

A Metamodeling Approach to Pattern-Based Model Refactoring

Robert France, Sudipto Ghosh, Eunjee Song, and Dae-Kyoo Kim,
Colorado State University

The success of model-driven architecture methods hinges on the support they provide for three things: creating meaningful models in a timely way; specifying and applying patterns that reflect useful and reusable design abstractions; and defining and systematically applying model transformations that support model evolution, refinement, and code generation. Model transformation occurs when we modify a source model to produce a target model. Applying a well-defined

design pattern is one way to transform a model. For example, a developer might modify an existing design using a pattern to produce a target model in which the pattern is realized.^{1,2} This article shows how to define and apply design patterns in the context of model transformations. The design models we discuss, both source and target, are expressed in the Unified Modeling Language.³

Pattern-based model refactoring

The process of transforming a model using a design pattern is called *pattern-based refactoring*. We can achieve rigorous pattern-based refactoring by developing metamodels called *transformation specifications* that characterize families of transformations. Or, you might think of metamodels as defining a transformation language; the characterized transformations are “sentences” in this language. The metamodels act as points against which we can check model transformations for conformance.

Figure 1 shows an example of pattern-based model refactoring using the Gang of Four’s Bridge pattern.¹ (The Gang, also known as “GoF,” includes Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.) The Bridge pattern shown in Figure 2 describes

Design patterns capture some of the best software development solutions to design problems in forms that are intended to facilitate reuse. The authors’ metamodeling approach to pattern-based refactoring incorporates the precise specification of design patterns and transformation rules. Their example uses the Abstract Factory pattern and a small UML model.

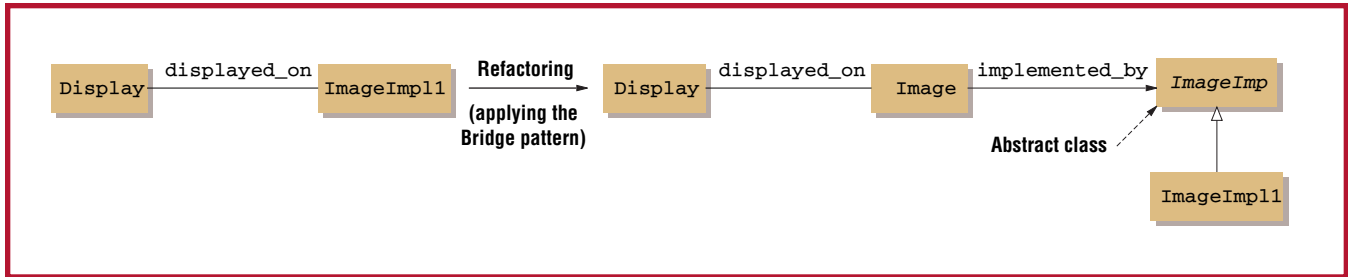


Figure 1. An example of pattern-based model refactoring.

how to separate an abstraction from its implementations so that you can vary the abstractions and implementations independently. The UML design model in Figure 1 has a `Display` class associated with an `ImageImpl1` class that defines a class of image implementations. This model is refactored so that the `Display` class is now associated with an `Image` class structure that allows image implementations linked to the display to be varied at runtime.

The refactoring shown in Figure 1 might proceed as follows:

1. Identify the design elements in the source model that will participate in the pattern solution. Modify them so that they can participate as the pattern specifies. In this case, we identify the `Display` class as the client (not shown in Figure 2) using the `Abstraction`, and the `ImageImpl1` class as a `ConcreteImp` participant.
2. Add new model elements as needed to realize the behavior specified by the pattern. Create two new classes representing image abstractions (the `Abstraction` participant in the pattern description) and implementation interfaces (the `Implementor` participant). The `Image` class plays the role of `Abstraction`, and the `ImageImp` class plays the role of `Implementor` in the refactored model. Create an association (`implemented_by`) between the `Abstraction` and `Implementor` classes, as the pattern requires. The design class `ImageImpl1` becomes a subclass of `ImageImp` in the refactored model—that is, it plays the role of an `Implementor` subclass.

We can do this refactoring manually with relatively little effort on small models. However, the process can become tedious and error-prone when we apply it to larger models. Tools that automate the process of applying pattern-based transformations help reduce the effort of consistently and correctly realizing

patterns across a design. Such tools must be pattern-aware—that is, the tools must embed codified knowledge of patterns that the system’s users can access. Pattern-aware tools present patterns as abstraction units that developers can use to construct models.

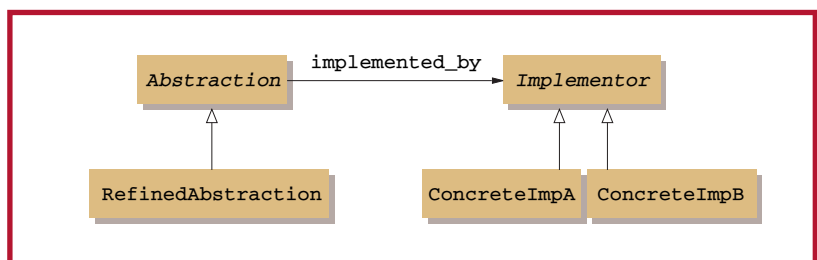
Popular descriptive forms of design patterns, although effective at communicating design experiences to software developers, are too informal to use as a basis for developing pattern-aware modeling tools. We need precise forms of patterns, called *pattern specifications*, to codify them in tools. Transformation rules also must be precisely specified if these tools are to support them. We use a metamodeling approach to specify both patterns and transformation rules. Because we probably can’t capture all the variations of a pattern in one specification, we can associate a single pattern with many pattern specifications, each representing a pattern variation. Each pattern specification in turn can be associated with multiple transformation specifications, where each specification characterizes a different way of embedding a pattern into a design.

Our approach

To support pattern-based UML model refactoring, a pattern specification should include

- *Problem specification*: a precise specification of the family of UML design problems that the pattern addresses
- *Solution specification*: a precise specifica-

Figure 2. A diagram of the Bridge pattern.



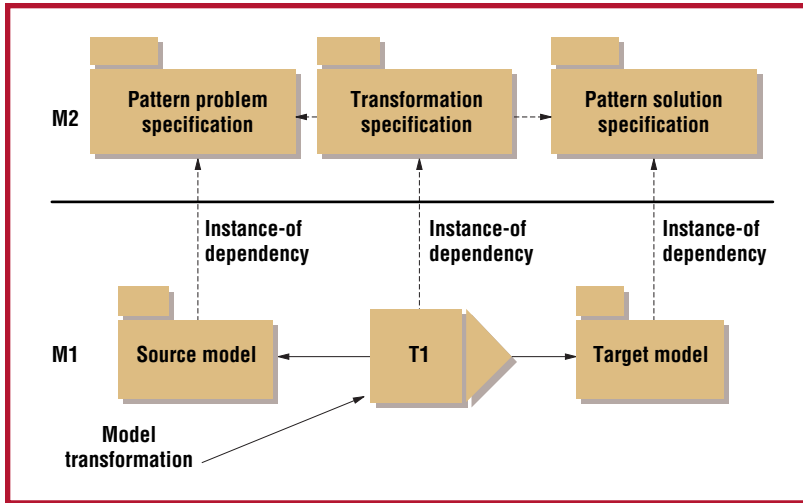


Figure 3. Metamodeling approach to specifying transformations.

tion of the UML designs representing solutions of the pattern

- *Transformation specification*: a specification of problem-to-solution transformations

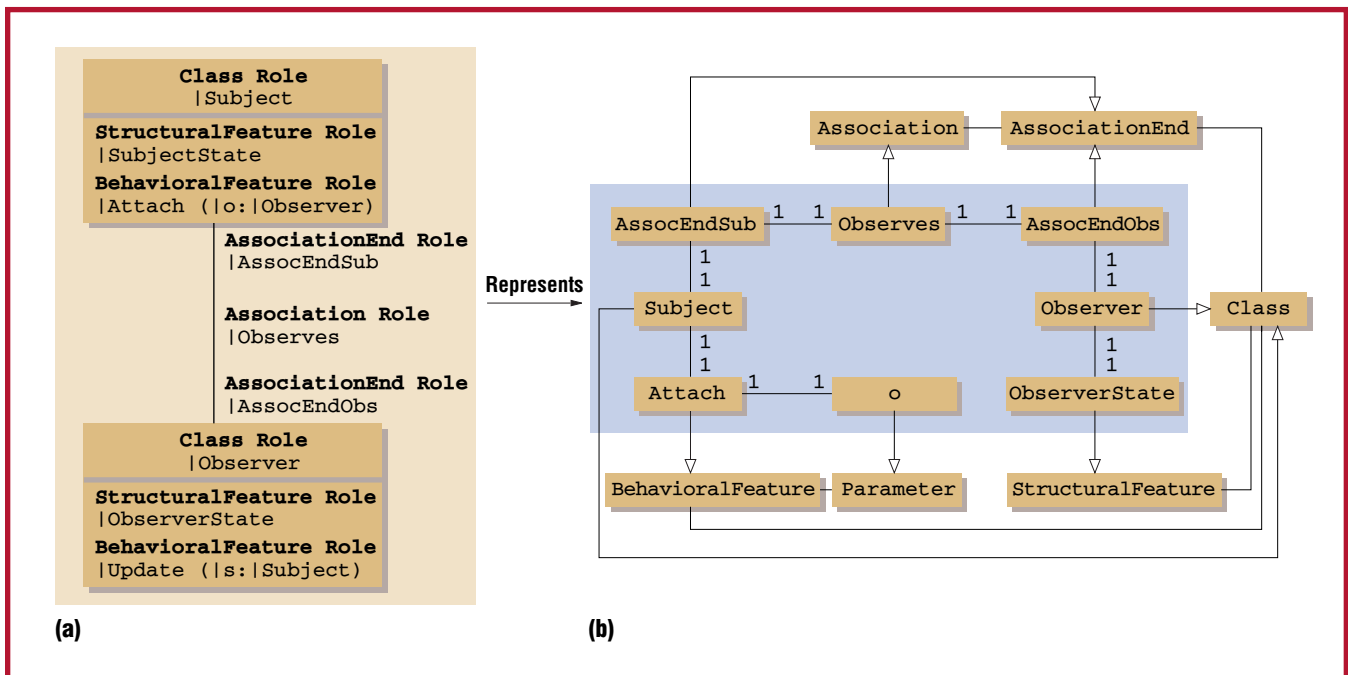
Figure 3 presents an overview of our metamodeling approach to specifying transformations. The M2 level is an extension of the UML metamodel level that supports the definition of a transformation language. The M1 level is an extension of the UML model level that supports representation of model transformations. A model transformation at the M1 level transforms a source model to a target model. The

M1 model transformations are characterized by a transformation specification at the M2 level that includes a pattern problem specification for the source models and a pattern solution specification for the target models. We can also view the transformation specification at M2 as defining a transformation language for the pattern, and the transformations at M1 as statements of the language. A UML model that conforms to a pattern problem or solution specification is said to be an instance of the specification. Similarly, a model transformation that conforms to a transformation specification is said to be an instance of the transformation specification.

Pattern problem and pattern solution specifications characterize a subset of UML models characterized by the UML metamodel, so we can view the specifications as defining UML metamodel specializations. UML tool developers can embed patterns by appropriately specializing their representations of the UML metamodel.

Figure 4 illustrates how we can represent the pattern solution specification for a simplified form of the Observer pattern as a specialized form of the UML metamodel. The Observer pattern specification characterizes UML class diagrams that model structural aspects of Observer pattern solutions. Each target class diagram characterized by the spec-

Figure 4. (a) A pattern solution specification for an Observer pattern; (b) the specialized UML metamodel.



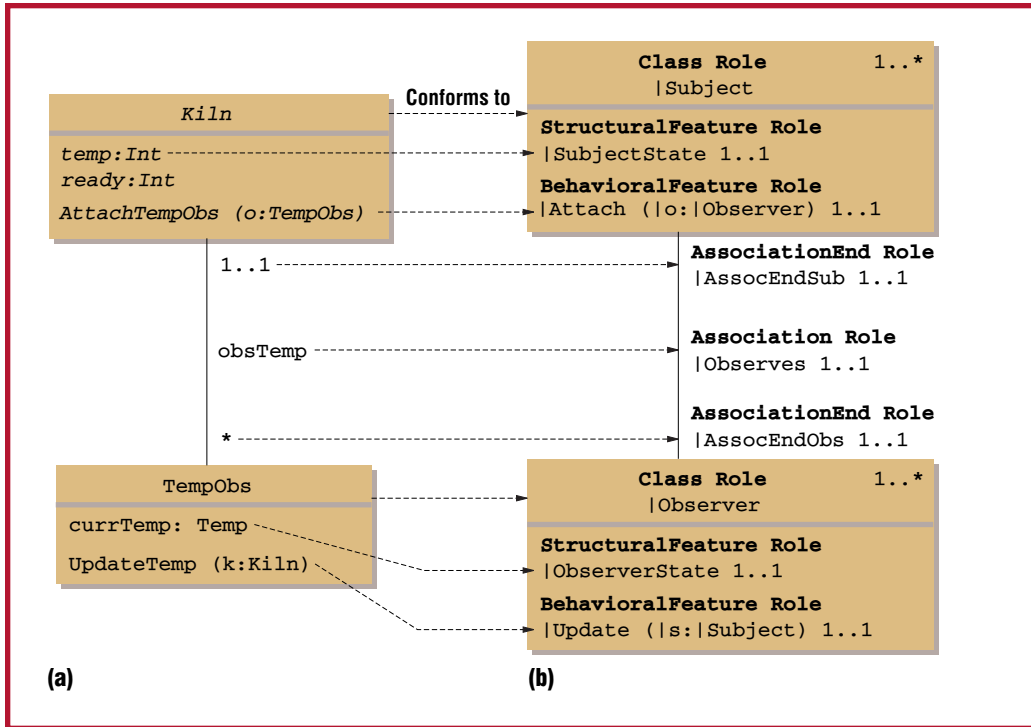


Figure 5. A realization of the Observer solution specification in Figure 4a: (a) Conforming class diagram; (b) Observer pattern specification.

ification consists of a **Subject** class (with an **Attach** operation and an attribute representing its state) that can be associated with an **Observer** class containing a state attribute and an update operation.

The specialized UML metamodel classes shown in the blue box in Figure 4b (this is only a partial view) define the family of UML class diagrams that represents solutions of the **Observer** pattern. The pattern specification in Figure 4a represents the specialized metamodel that more clearly reflects the structure of pattern solutions. This specification is expressed in terms of pattern roles that define participants in pattern solutions. Each role defines a specialization of a UML metamodel class. For example, **Subject** and **Observer** define specializations of the **Class** metamodel class named **Subject** and **Observer**, respectively, and the **Observer** role defines a specialization of the UML metamodel class **Association**. (Detailed descriptions of the pattern specification notation are available elsewhere.⁴⁻⁶) Figure 5 presents a simple class diagram that is an instance of the **Observer** pattern solution specification shown in Figure 4. Dashed arrows show a mapping between conforming model elements in Figure 5a and roles in Figure 5b. For example, the `AttachTempObs` operation plays the `Attach` role.

To refactor a design model using a selected

design pattern, first check the design model against the pattern problem specification to determine if you can apply the chosen design pattern to the source model. If so, develop the transformation steps using the transformation language defined by the transformation specification at the metamodel level. Then apply the transformations to the source model to produce a target model.

Example

We illustrate the refactoring process using the maze game model described by the GoF.¹ To simplify the presentation, we describe only the refactoring of class diagrams. Figure 6 shows the maze game class diagram. The **MazeGame** class is responsible for creating the different types of mazes (there are two, **BombedMaze** and **EnchantedMaze**) and their parts. A maze consists of rooms with doors and walls. If a new type of maze or maze part were added, **MazeGame** would have to change significantly. Incorporating the **Abstract Factory** pattern into the design results in a more flexible design, in which the maze creation aspects are localized in factories that the **MazeGame** class can access.

Figure 7 shows the **Abstract Factory** problem specification for the pattern used to refactor the maze game design (we omitted the



Figure 8. An Abstract Factory pattern solution specification.

the appropriate factories. The associations between the primary product classes and their subparts are used to determine where the `Create` operations should reside. For example, `Create` operations for `RoomWithBomb`, `Door`, and `BombedWall` objects (parts of `BombedMaze` objects) are placed in the `BombedMazeFactory`.

3. *Link the client to factories.* The pattern user can use associations or dependencies to link the client to the factories. In this case, we chose to use an association to link `MazeGame` to `MazeFactory`.

Figure 9 shows the result of the refactoring.

We are now providing tool support for our approach. The tool provides two interfaces, one for the pattern engineer to evolve and manipulate the tool's representation of the UML metamodel, and the other for the application engineer to create, manipulate, and evolve UML models using patterns.

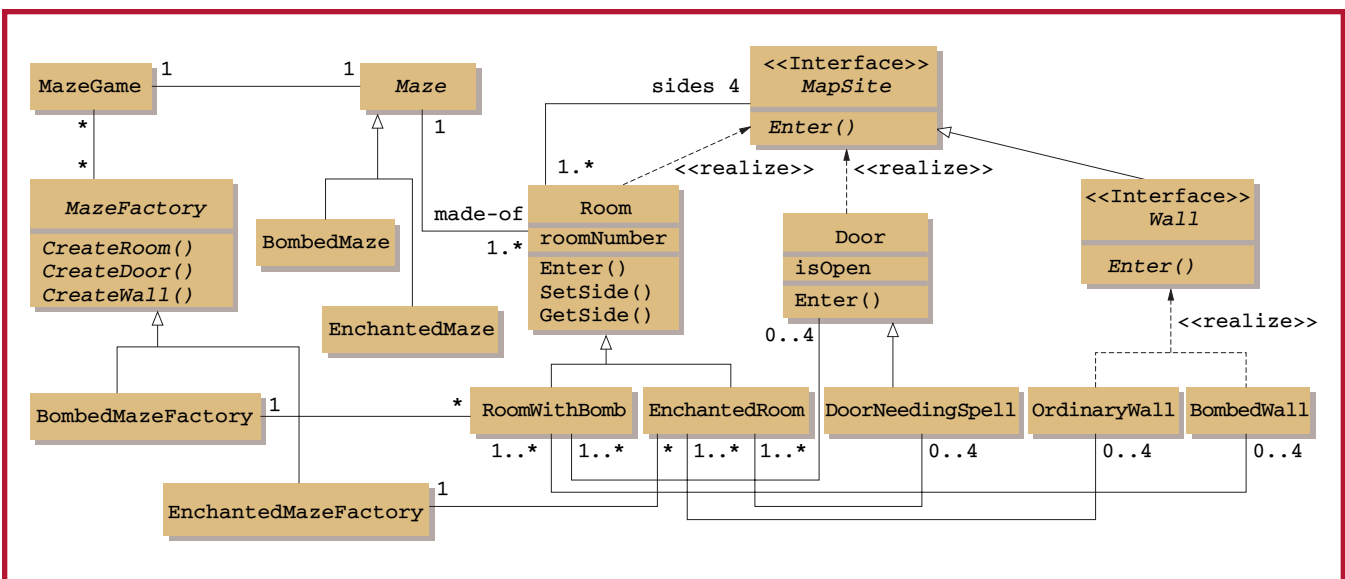
There is significant potential for further research. Two major concerns must be addressed

when developing model refactoring techniques:

- *Preserving properties.* A refactoring preserves functional properties. When defining a refactoring, we must specify the properties that are unchanged when the transformations are completed. In our approach, these properties are defined as part of the problem and solution specifications (that is, the invariant properties are those that are common to both specifications). We are developing tools that can help establish conformance of models to the specifications.
- *Identifying and resolving pattern interference.* Composing two or more design patterns could lead to conflicts that must be resolved, and this process could involve trade-off analysis. Functional validation for models that contain composed patterns is necessary.

Clearly, it is possible to use well-defined transformations to support systematic application of patterns. Tools that support pattern-based model transformations provide developers with abstraction building blocks that go beyond those provided by general modeling

Figure 9. The maze game refactored with the Abstract Factory pattern.



About the Authors



Robert France is an associate professor at Colorado State University. His research interests are software engineering, object-oriented analysis and design methods, aspect-oriented modeling, and development of complex systems. He was a member of the OMG's task force for UML 1.3 and 1.4 and is involved with evaluating UML 2.0 submissions. He is also coeditor in chief of the *Journal of Software and System Modeling*. He received his PhD in computer science from Massey University in Palmerston North, New Zealand. Contact him at the Computer Science Dept., Colorado State Univ., Fort Collins, CO 80523; france@cs.colostate.edu.


Sudipto Ghosh is an assistant professor in the Computer Science Department at Colorado State University, Fort Collins. His teaching and research interests include modeling, designing and testing of object-oriented software, distributed object systems, and aspect-oriented and component-based software development. He received his PhD in computer science from Purdue University. He is a member of the IEEE Computer Society and the ACM. Contact him at the Computer Science Dept., Colorado State Univ., Fort Collins, CO 80523; ghosh@cs.colostate.edu.



Eunjee Song is a PhD candidate and a member of the Software Assurance Laboratory in the Computer Science Department at Colorado State University. Her research interests include software design and specification, model refactoring, and design patterns. She received an MS in computer science from Colorado State University and is a student member of the IEEE. Contact her at the Computer Science Dept., Colorado State Univ., Fort Collins, CO 80523; song@cs.colostate.edu.

Dae-Kyoo Kim is a PhD candidate in computer science at Colorado State University. His research interests are pattern formalization, model refactoring, aspect-oriented modeling, security, and component-based software development. He received his MS in computer science from Western Michigan University. He is a member of the IEEE and the International Systems Security Association. Contact him at the Computer Science Dept., Colorado State Univ., Fort Collins, CO 80523; dkkim@cs.colostate.edu.



languages. Using domain-specific patterns in particular can result in order-of-magnitude savings in development cost and time. We have used this pattern specification technique to define not only general-purpose design patterns but also domain-specific patterns. Our current work focuses on extending the transformation approach to support the application of product-line architectures. 

References

1. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
2. W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.
3. *The Unified Modeling Language: Version 1.4*, tech. report formal/2001-09-67, Object Management Group, 2001.
4. R. France et al., "Using Roles to Characterize Model Families," *Practical Foundations of Business and System Specifications*, Haim Kilov, ed., Kluwer Academic Publishers, 2002.
5. R.B. France, D.K. Kim, and E. Song, *Patterns as Precise Characterizations of Designs*, tech. report 02-101, Computer Science Dept., Colorado State Univ., 2002.
6. D. Kim et al., "Using Role-Based Modeling Language (RBML) as Precise Characterizations of Model Families," *Proc. 8th IEEE Int'l Conf. Eng. of Complex Computer Systems (ICECCS 2002)*, IEEE CS Press, 2002.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.

SOFTWARE ENGINEERING GLOSSARY

Reviews

review: A process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users, customers, or other interested parties for comment or approval. Types include joint acquirer-supplier reviews (sometime called milestone reviews), management reviews, technical reviews, peer reviews (includes inspections and walkthroughs), and audits. [IEEE Std 1028-1997]

architectural design review (ADR): Evaluation of the technical adequacies of the software architectural design as depicted in the software design descriptions. Sometimes synonymous with preliminary design review.

formal qualification review (FQR): The test, inspection, or analytical process by which a group of configuration items comprising a system are verified to have met specific contractual performance requirements. [ANSI/IEEE Std. 610.12 1990]

joint acquirer-supplier review: Evaluation of an activity's status and products, conducted jointly between the system's acquirer (representing the user) and the supplier (representing the developer), at the completion of a major software development milestone. Usually thought of as visibility and status mechanisms, but its gate-keeping and action item aspects are also powerful control mechanisms.

system requirements review (SRR): A system milestone review that ascertains the adequacy of the developer's efforts in defining system requirements. Conducted when a significant portion of the system's functional and performance requirements have been established, normally in the definition phase.

technical review: Evaluation of products or services under consideration to provide evidence that they are complete, comply with standards and specifications, and are ready for the next activity.

test readiness review (TRR): Determination that software test procedures are complete and the developer is prepared for formal software performance testing; results of informal testing also are reviewed. [Military Std. 1521B-1985]