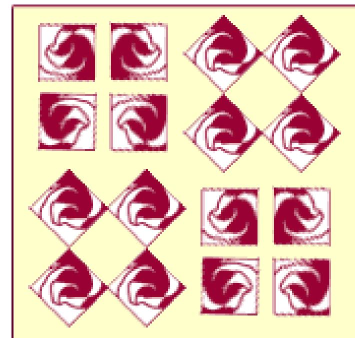
Pattern-Oriented Development with Rational Rose

Professor Peter Forbrig, Department of Computer Science, University of Rostock, Germany;
Dr. Ralf Laemmel, Department of Information Management and Software Engineering, Free University, Amsterdam; and
Danko Mannhaupt, Software Engineer, Software Design & Management (SD&M), Munich, Germany



A design pattern describes a solution to a recurring problem in a systematic and general way, and design patterns are an accepted means of representing a communication experience in software design. Up until now, however, only single patterns have been used in case tools; there has been no support for combining patterns. This article will show how patterns can be combined in Rational Rose to develop new patterns. It will also show how the entire software specification process can be based on a combination of patterns.

The ideas in this article are based on a pattern-oriented programming model developed in a master's thesis by Normen Seemann in 1999 at the University of Rostock's Department of Computer Science. This model led to the development and implementation (by Stefan Buennig and others in the department) of a Pattern-oriented Language (PaL) and PaL-based graphical editor. Based on this work, Danko Mannhaupt recently developed a set of scripts for Rational Rose that supports most features of the pattern-oriented programming model to accomplish object-oriented design.

Limitations of Object-Oriented Specifications

Object-oriented specifications are very successful because they make it possible to reuse existing models. But they have their limitations. Let's say we have two classes, A and B, along with their methods mA and mB. Class B inherits from class A, so that B has two methods: mB and mA, inherited from A. The corresponding class diagram is shown in Figure 1.

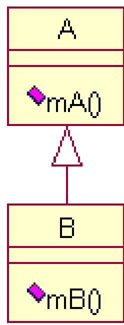


Figure 1: A Class Diagram for a Simple Inheritance

Now, suppose we have two other classes: AS with method mAS, and BS with method mBS. Theoretically, the scheme of inheritance from A to B should be reusable for these two classes. One arrangement would be for AS to inherit from A and BS to inherit from B. Figure 2 shows the corresponding class diagram.

But the problem is that now, BS does not inherit from AS. That inheritance relationship could be inserted manually, but that would create a multiple inheritance problem. In this article, we explain how to tackle this problem by using Rational Rose scripts. First, however, it is important to understand our programming model.

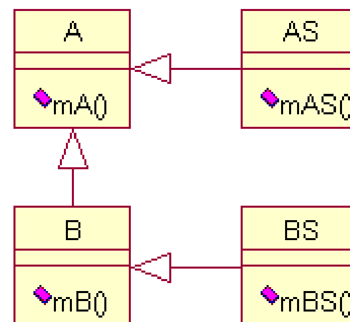


Figure 2: Attempt to Reuse the Inheritance Structure of Figure 1

The Extended Object-Oriented Programming Model

In the traditional programming model, patterns must be coded as conglomerations of classes. This results in both a lack of traceability and encapsulation. Patterns should be traceable in the source text so that the reader can easily identify them. E. E. Jacobsen argues that patterns are abstractions operating on top of programs. The new programming model provides a corresponding kind of abstraction to protect the overall structure underlying a pattern. It treats a pattern as a nested class that encapsulates participating classes.

In the object-oriented world, reuse is based on concepts such as inheritance, composition, genericity, and interfaces. These concepts, however, are not sufficient to implement patterns in a reusable way. Instead, programmers are forced to code the solution a pattern provides for each specific application context.

The simple example in Figure 2 illustrates the limitations of inheritance when it comes to facilitating reuse of class structures; the original inheritance relationship is not preserved in the refined class system (AS

and BS). Among other problems, the refined system cannot support polymorphism sufficiently. Also, generic classes do not provide a general solution for refinements of class structures. The duplication of classes participating in a pattern cannot be modelled within that framework, but we are able to do so within ours.

Figure 3 shows two patterns that introduce superimposition on class structures as another form of reuse. This is "grey-box" reuse rather than white-box reuse by inheritance, which works at the class level. In graphical notation, superimposition can be depicted with arrows. Entities (e.g., class structures, participating classes, or methods) are connected with dotted lines. Arrows, which go from the superimposed class structure to the resulting class structure, are used only if the classes or methods are renamed. If the class or method names remain the same, then the arrows are omitted. Note that the graphical notation does not emphasize parts of the resulting structure that are *not* provided by the reused class structures. Instead, it emphasizes the resulting structure and indicates the reused parts.

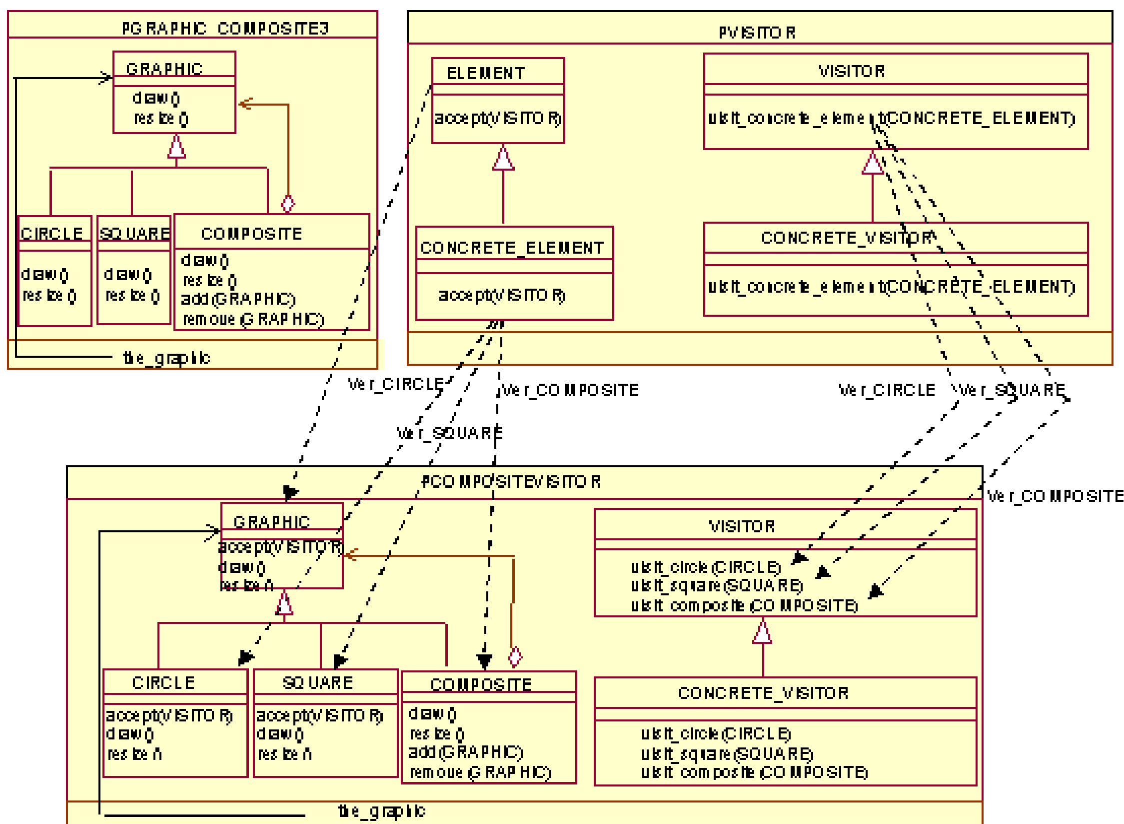


Figure 3: Class Structures and Superimposition: Visitor and Composite
(View full size graphic in new window)

The right portion of Figure 3 illustrates the class structure underlying the Composite pattern and its derivation by superimposition. The final class structure, PCOMPOSITE, is a slight abstraction of the corresponding GoF variant. There is an abstract superclass COMPONENT with a method operation and the subclass LEAF and COMPOSITE. The latter class enforces an interface to add and remove components. COMPONENT declares an abstract method operation. The abstract class PARAMETER models parameters for operation.

Now, let us focus on superimposition. The class structure PCOMPOSITE was derived from the class structure (auxiliary patterns) PCONTAINER, modelling a minimal interface for container functionality, and PARAMETER, concerned with the idiom for methods with abstract parameters. Note that PCOMPOSITE adds structure to the reused class structures. LEAF was not present at all in PCONTAINER. Also, the inheritance relationship is established as required for composite. Finally, note that classes enclosed by PCONTAINER are renamed in PCOMPOSITE.

The combination of patterns is an even more intricate problem than the reusable implementation of patterns. In the suggested model, combination is made possible by certain key features of superimposition that facilitate reuse of class structures. It is possible, for example, to unify or duplicate classes in a class structure and merge different class structures. It is impossible, however, to perform such adaptations in terms of genericity and inheritance without breaking the class structure.

Integrating Patterns into Rational Rose

With this understanding of our model, we can now explore how to integrate object-oriented design based on patterns with a CASE tool such as Rational Rose.

Introducing a new static element
To supplement the existing static elements in Rational Rose -- *Package, Class* and *Interface* -- let us add a new model element: *Pattern*. This element is a container for components, similar to a package. It has a name and may or may not be part of a package. Classes and interfaces can be pattern components. (Patterns can also be components inside of other patterns, but this would add unnecessary complexity to our model.)

The pattern model offers combination as a means of teaming up patterns. Associations and inheritance relationships complete the static model of the new pattern element.

The pattern specification includes a detailed description like those in pattern catalogs. Potential documentation fields are: motivation, application, domains, benefits, liabilities, consequences, examples, and related patterns.

Rational Rose can represent patterns -- which are related to packages -- in much the same way as packages. Rose's model tree can include patterns as structural elements with components and relationships. Each pattern has at least one class diagram that shows the pattern structure. A new type of diagram, the *Pattern Diagram*, shows patterns and relationships between them. So Rational Rose displays both the development of the pattern model and the architecture of the system.

Specifying dynamic aspects
The specification of dynamic aspects of a pattern is very important from

the point of view of documenting. Therefore, it must be possible to create collaboration, sequence, and activity diagrams for a pattern and its components. These diagrams document co-operations and interactions among class-components. They support developers with detailed design specifications, because they describe responsibilities and interfaces of components. Furthermore, interaction diagrams simplify implementation of pattern classes with object-oriented programming languages.

Working with Design Patterns in Rational Rose

Creating patterns
Patterns can be created within Rational Rose using the menu bar, the context menu of packages or patterns in the model tree view, or the diagram presentation. An empty design pattern without components is created at the current level -- that is, as part of the currently selected package.

Editing patterns
Pattern properties and components can be edited similarly to packages, using either the model browser or diagram views. A developer can initiate changes to the specification and documentation of the pattern itself by selecting a command from the pattern's context menu. This produces a dialog box that shows pattern properties. Pattern components are edited as if they were member classes of a package. Consequently, developers can add components by selecting a command from the menu bar or the pattern's context menu and edit these components using standard Rational Rose tools. They can also add and change attributes and operations and edit specifications and documentation.

They can also add interaction diagrams that include pattern components and their instantiated objects. Interaction diagrams document a pattern's behavior in certain scenarios. Therefore, a number of different diagrams are required to describe a pattern in detail.

Refining and combining patterns
The refinement and combination process represents the core of the pattern model. The difference between refinement and combination is the number of source patterns involved in constructing a new design pattern. With refinement, one pattern is edited toward a particular domain or application. This can be achieved by editing a pattern's properties and adding or changing pattern components.

Combination refers to the creation of a pattern based on a number of source patterns. Combination and self-combination (the same pattern occurs several times as a source pattern) represent a challenge to CASE tools. Essentially, a number of model elements have to be merged into one, and several requirements have to be taken into consideration.

Components
By default, the component set of the new pattern comprises the union of the component sets of all source patterns. It can be conceptualized as a "bag" that contains more than one exemplar of the same component if the designer has combined patterns with identical components.

The designer has several options to change the default component set-up. Components can be merged, but only if they are from different patterns. Typically, the number of combined components would be two or less, but in rare cases it may be necessary to merge more than two components.

By default, a combined component contains all attributes and operations from its source components. In fact, however, although components in combined patterns can be joined, attributes and operations may or may not be joined. Let us suppose, for instance, that each of two source components has an attribute date. Only one attribute of this kind is necessary for the combined component. Deletion of properties would not be supported, however, because that would reduce functionality. Therefore, instead of deleting one attribute, the designer would join both attributes. That way, the specification and documentation of both source attributes would be sustained, and the semantics would remain intact. Associations of source components would also be preserved in the combined component. Once again, however, associations can be joined to remove redundancies. The same applies to inheritance relationships, but joining inheritance relationships can create multiple inheritance situations.

A Prototype Implementation of the Pattern-Oriented Approach

Rational Rose provides an extensible interface that allows users and developers to enhance its functionality. This interface provides access to model elements, so that existing objects can be manipulated or removed and new objects can be added. Access is provided via an ActiveX control or by using RoseScript, a Basic environment within Rose.

At the University of Rostock, two scripts were developed to create a new pattern and to combine patterns. Both scripts can be executed manually, but they can also be integrated into the menu bars. A text menu file defines menu extensions.

The menu file for the pattern extension is presented in Figure 4. It defines a new sub-menu -- "Pattern" -- for the tools menu in Rose. Figure 5 shows how this sub-menu actually appears in Rational Rose.

```
Menu Tools
{

     Separator
     Menu "Pattern"
     {

          option "&New Pattern"
          {
           RoseScript
          $PATTERN_PATH\Scripts\new_pattern.ebs
          }

          option "&Combine Patterns"
          {
           RoseScript
          $PATTERN_PATH\Scripts\combine.ebs
          }

     }

}
```

Figure 4: Pattern Extension Menu File: pattern.mnu

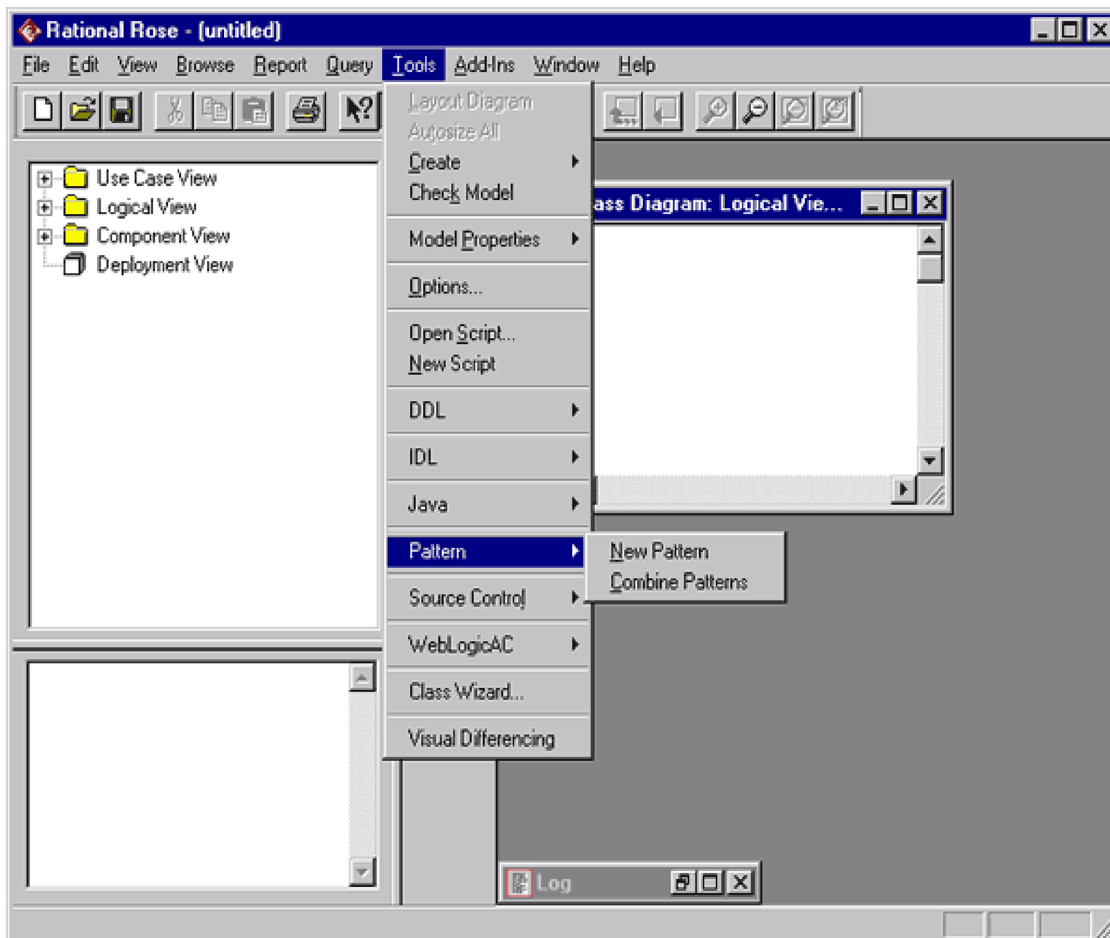


Figure 5: Visual Representation of the Extended Sub-menu Tools in Rational Rose
(View full size graphic in new window)

Using Pattern as a Model Element

Rational Rose does not support the sort of fundamental extensions that would permit us to implement a new model element with unique properties, so as an alternative, we used Package, a standard Rose element, as a container for pattern components.

The ability of packages to function as containers for other elements also enables them to serve as pattern representations in Rational Rose. However, they do not make it possible to reference pattern components in other patterns. Nor is it possible to indicate that a pattern contains other patterns.

To identify the new pattern packages as distinct from ordinary packages, we use UML stereotypes. Every pattern is marked with <<Pattern>>, and the stereotype was added to the standard stereotype list.

Components of patterns are regular classes without either stereotypes or extensions. Therefore, whenever something is written about pattern components, it can be assumed that they contain all the properties and features of classes.

The pattern interface is represented as a designated component of the pattern. It is marked with the stereotype <<PatternInterface>>. Like the pattern model, the interface contains public properties of the pattern and provides services to other patterns or independent classes. Properties are represented with attributes that often refer to an instance of one of the pattern components -- the starting point of a pattern structure.

Documentation for patterns can be placed in standard documentation fields for packages. We recommend that the documentation text be structured according to standard pattern descriptions, stating the context, problem description, solution, consequences, and perhaps examples. The documentation can also include references to other patterns, keywords, benefits, and liabilities.

With static and dynamic diagrams, designers can visualize pattern properties and behavior. Class diagrams are used to show components and their relationships, including associations and inheritance.

The file PatternStereotypes.ini, for example, contains the following definition:

```
[Stereotyped Items]
Package:Pattern
Class:PatternInterface
```

```
[Package:Pattern]
Item=Package
Stereotype=Pattern

[Class:PatternInterface]
Item=Class
Stereotype=PatternInterface
```

Manipulating Design Patterns

To understand how to manipulate design patterns in Rational Rose, let us return to our first example: the simple reuse of an inheritance structure. First, we need to develop a pattern for the inheritance relationship, which should be reusable. Let us call this pattern InheritanceAtoB. Next we must specify a pattern that contains both classes AS and BS. We will call this pattern, which contains only these classes, PatternASandBS. We create these patterns the same way we ordinarily create class diagrams, which we will not describe in detail here. Figure 10 shows both patterns displayed in Rose.

If we combine these patterns, we see the window in Figure 6.

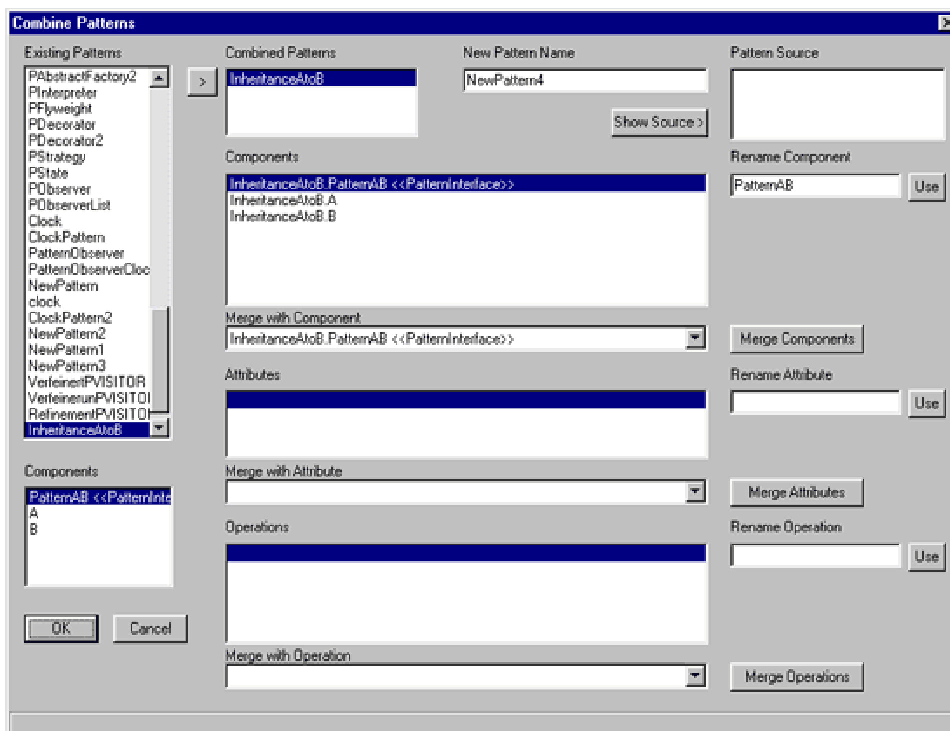


Figure 6: Screen Dump After Selecting a Pattern: PatternAtoB.
(View full size graphic in new window)

Existing patterns are displayed in the left box; we can click on one to

select it. Using the (>) button at the top, we can insert the selected pattern into the list of patterns in the Combined Patterns box, and the components are displayed below. If we select a component, we can see its attributes and methods. According to our theory, a pattern can have higher-level methods, which are represented by an interface class. Although these classes are used here, they have no importance for this example.

Figure 7 shows a screen dump following the selection of PatternASandBS. The resulting combination pattern now bears the name ReusedInheritance.

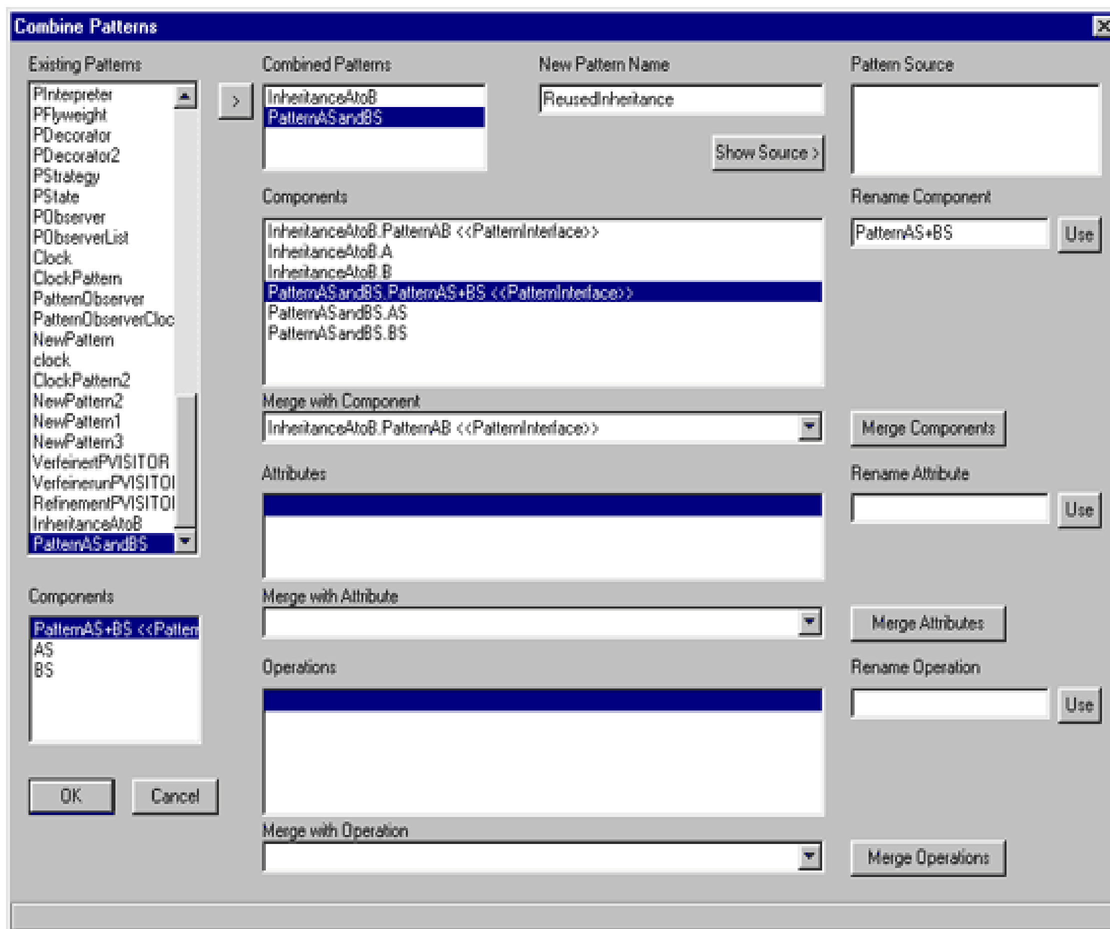


Figure 7: Screen Dump Following the Selection of a Second Pattern.
(View full size graphic in new window)

Now we have to describe which components are merged together. We click first on a component we select within the listbox and then click on a second component in a drop-down menu in a listbox. According to the standard, the first element

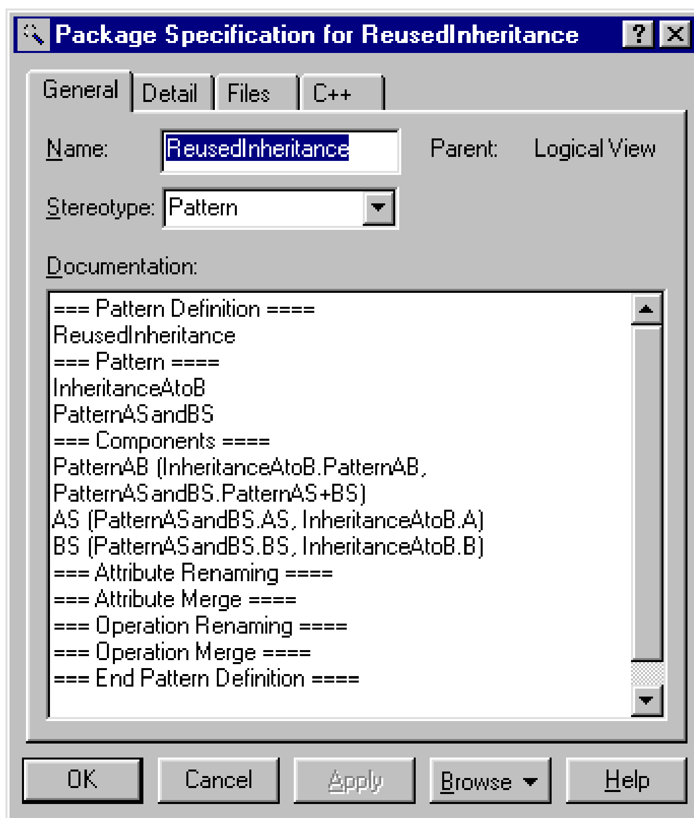


Figure 8: Displaying the Results After Combining Two Patterns

determines the name of the combination, but this can be changed manually. The procedure would be the same if attributes or methods of a component had to be merged. Once all necessary merges have been performed, we can press the OK button to generate the combined pattern. The first window to appear summarizes the combinations, as shown in Figure 8.

In our example, the generated class diagram for the new pattern is already perfect. No manual changes in the layout are necessary. Of course, this is not always the case, and manual changes are sometimes necessary.

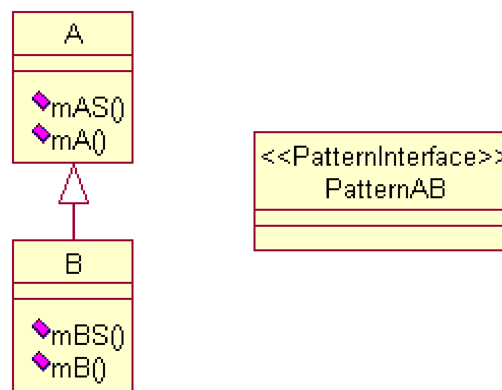


Figure 9: Class Diagram of the Resulting New Pattern

The class diagram, which can be found within the pattern (package) ReusedInheritance within the browser (see also left side of Figure 10), appears as shown in Figure 9.

Class AS plays the role of class A and BS the role of class B. These new classes reuse the inheritance structure of the given pattern and also inherit the methods of the corresponding classes.

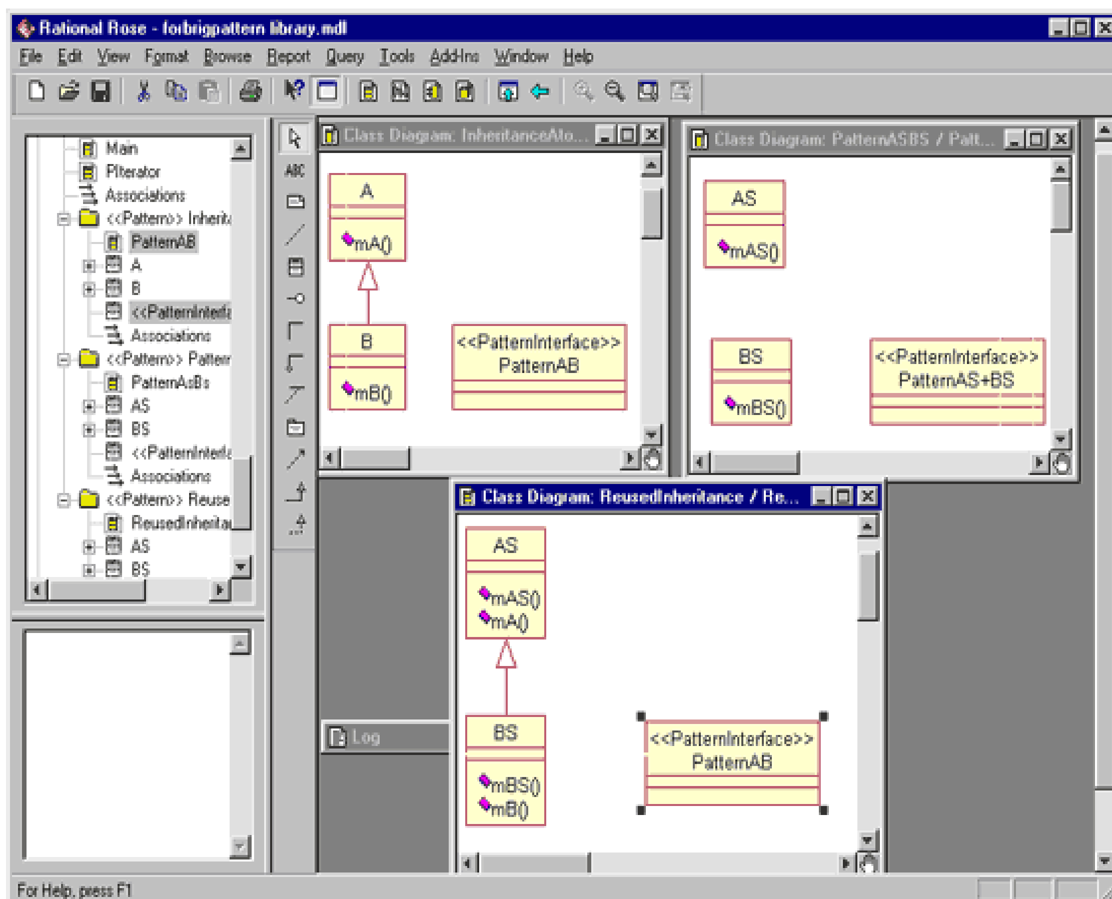


Figure 10: Two Patterns that were Combined and the Resulting New Pattern
(View full size graphic in new window)

Conclusion

In this article, we have shown that a pattern-oriented programming model tool can support pattern-oriented design, which, in turn, provides for the kind of reuse that an object-oriented approach cannot support. If you would like to try out the patterns approach yourself, you can download the scripts for Rational Rose that we used in our prototype implementation from the file RosePatterns.zip at http://wwwswt.informatik.uni-rostock.de/dokumente/patterns.

The script new_pattern.ebs generates a new, empty pattern. Components can be inserted into this pattern in the traditional way that class diagrams are developed. The second script, combine.ebs, is used for combining existing patterns, as described in this article. A library of services used by the scripts is implemented by library.ebs, and a compiled version is library.ebx.

The files pattern.mnu, pattern_addin.reg, and patternstereotypes.ini are necessary to register the stereotypes and new menu items. The file readme.txt describes what to do with these files.A library of patterns is also available as a Rose model.

References

Bosch, J.: "Design Patterns & Frameworks: On the Issue of Language Support." In Bosch, J.; Hedin, G.; Koskomies, K. (Ed.), *Proceedings*, LSDF'97: Workshop on Language Support for Design Patterns and Object-Oriented Frameworks, 1997.
http://www.hk-r.se/

Bosch, J.: "Design Patterns as Language Constructs." *Journal of Object-Oriented Programming*, 11(2): 18-32, May 1998.

Budinsky, F.J.; Finnie, M.A.; Vlissides, J.M.; Yu, P. S.: "Automatic Code Generation from Design Patterns," *IBM Systems Journal*, 35(2), 1996.

Buennig, S.: "Entwicklung einer Sprache zur Unterstuetzung von Design Patterns und Implementierung eines dazugehoerigen Compilers." Master's thesis, University of Rostock, Department of Computer Science, 1999.

Buennig, S.; Forbrig, P.; Laemmel, R.; Seemann, N.: "A Programming Language for Design Patterns." Informatik 99, Reihe Informatik aktuell, Springer, 1999. Presented at Arbeitstagung Programmiersprachen'99.

Forbrig, P.; Laemmel, R.: "Programming with Patterns." TOOLS 2000, Santa Barbara, California. Proceedings, Tools 34-USA 2000, IEEE Computer Society. http://www.eiffel.com/tools/index.html

Gamma, E.; Helm, R.; Johnson, S.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

Jacobsen, E.E.: "Design Patterns as Program Extracts. In Bosch, J.; Hedin, G.; Koskomies, K. (Ed.), *Proceedings*, LSDF'97, Workshop on Language Support for Design Patterns and Object-Oriented Frameworks, 1997.
http://www.hk-r.se

Mannhaupt, D.: "Integration of Design Patterns into Object-Oriented Design Using Rational Rose." Master's thesis, University of Rostock, Department of Computer Science, 2000.

Palsberg, J.; Schwartzbach, M.I.: "Type Substitution for Object-oriented Programming." SIGPLAN Notices, 25(10), October 2000. Proceedings, OOPSLA/ECOOP'90 (European Conference on Object-Oriented Programming.

Pulsipher, D.: "Defining and Using Design Patterns in Rational Rose." July, 1999. http://www.qoses.com/ruc/Quarry/index.htm

Seemann, N.: "A Design Pattern Oriented Programming Environment." Master's thesis, University of Rostock, Department of Computer Science, 1999.

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

Copyright Rational Software 2001 | Privacy/Legal Information