-centric development and modeling

Joanne L. Scouler
Martin R. Bakal

January 24, 2012

Modeling is an essential step in the process of analyzing existing code so that you can make decisions about what gets reused. When you understand the overall architecture of the code, building your products into a complete product line engineering (PLE) workflow becomes much easier to do and to maintain. You need to understand how the code fits together, how you could best use it in the future, and which parts you might want to modify. The main reasons to analyze code are for documentation, reuse, modification, or maintenance. This article explains the keys to successful reuse and how to reuse code by combining code-centric development with modeling.

New development projects often start with existing source code. Understanding the code is essential to reuse, which is not as simple as it seems. For example, maybe the original developer has moved on or retired. The code may have been written by multiple developers or come from multiple sources. The code might exist as code that your company owns, external code (for example, open source or off the shelf), or as libraries. It probably has changed and evolved over many years. Whatever the code history is, you need to know how the source code interacts and what it contains. You need to understand how the code fits together, how you could best use it in the future, and which parts you might want to modify.

Reasons for code analysis

The main reasons to analyze code can be summarized as: documentation, reuse, modification, or maintenance. Depending on the situation, the motivators are:

- Document your code so you understand it better and are prepared the next time that a developer retires or moves on ("documenting" implies that there are diagrams that explain the code)
- Build a product line with functionality in one product that you might not need in another
- Revise or update the application to a newer version
- Maintain your existing code

All of these reasons make good sense for understanding your code better. This article focuses on reasons for reuse and the code analysis that is necessary to get to reuse.
Risks with reuse

There are potential risks with reuse, some specific reasons why reuse can fail. One of the challenges in justifying the investment is that some managers will not allow a team to spend time understanding code that is old. Another potential risk for failure is lack of understanding of the domains and components of the code. Analysis paralysis is another potential risk.

Striving to understand all the code can be a huge task – breaking it down and identifying the key code to reuse is important. Take the time up front to assess which code components should be reused. In some cases it may be necessary to document everything in your code – such as for regulatory reasons but trying to reuse and document all your code is usually not efficient. For the most part, the goal is to be intelligent about which pieces of code to document, and which pieces to reuse.

The four keys to success

To achieve success the keys are to:

1. Understand the architecture of the original code to identify the components, boundaries, and interfaces
2. Determine what is potentially reusable
3. Estimate the time to reuse versus rebuild the components
4. Make a decision on a component by component basis on what to reuse and how to reuse - no change, minor update, major update

Clarify software code with the Unified Modeling Language

Whenever you want to acquire a clear understanding of your code the Unified Modeling language (UML) is a great way to do it. UML is a standard graphical language for representing code. It allows you to illustrate what the code is actually doing which in turn enables you to see the big picture of the component design. It enables you to understand the architecture of the original code to identify the components, boundaries, and interfaces. UML clarifies how the code fits together. You use the UML to analyze your code. Analysis is the first step prior to documenting the code you have.

Combining code-centric development with UML modeling

One of the four keys to success stated earlier in this article is to make a decision on a component by component basis on what to reuse and how to reuse it. If you want to maintain your existing code and document it, you could continue to do code-centric development the way you always have, but combine that with modeling. You probably would not need to generate your code in the model but you do want to better communicate and learn how the existing code fits together. This can be done with code-centric development combined with modeling.

If you want to understand existing software code in order to possibly reuse it to create new products or maintain multiple existing products, model-driven development may be the best solution. By doing code-centric development combined with model-driven development, you identify which part of your code you want to maintain and reuse.
Terminology definitions

**Model-centric**
The model not only drives the architecture but also your final application code.

**Code-centric**
Code is imported into the model as a reference for visualization. This means that the model is used to help show the architecture but that never changes the code. To update the design, you need to update the code and then bring it back into the model.

**Dynamic Model Code Associativity (DMCA)**
In the IBM® Rational® Rhapsody® modeling software, this refers to the direction of updates. You can go from model to code, or code to model, or you can have automatic bidirectional updates. *Round-tripping* is the process of bringing updates of the code into the model.

Different types of users prefer certain development scenarios over others. Users who like to work in models are referred to as "model-centric users," whereas code-centric users prefer to do everything in the code. And then there are people in between. You can accommodate the various types of users with the type of code reuse and modeling that you choose. The different kinds of users can collaborate and work together cohesively with a combination of traditional coding and model-driven development.

The technology definitions can be mapped into the three different scenarios described in the section that follows.

**Note:**
These can be used on a component-by-component basis, meaning that one component in an application can be model-centric, while another is code-centric.

**Scenarios for combining code-centric and model-driven development**
There are several ways that you can combine traditional code reuse with modeling. The spectrum of choices can be described, at a high level, as one of these four scenarios:

- **Visualizing external code** (visualization only) is a code-centric process. The external code stays external to your visual modeling tool. You reference the external code in the modeling tool to use and test it.
- **Ongoing coding with updated documentation** is a code-centric process with support for sending updates to the model. You can visualize code in a model, but never change the model.
- **Code-centric development** with some changes made in the model that are generated to the code. The code is the master.
- **Modernizing code using model-driven development**, where the model is the master. A portion of the external code is identified for reuse and brought directly into the model.

**Visualizing external code**
Visualization implies that you have diagrams that help you understand the design, structure of the code, and relationships between objects (such as classes, files). As a user you also have the ability to create new diagrams based on the model elements, add comments in the diagrams, and connect model elements to requirements. A description of the elements that arrived from the code is kept in the model and can be generated to a report by doing a report on the model. There are changes that can more easily be done in a model, such as adding elements that can be drawn on a diagram, copying elements on a diagram, inheritance and more.

With code visualization, updates to the code are done in the code outside the modeling tool. Visualization of the external code is done to show the relationships the code has to a model. For example, you may want to reference an external C library and show that reference in a diagram. The relationship the library has to other parts of the model is visualized and documented this way. Model-driven testing of the library can be done inside the model. With model-driven development not only do you visualize your code but you also execute your model to verify and test it.

**Ongoing coding with updated documentation**

In the second scenario, as changes are made to the external code, round-tripping can be done in order to send the code changes into the model. This brings in changed code from the external code thereby updating the documentation. If you choose the option to visualize and update code you can continue to develop the code in the model, or in the external code, and keep both in sync.

**Code-centric development**

As mentioned earlier in scenario 3, you can modernize code using model-driven development by generating code in Rational Rhapsody and updating the external code with the changes. Code generation can be done in the modeling tool while preserving existing code guidelines in the original source files. The option to run code generation in the model is useful for getting changes from the model back to the code. Model animation can be used to show the code behavior. In this scenario, the code is the master.

**Modernizing code using model-driven development**

You may choose to maintain the code in the model. This is the basis of scenario 4. The key criteria to use in determining if you want to visualize code as is done in scenario 1-3 or maintain it directly in the model to do model-driven development, is whether or not you want to generate code strictly in the model. If you maintain the code in the model, additional design changes and enhancements are more efficient to make. The model is the master.

Rational Rhapsody now shows a simple example of how UML enables reuse of existing code. This example is of a user building a new cash register based on existing hardware. The first diagram is called a Package diagram. It provides a high-level view of the packages of the system. The packages are the CashRegister, the Hardware package, and a link between those through the
Interfaces package. The Interface package allows easy reuse of the Hardware, as well as the ability to swap existing hardware for newer hardware as needed.

**Figure 1. Package diagram**

![Package diagram](image)

The class diagram below shows the Cash Register inheriting from the interface IBarcodeReader. This allows the CashRegister class to implement the interface and receive communication from the BarcodeReader class, which came from the external hardware package. The BarcodeReader class does not need to be shown in the diagram, although the Barcode Reader interface is shown (Figure 2), because the CashRegister needs to implement only the interface to get the desired behavior.

**Figure 2. Class diagram**

![Class diagram](image)

The class diagram in Figure 3 shows the Tester class doing an include of a C source file from an external source. The external source is visualized to show the operations and variables from the point_of_sale file so the Tester class can determine what it can call.
Successful code reuse with code-centric development and modeling

Figure 3. Tester class including a C source file from an external source

![Tester class including a C source file from an external source](image)

In Figure 4, an animated sequence diagram shows a CashRegister class scanning barcodes on products and adding the products to the customer bill.

Figure 4. Animated sequence diagram

![Animated sequence diagram](image)

Animation enables you to execute an application on either a host or a target and then view results in the design with Rational Rhapsody. This is an important tool for viewing code behavior and debugging code, which is important when you are reusing code. Viewing static code shows you only potential paths through it. A sequence diagram like the one in Figure 4 shows the path through the application as it actually responds to an external stimulus. Consequently, you see what path gets executed in the real lifecycle of an application. So animating a model is a way to test and verify your reusable code behavior.
Reuse and product line engineering, or PLE

The holy grail of reuse is something called **product line engineering** (PLE). It is the practice of identifying features in your application that will vary from one product to the next and mapping those features to specific variations of each product so that, in the end, you have one set of code that supports multiple products. This is the holy grail because the main goal of reuse is to reuse in the next product coming out. That is also the goal of PLE.

The other options for building multiple products are either clone and own or using **ifdefs**.

- Clone and own brings up a host of issues, such as when you find a bug in one product, does it exist in the other products, as well? What happens when you want to propagate a feature through the products? This becomes a nightmare in looking at changes in the clones.
- Ifdefs make the code unreadable and very tough to navigate when trying to work on one specific product.

Modeling helps resolve this dilemma by adding a level of abstraction above the code. You can identify a specific model element (a class, function, or anything else) as mapping to a feature and then add tags for the product-specific differences. Because it is graphic, the tags can be links that can be investigated as needed. Also, when you generate the code from the model, it uses only the tags for the specific product, so you get customized code for each product. Then you make changes in the model, and every product that has that feature gets those changes. There is no need to propagate changes from product to product.

**Summary**

In summary, because timelines are short, reuse is essential in meeting deadlines. The key is how to build effective reuse into what you do. First, be sure that you understand your architecture, and then dive deeper into the components that you want to reuse. Modeling is valuable in this process because it helps you analyze the code and make decisions about what gets reused. When you understand the overall architecture, building your products into a complete PLE workflow becomes much easier to do and to maintain.
Related topics

- To learn more about this tool for collaborative, model-driven development for embedded systems, start with the Introduction to Rational Rhapsody and the Rational Rhapsody page on IBM developerWorks. For documentation, see the Rational Rhapsody 7.6 information center (see Changing the location of help content to get a local copy of the documentation).
- Check the IBM® Rational® Rhapsody® Design Manager to collaborate, share, review and manage designs and models with the entire engineering team.
- 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, any time, and many of the "Getting Started" ones are free.
- Download Rational Rhapsody Developer and try it free for 30 days.
- Evaluate IBM software in the way that suits you best.

© Copyright IBM Corporation 2012
Trademarks
(www.ibm.com/developerworks.ibm/trademarks/)