A Metamodeling Approach to Model Refactoring

Sheena R. Judson[1], Doris L. Carver[1], and Robert France[2]

[1]Department of Computer Science, Louisiana State University
Baton Rouge, Louisiana USA
sheena.r.judson@lmco.com, carver@bit.csc.lsu.edu
[2]Department of Computer Science, Colorado State University
Fort Collins, Colorado USA
france@cs.colostate.edu

Abstract. The Model Driven Architecture (MDA) initiative formulated by the Object Management Group (OMG) provides a framework for a set of standards supporting a model-centric style of development. MDA is intended to support the use of models as the primary artifacts of software development. It represents an approach for delivering software-intensive systems through the transformation of models – primarily expressed in the UML – to executable components and applications. Techniques and technologies that support rigorous definition and application of model transformations are required to realize the MDA vision. In this paper we describe a technique that supports rigorous definition and use of a type of transformation called *model refactoring*. The paper describes how families of model transformation can be defined at the UML metamodel level.

1 Introduction

The Model Driven Architecture (MDA) initiative of the Object Management Group (OMG) provides a framework for a set of standards supporting model-centric system development. The main goal of MDA is to elevate models to the forefront of software development, that is, models are the primary artifacts of system development in an MDA approach. Well-defined model transformations that support rigorous model evolution, refinement, and realization in code are considered to be key elements of an MDA approach.

A model transformation can have *vertical* or *horizontal* dimensions [1]. *Vertical transformations* occur when a source model is transformed into a target model at a different level of abstraction. Refining a model and realizing a model in a target programming language are examples of vertical transformations. A *horizontal transformation* occurs when a source model is transformed to a target model that is at the same level of abstraction as the source model. Horizontal transformations are carried out to support model evolution. Two examples of model evolution are (1) adding new features to a design and (2) restructuring a design to enhance existing features. As in code evolution, three types of model evolution can be identified:

- Corrective evolution is concerned with correcting errors in design,

- Adaptive evolution is concerned with modifying a design to accommodate changes in requirements, and
- Perfective evolution is concerned with modifying a design to enhance existing features.

This paper focuses on transformations that support perfective model evolution. This type of transformation is referred to as *model refactoring.* Ad-hoc approaches to UML model refactoring can lead to convoluted designs that are difficult to evolve, analyze, and refine. Controlled model refactoring can be accomplished by developing metamodels for refactoring. The metamodels can be used to constrain how the refactoring is carried out at the UML model level and can act as points against which model-level refactoring can be checked for conformance. In this paper we describe an approach to defining metamodels for refactoring and illustrate its use by defining a metamodel for a pattern-based model refactoring.

Section 2 provides a brief overview of pattern-based model refactoring. In Section 3 we describe the parts of UML metamodel that are pertinent to the descriptions given in the paper. In Section 4 we describe our approach to metamodeling refactoring and give an example of its application using the Abstract Factory pattern. An overview of other work on model transformation is given in Section 5. We conclude in Section 6 with an overview of our plans to further evolve this work.

2 Pattern-Based Model Refactoring

A design pattern describes a family of solutions for a class of recurring design problems [2, 3]. Incorporating a pattern into a source model to produce a target model is called *pattern-based model refactoring*. In this paper we focus on metamodeling of pattern-based refactoring. A pattern-based model refactoring is carried out when it is determined that a pattern can help improve how a design accomplishes its objectives. Fig. 1 shows an example of model refactoring. In the example, a model in which a *Display* class is associated with a specific implementation class (*ImagImpl1)* is transformed using the Bridge pattern [3] to a design model in which the *Display* class is associated with a class structure that allows the image implementation to be varied.

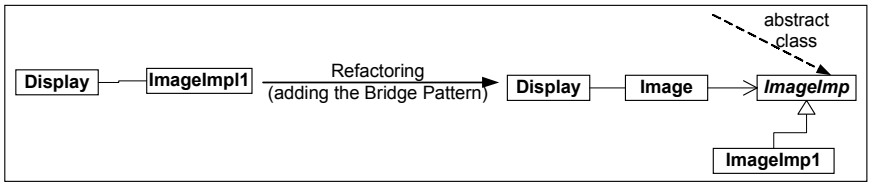

Fig. 1. Modeling Refactoring.

The Bridge pattern refactoring shown in Fig. 1 is an example of a model-level refactoring. A more generic description of the Bridge pattern transformation that captures common aspects of the model-level transformations is possible. The approach described in this paper allows one to define one or more metamodels for

pattern-based transformations, where each metamodel characterizes a family of model-level transformations. More than one metamodel may be needed to define refactoring based on a single pattern when transformation variations cannot be conveniently addressed in a single metamodel.

The metamodeling approach described in this paper defines families of transformations in terms of classes of model elements that are created and deleted during transformations. The classes of model elements are distinguished in the metamodel as subclasses of UML metamodel classes. As an example consider the Bridge pattern shown in Fig. 1. The source model can be characterized as a structure consisting of a client class (e.g., *Display*) associated with the implementation of a product class (e.g., *ImagImpl1*). This structure can be modeled in the UML metamodel by defining a subclass of *Class* whose instances are client classes, another subclass of *Class* whose instances are product implementation classes, and a subclass of *Association* whose instances represent associations between client and product implementation classes. A generic description of the refactoring can then be expressed by specifying the effect of the transformation on the specialized UML metamodel. For the Bridge pattern refactoring, the change involves (1) adding two new specializations of *Class*, one representing product abstraction classes (e.g., *Image*), and the other representing product implementation abstraction classes (e.g., *ImageImp*), (2) removing the specialized *Association* class representing associations between client classes and product implementations, (3) adding a specialization of the *Association* class that represents associations between product abstraction and product implementation abstraction classes, and (4) adding a specialization of the *Generalization* class representing the generalization relationship between product implementation abstraction and product implementation classes. A more detailed description of how this can be accomplished for a more complex pattern-based model refactoring is given in Section 4.

3 The UML Metamodel

The UML metamodel characterizes the set of valid models. A fragment of the UML metamodel class diagram that is pertinent to the work described in this paper is shown in Fig. 2. The metamodel used in this work is based on the UML 2.0 submission from the U2 Partners [4].

The metamodel shows (1) the relationship between UML classifiers (instances of *Classifier*) and their properties (e.g., attributes) and operations, (2) the relationship between classifiers and associations, and (3) the relationship between operations and the actions they define. There are many types of actions, one of which is the *CreateObjectAction*. Instances of this class are actions that create instances of the *Classifier* they are associated with. The following is a brief description of some of the classes shown in Fig. 2 .

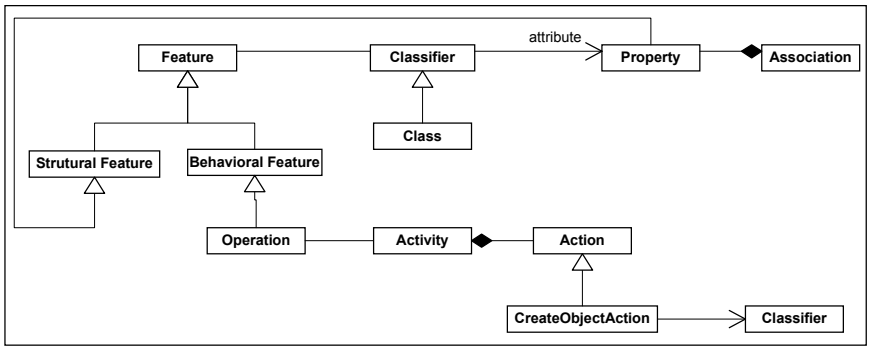

Fig. 2. A Fragment of the UML 2.0 Metamodel.

- An instance of *Property* is a structural feature of a classifier that characterizes instances of the classifier that represents a declared state of one or more instances in terms of a named relationship to a value or values. When a *class owns a property* it represents an attribute. In this case it relates an instance of the class to a value or set of values of the type of the attribute. When an *association owns a property* it represents an end of the association. In this case the type of property is the type of the classifier at the end of the association.
- An instance of *Feature* can be either a property or behavior. An instance of *Behavioral Feature* describes behavioral aspects of one or more classifiers. An instance of *Structural Feature* is a typed feature of a classifier that declares a structural aspect of the instances of a classifier.
- An instance of *Operation* is a specification of a service. An operation can consist of activities.
- An instance of *Activity* is a specification of behavior. An activity consist of actions.
- An instance of *Action* is the fundamental unit of a behavioral specification. It takes a set of inputs and converts them into a set of outputs.
- An instance of *CreateObjectAction* is an action that creates an object that is an instance of a classifier.

4 Approach

An overview of the transformation approach described in this paper is shown in Fig. 3. The M2' level is an extension of the UML metamodel level that supports the specification of transformation families. The M1' level is an extension of the UML model level that supports representation of model transformations. A model transformation at the M1' level (model level) takes a source model and transforms it to a target model. The M1' model transformations are characterized by a transformation pattern at the M2' level that includes a source pattern that characterizes source models and a target pattern that characterizes target models. A

UML model that conforms to a source or target pattern is said to be an *instance* of the pattern. Similarly, a model transformation that conforms to a transformation pattern is said to be an instance of the transformation pattern.

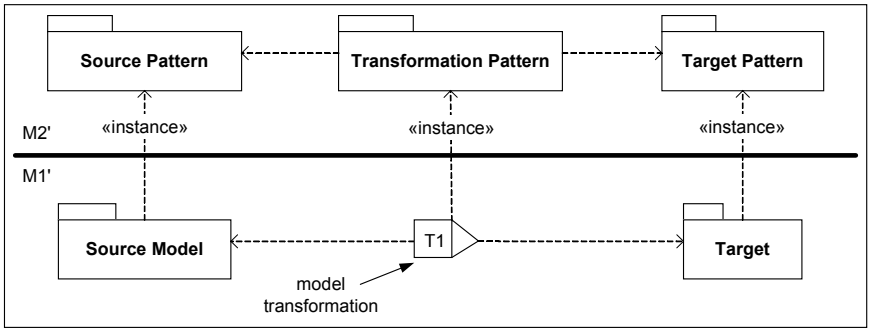

Fig. 3. Transformation Overview.

We illustrate our approach to model refactoring using the MazeGame class diagram shown in Fig. 4 [adapted from 3]. In the model, the class MazeGame creates two types of mazes: BombedMazes and EnchantedMazes. A BombedMaze consists of rooms with bombs (instances of RoomWithBomb), and an EnchantedMaze consists of enchanted rooms (instances of EnchantedRoom). A room with a bomb consists of doors (instances of Door) and bombed walls (instances of BombedWall) and an enchanted room consists of doors that need spells (instances of DoorNeedingSpell) and ordinary walls. The problem with this design, as pointed out in [3], is that the creation of mazes is hardcoded into the client (MazeGame), and thus introducing a new maze type requires modification of the client. The Abstract Factory (AF) pattern defines a generic solution that can be used to make a client independent of how the products it manipulates are created.

Applying the AF pattern to the MazeGame class diagram results in a new class diagram where maze creation operations are removed from the client and operations for creating mazes and their parts are placed in classes called factories, as shown in Fig. 5. The primary responsibilities of the factories are the creation of mazes. A client uses the factories to create mazes. This separation of concerns allows changes to be made to maze creation without impacting the client.

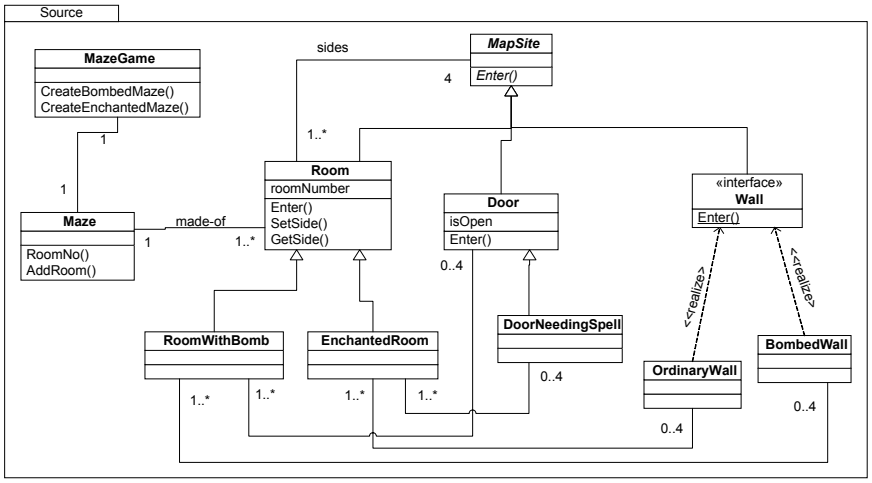

Fig. 4. Maze Game Source Model.

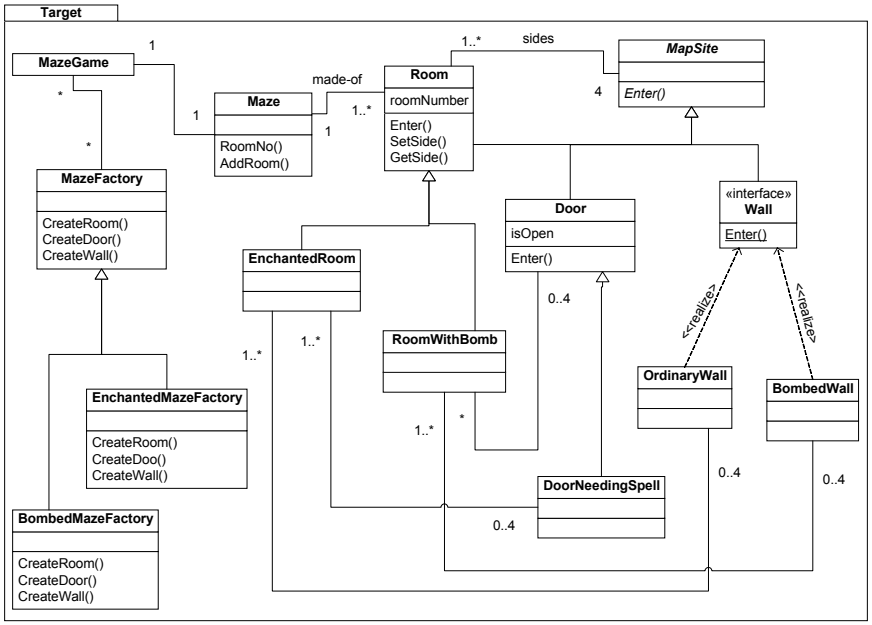

Fig. 5. Maze Game Target Model.

4.1 Transformation Patterns: Defining Transformations at the Metamodel Level

A transformation pattern for the AF pattern is shown in Fig. 6. The pattern consists of three parts:

- The *Source Pattern* defines the set of source models to which transformations specified by the transformation pattern can be applied.
- The *Transformation Schema* identifies the model structures that are created and deleted by the transformation. The classes enclosed in the dashed box represent classes of model elements that are added and classes with an **X** represent classes that are deleted by the transformations.
- The *Transformation Constraint* states the relationship that must hold between target and source model elements.

The Source Pattern, given in Fig. 6(a), characterizes source models that consist of (1) client classes (e.g., MazeGame) associated with primary products (e.g., Maze), (2) product whole-part structures (e.g., Maze-Room and RoomWithBomb-Door relationships), and (3) create operations in client classes that create products. Each of the classes shown in the source pattern are specializations (subclasses) of classes in the UML metamodel. The specialization relationships are described below. They are not shown in the diagram to reduce diagram clutter.

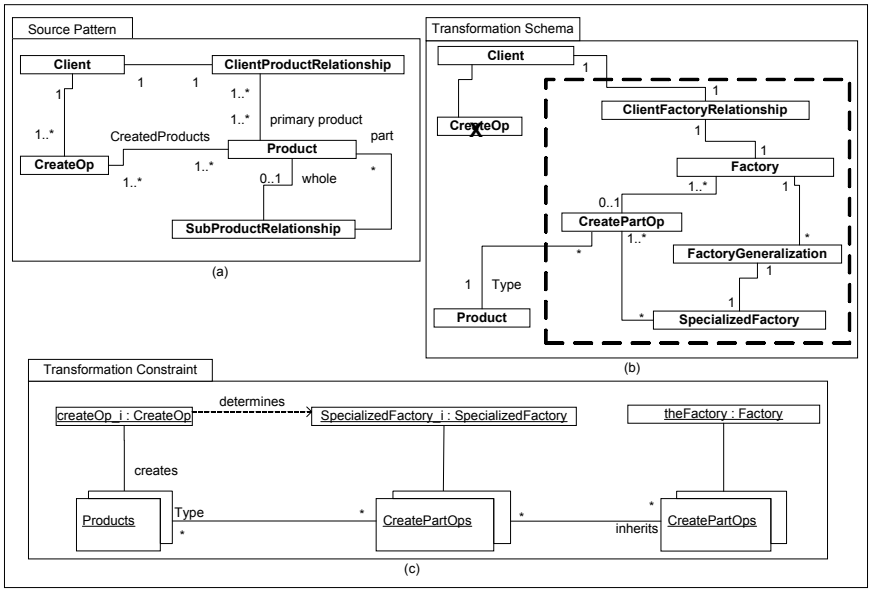

Fig. 6. Abstract Factory Transformation Pattern.

- *Client* is a subclass of *Class* in the UML metamodel whose instances are classes representing clients in the application domain.

- *CreateOp* is a subclass of *Operation* in the UML metamodel. Instances of this class are operations associated with exactly one instance of *Client*. A *Client* instance (a client class) must consist of at least one operation that is an instance of *CreateOp*.
- *Product* is a subclass of *Class* whose instances are classes representing products in the application domain.
- *ClientProductRelationship* is a subclass of *Association* whose instances are associations between client classes and product classes designated as primary product classes.
- *SubProductRelationship* is a subclass of *Association* whose instances are associations between product classes. The associations represent whole-part relationships between product classes.

The *CreatedProducts* association between *CreateOp* and *Product* is a calculated relationship that is needed by the AF transformation. An AF transformation needs information about the type of products created by each create operation in a client in order to assign create operations to the appropriate factories. For example, an AF transformation will need to know what products are created by *CreateBombedMaze* operation in the source *MazeGame* model in order to determine what products *BombedMaze* factory should create (later we will see how one can specify that a specialized factory is created for each create operation in a source client's class). The *CreatedProducts* association link each client's create operation to the products it creates. The association is defined by the following OCL constraint:

```
Context CreateOp
def: CreatedProducts : Set(Product) =
self.activity.action ->
select(self.oclIsTypeOf(CreateObjectAction)) ->
select(classifier.oclIsTypeOf(Product))
```

The model shown in Fig. 4 conforms to the source pattern. The conforming parts of the model are given in the object diagram shown in Fig. 7.

The *Transformation Schema*, shown in Fig. 6(b), indicates the new classes of model elements that are introduced by the transformation and the existing classes of model elements that are removed by the transformation. The classes shown in the schema are all specializations of UML metamodel classes (not shown in Fig. 6(b)). The schema indicates that the AF pattern transformation introduces factory classes (instances of *Factory*), specializations of factory classes (instances of *SpecializedFactory*), create operations associated with the factory classes (instances of *CreatePartOp*), and associations between factory and client classes. The schema also indicates that the create operations in client classes are removed.

The transformation schema determines the basic structure of the target model. The *Transformation Constraint*, given in Fig. 6(c), further constrains the basic structure by defining relationships that must hold between elements in the target model and between source model elements and target model elements. For example, in an AF pattern transformation a unique specialized factory must be created for each create operation (instance of *CreateOp*) in the source client. In the source MazeGame model there are two such operations (*CreateBombedMaze*, *CreateEnchantedFactory*), and

thus two factory classes must be created. This constraint is expressed by the *determines* dependency, shown in Fig. 6(c), between unique instances of *CreateOp* and unique instances of *SpecializedFactory*.

The create actions defined by the instances of *CreateOp* become operations in the specialized factories corresponding to the *CreateOp* operations in an AF transformation. This means that for each product created by an instance of *CreateOp* (i.e., each product in the set determined by the calculated relationship *CreatedProducts*) there must exist an instance of *CreatePartOp* that creates the product in the specialized factory corresponding to the *CreateOp* instance. This constraint is defined by the set of links between the sets *Products* and *CreatePartOps*. The set of links between the sets of *CreatePartOps* in specialized factories and the factory indicates that the factory class (the root generalization) consists of create operations that are inherited by the specialized factories.

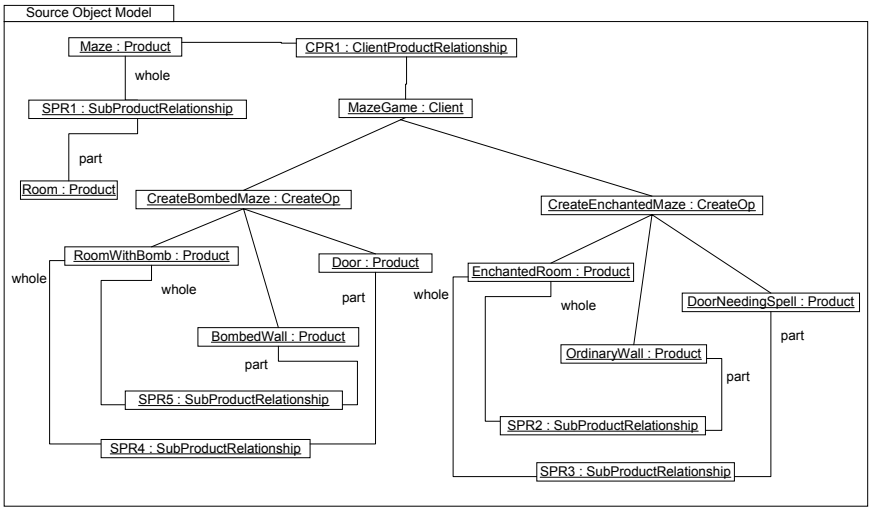

Fig. 7. Source MazeGame Object Model.

4.2 Transforming Models using Transformation Patterns

The transformation pattern characterizes a family of model-level transformations. In this subsection, we define a procedural transformation for the MazeGame model that conforms to the transformation pattern described in the previous subsection. We use object diagrams to illustrate the transformation procedure. The parts that are added are enclosed in dashed boxes. Deleted parts are marked by **X**'s.

The following is the transformation procedure defined for the MazeGame source model:

1. Create a *Factory* and associate it with a *Client*. As shown in Fig. 8, the *MazeFactory* is created as an instance of *Factory*. The diagram shows that a new instance of *ClientFactoryRelationship* (*M-to-F*) and a new instance of *Factory* (*MazeFactory*) are created.

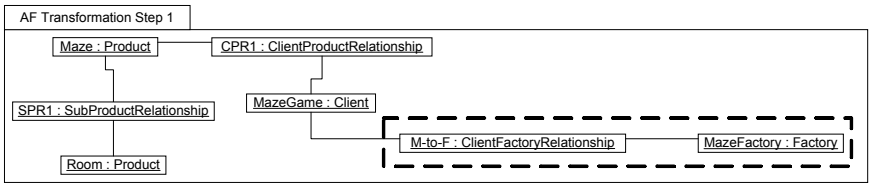

Fig. 8. MazeGame Transformation Step 1.

2. For each create operation in the client, create a specialized factory. Relate each specialized factory to the factory using a generalization relationship. As shown in Fig. 9, two specialized factories (*BombedMazeFactory* and *EnchantedMazeFactory*) are created. *CreateBombedMaze* determines the creation of *BombMazeFactory* and *CreateEnchantedMaze* determines the creation of *EnchantedMazeFactory* as required by the Transformation Constraint shown in Fig. 6(c).

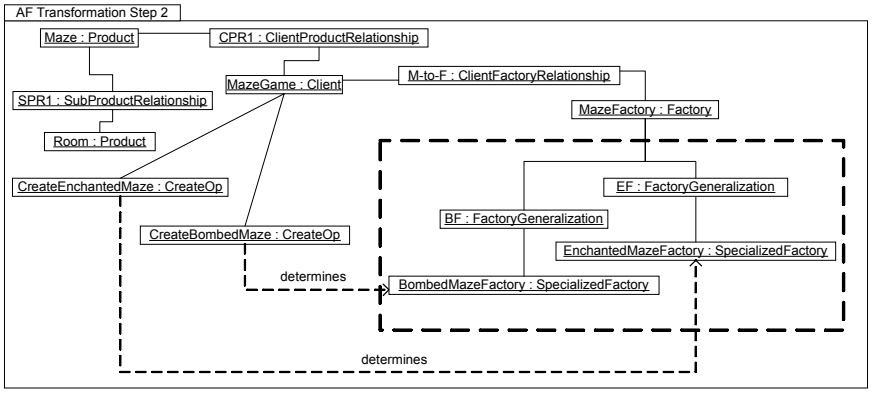

Fig. 9. MazeGame Transformation Step 2

3. For each specialized factory, include a create operation for each product that is created by the corresponding client's create operation. In the diagram shown in Fig. 10, a create operation (an instance of *CreatePartOp*) is added for each product created by the *CreateBombedMaze* operation. For example, a *CreateRoomB* operation is added to the *BombedMazeFactory* and linked to the *RoomWithBomb* class. Similarly, create operations are added to the *EnchantedMazeFactory* class. The *EnchantedMazeFactory* structure is not shown in Fig. 10 to reduce clutter.

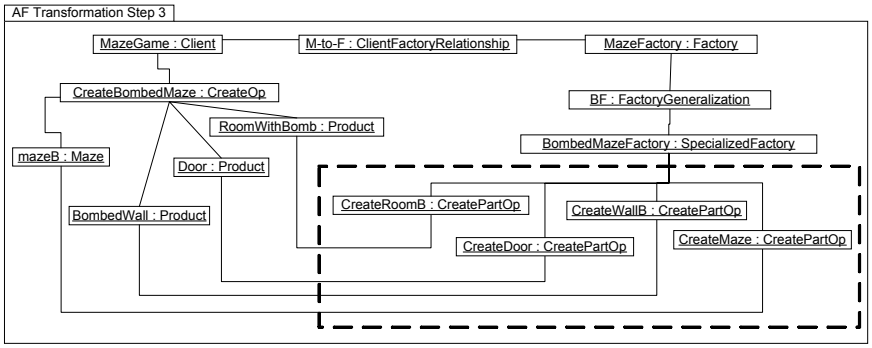

Fig. 10. MazeGame Transformation Step 3.

4. Remove the create operations from the client. All create operations in the *MazeGame* are deleted from the model. Fig. 11 shows the object diagram produced by this step.

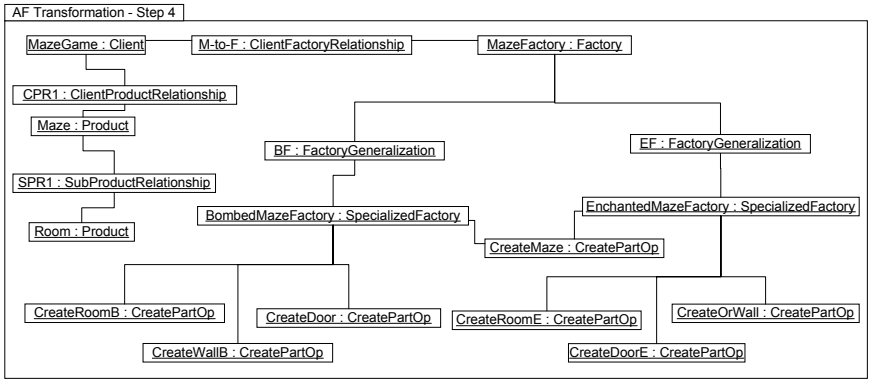

Fig. 11. MazeGame Transformation Step 4.

5. Add create operations to the factory class. As shown in Fig. 12, a create operation (an instance of *CreatePartOp*) is added for each product created by the *CreateBombedMaze* and *CreateEnchatedMaze* operation. For example, a *CreateMaze*, *CreateRoom*, *CreateDoor* and *CreateWall* operations are added and linked to the *MazeFactory* class.

The result of applying the above procedure to the MazeGame source model is the target model shown in Fig. 5. The transformation steps outlined above adhered to the Transformation Schema and the Transformation Constraint of the AF transformation pattern, and thus the result is a conforming target model. While the above description of the transformation is informal, it should be evident that the steps can be formalized in a transformation language supporting creation and deletion of model elements.

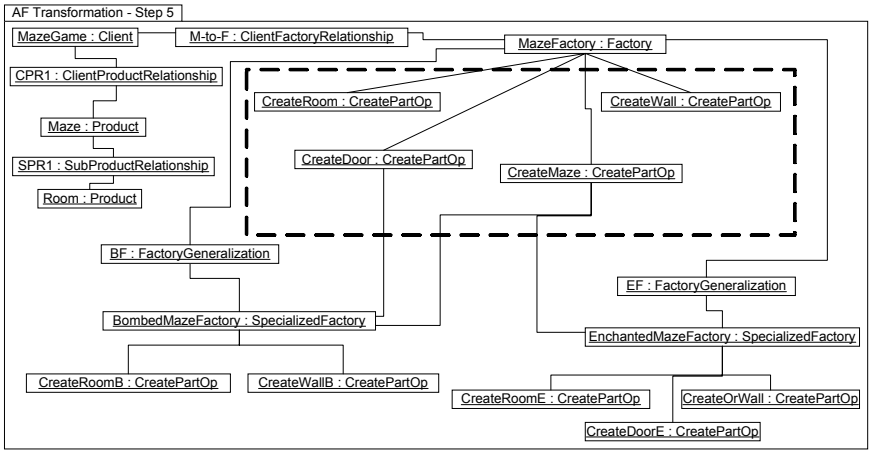

Fig. 12. MazeGame Transformation Step 5.

5 Related Work

There has been considerable work on modeling of code-level refactoring. Tom Mens et al. [5, 6] use graphs to represent the aspects of source code that should be preserved by a refactoring, and express transformations as graph rewriting rules. PROgramming Graph REwriting System, PROGRES [7, 8], is a visual language and environment supporting the design of graph structures and implementation of graph manipulating tools using a graph rewriting systems. PROGRES is not suitable for specifying model transformation because [9]: (1) PROGRES structures are expressive but not as expressive or as widely used as UML class diagrams with respect to the specification of integrity constraints, (2) PROGRES deals with transformations on a single graph and do not produce a new graph that conforms to a different schema/metamodel, and (3) PROGRES is mainly a programming language with graphical productions and thus not at the level of abstraction desired for specifying model transformation.

Model Integrated Computing (MIC) [9, 10] is a software and system development approach that advocates the use of domain specific models to represent relevant aspects of a system. The technique is based on graph transformations, where UML class diagrams represent graph grammars of the input and the output of the transformations and the transformations are represented as explicit sequenced elementary rewriting operations. Their work relies on the use of a pattern specification language that uses UML concepts and on pattern matching to effect transformations.

Our work differs from the above work in that we focus on model refactoring and on characterizing families of transformations rather than defining mechanism for effecting transformations.

6 Conclusion

A goal of our work is to develop support for modeling transformations at the metamodel level to facilitate controlled evolution of models. In this paper, we describe a technique for describing transformations at the metamodel level as Transformation Patterns and illustrate its use by using it to develop a Transformation Pattern for an Abstract Factory pattern refactoring.

We are currently developing a formal basis for model-level transformations that can be used to support rigorous conformance checking against Transformation Patterns. We are also investigating the applicability of OCL 2.0 to express constraints that cannot be conveniently expressed in the diagrams associated with Transformation Patterns.

References

[1] France, Robert and James Bieman. Multi-view Software Evolution: A UML-based Framework for Evolving Object-Oriented Software. In Proceedings International Conference on Software Maintenance (ICSM 2001), November 2001.

[2] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. Pattern-Oriented Software Architecture – A System of Patterns. Wiley and Sons, 1996.

[3] Gamma, E., Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[4] The Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.0, 2nd revision, (OMG doc# ad/03-03-02). OMG, http://www.omg.org, March 2003.

[5] Mens, Tom, Serge Demeyer, and Dirk Janssens. A Graph Rewriting for Object-oriented Software Refactoring. FWO-WOG, January 2002.

[6] Mens, Tom, Serge Demeyer, and Dirk Janssens. Formalizing Behavior Preserving Program Transformations. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, Proceedings of the International Conference on Graph Transformation (ICGT 2002), volume 2505 of Lecture Notes in Computer Science, pages 286-301. Springer-Verlag, 2002.

[7] Schürr, Andy. Developing Graphical (Software Engineering) Tools with PROGRES. In Proceedings of the International Conference on Software Engineering (ICSE ’97). 1997.

[8] Bardohl R., M. Minas, A. Schurr, and G. Taentzer. Application of graph transformation to visual languages. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, Handbook of Graph Grammars and Computing by Graph Transformation, volume II: Applications, Languages and Tools, pages 105--180. World Scientific, 1999.

[9] Agrawal A., Karsai G., Shi F.: A UML-based Graph Transformation Approach for Implementing Domain-Specific Model Transformations, International Journal on Software and Systems Modeling, (Submitted), 2003.

[10] Agrawal A., Karsai G., Ledeczi A.: A UML-based Graph Transformation Approach for Implementing Domain-Specific Generators, Second International Conference on Generative Programming and Component Engineering, (GPCE '03) Submitted, Erfurt, Germany, September 22, 2003.