Using Roles for Pattern-Based Model Refactoring

Eunjee Song, Robert B. France, Dae-Kyoo Kim, and Sudipto Ghosh

Colorado State University, Fort Collins, CO 80523, USA

Abstract. Design patterns capture some of the best software development experiences in forms that are intended to facilitate reuse. We treat a design pattern as a characterization of a family of solutions, where the solutions are expressed as UML (Unified Modeling Language) design models. We present a new notation that we call Role Models to characterize pattern solutions, and describe how they can be used to support systematic pattern-based model refactoring.

Keywords: Design patterns, model refactoring, reuse, object-oriented models, role models, UML

1 Introduction

Design patterns (e.g., see [2, 8, 11]) are intended to capture high-quality design experiences in a form that facilitates their reuse. Conceptually, a typical design pattern consists of two major parts, the *usage context* and the *pattern specification*. The *usage context* contains information that is used to determine the appropriateness of a pattern to a specific problem (e.g., see the Intent, Motivation, Applicability, Consequences, Implementation, and Sample Code parts of the GoF ("Gang of Four") patterns [8]). The *pattern specification* contains a description of behavioral and structural aspects of the solutions it characterizes (e.g., see the Structure, Participants, and Collaborations parts of the GoF pattern descriptions [8]).

In this paper we present a notation for precisely expressing pattern specifications. A design is said to *realize* a design pattern if the design possesses the properties specified in the pattern. Such a design is called a realization of the pattern. Pattern realizations are design models expressed in the Unified Modeling Language (UML). Structural and behavioral properties of a pattern are specified using our *Role Models*. A Role Model characterizes a family of UML design models.

Precise specification of patterns paves the way for the development of systematic pattern-based model refactoring techniques. *Model refactoring* occurs when a design model is transformed to improve specific qualities (e.g., flexibility). Model refactoring may be done for various reasons, such as (1) to meet design goals, (2) to address deficiencies uncovered by design analyses, and (3) to explore alternative designs. Incorporating a pattern into a design to improve design quality attributes is called *pattern-based model refactoring*. Precise pattern specifications can help one determine the transformations needed to incorporate a pattern into a design.

2 Role Models for Pattern Specifications

The Role Models that we use to precisely express pattern specifications characterize UML models, thus they are based on the UML metamodel. The UML metamodel is defined at level M2 of the UML metamodel architecture. Level M1 consists of UML models (i.e., instances of the M2 metamodel) and level M0 consists of instances of the models at level M1. In our work, *a role* is a property-oriented specification that determines a subset of the role's base instances, where a role *base* can be any UML metamodel class (e.g., *Class*, *Generalization*). For example, a role with the *Class* base determines a subset of class constructs. An instance of a role's base that has the properties specified in a role can *play the role*, that is, it is a *realization* of the role. *A Role Model* is a structure of roles. A *Role Model realization* is a model (e.g., a static structural diagram, sequence diagram) that consists of realizations of the roles in the Role Model.

We have developed two types of Role Models that can be used in a pattern specification:

Static Role Models (SRMs) : A SRM is a characterization of a family of UML static structural models, that is, models that depict classifiers (e.g., UML classes and interfaces) and their relationships with each other (e.g., UML associations and generalizations).

Interaction Role Models (IRMs) : An IRM is a characterization of a family of interaction diagrams (e.g., collaboration and sequence diagrams).

This paper focuses on SRMs and how they can be used to support Class Diagram refactoring. SRMs and IRMs are more fully described in [6] and [7].

2.1 Overview of Static Role Models (SRMs)

A SRM consists of roles and relationships between roles. In this subsection we describe roles in SRM and the relationships that can exist between them.

SRM Roles. A SRM role characterizes a set of UML static modeling constructs (e.g., class, and association constructs). For example, a SRM classifier role (i.e., a SRM role with the metamodel class *Classifier* as a base) defines properties that classifiers (e.g., classes, interfaces) must have if they are to realize the role, while a SRM relationship role defines properties that UML relationships (e.g., associations, generalizations) must have if they are to realize the role.

The structure of a SRM role is composed of three compartments. The top compartment has three parts: a role base declaration of the form $\ll$ *Base Role* $\gg$, where *Base* is the name of the role's base (i.e., the name of a metamodel class); a role name declaration of the form */RoleName*, where *RoleName* is the name of the role; and a *realization multiplicity* that specifies the allowable number of realizations that can exist for the role in a realization of the SRM that includes the role. The remaining compartments contain specifications of the properties that realizations of the role must possess.

Metamodel-level constraints in the second compartment are well-formedness rules, expressed in the Object Constraint Language (OCL) [14], that determine the form of UML metamodel class instances that can realize the role. Specifically, the UML well-formedness rules and the metamodel-level constraints defined in a SRM role determine the form of its realizations. The third compartment specifies *Feature roles* that characterize application-specific properties. A feature role consists of a name, a *realization multiplicity*, and a property specification expressed as a *constraint template*. The realization multiplicity specifies the number of realizations a feature role can have in a SRM realization. In this paper, we do not show feature role realization multiplicities if they are "1..*". The *constraint template* of a feature role determines a family of application-specific properties. Features (e.g., attributes, operations) of model constructs that play a SRM role realize feature roles of the SRM role. There are two types of feature roles: (1) *Structural roles* specify state-related properties that are realized by attributes or value-returning operations in a SRM role realization. (2) *Behavioral roles* specify behaviors that are realized by a single operation or method, or by a composition of operations or methods in a SRM role realization. Feature roles are detailed in [6] and [7].

Role Relationships. A role can be associated with another role, indicating that the realizations of the roles are associated in a manner that is consistent with how the bases of the roles are related in the UML metamodel. We use the UML form of association to represent role associations. Role associations can be named and can have multiplicities associated with their ends.

The SRM shown in Fig. 1(a) contains two associations, *has-child* and *has-parent*, between the roles *FactoryGeneralization* and *Factory*. These associations indicate that realizations of *Factory* can be related to each other via generalizations (realizations of *FactoryGeneralization*). Roles with a common set of characteristics can be generalized by a role, called a *role generalization*, that consists only of the common characteristics. The roles that are generalized are called *role specializations*. A role specialization inherits the associations, the metamodel-level constraints and the feature roles defined in the generalization. We use the UML generalization symbol to indicate a generalization/specialization relationship between roles. A role specialization characterizes a subset of the realizations characterized by its parent role (currently, a specialization role is restricted to having only one parent role) [6] [7]. In Fig. 1(a), the *Factory* role captures properties that are common to *AbstractFactory* and *ConcreteFactory* roles. The constraint $\{XOR\}$ between *FactoryGeneralization* and *FactoryRealization* indicates that there cannot be a realization of both these roles between any two *Factory* realizations. This is an example of a (pre-defined) constraint across roles.

All realizations of *Factory* are either realizations of *AbstractFactory* or *ConcreteFactory*. Such a role is said to be *abstract*. Each role is associated with a tagged value that indicates whether it is abstract or not: the tagged value {*realizable* = *false*} indicates that the role is abstract while {*realizable* = *true*} indicates that the role is not abstract. In this paper, the {*realizable* = *true*} tag is omitted if the role is not abstract.

2.2 Role Model Abstraction

SRMs can be abstracted at different levels in order to gain an understanding of a family of UML designs without being distracted by detailed information. A detailed SRM shown in Fig. 1(a) is abstracted into SRMs in Fig. 1(b) and (c). SRM Abstraction is expressed in three levels: detailed, abbreviated (unfolded) and folded level.

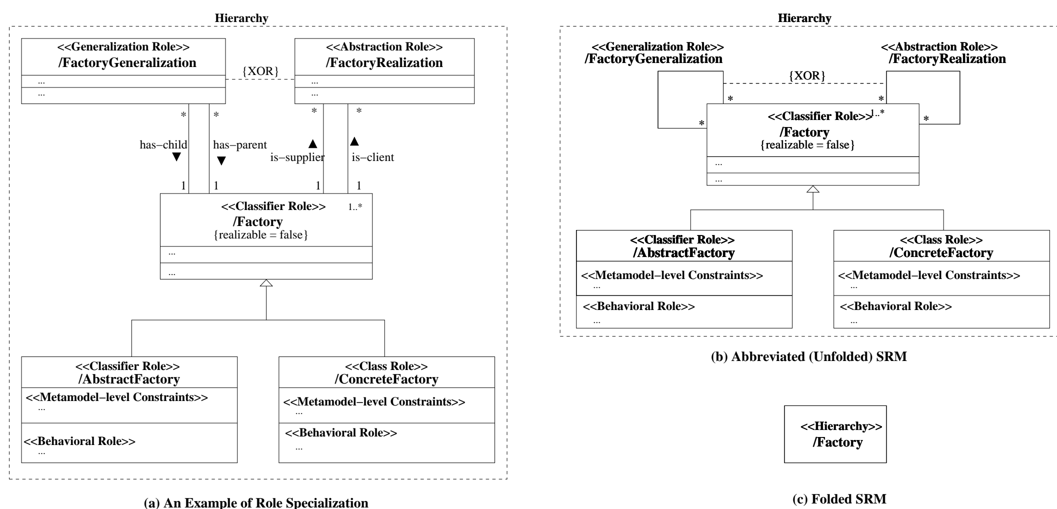

Fig. 1. An example of SRM in three abstraction levels

Detailed SRM. Detailed abstraction fully describe properties that must be specified precisely at metamodel-level and model-level. Such abstractions also includes relationship roles expanded into the same structure as in the UML metamodel (e.g., Association must be connected through AssociationEnds). Fig. 1(a) has shown an example of detailed SRM.

Abbreviated SRM. Abbreviated SRM includes feature roles and hierarchies that were folded. SRMs are abbreviated in the sense that they do not show details of model-level constraint templates. Abbreviated SRMs also hide details of relationship roles such as *AssociationEnd*, *Generalization*, and *Realization*. An examples of abbreviated abstraction is shown in Fig. 1(b).

Folded SRM. SRMs often contain recurring structures which can be viewed as a pattern. An example of such a structure is shown in Fig. 1(a) and (b). The notion of hierarchy can be abstracted by a stereotype structure $\ll$ *Hierarchy* $\gg$ to obtain a terse structural view of a role model. Properties not shown in this level are: structures inside of hierarchies, metamodel-level constraints, feature roles. An example of folded abstraction is shown in Fig. 1(c).

2.3 Realizing SRMs

A model is said to *realize a SRM* if the following holds:

(1) The model constructs that are intended to realize SRM roles do realize the roles. This means that the model elements satisfy the metamodel-level constraints of the role, and the constraints expressed in

the realizing model (e.g., pre- and post-conditions for operations, and constraints on attributes) imply the model-level constraints obtained by instantiating the parameters of the feature roles.

(2) The model conforms to constraints expressed across roles or their realizations (e.g., realization multiplicities, role association multiplicities).

3 An Abstract Factory (AF) SRM

Fig. 2 shows an abbreviated SRM characterizing a well-defined subset of Abstract Factory (AF) pattern realizations.

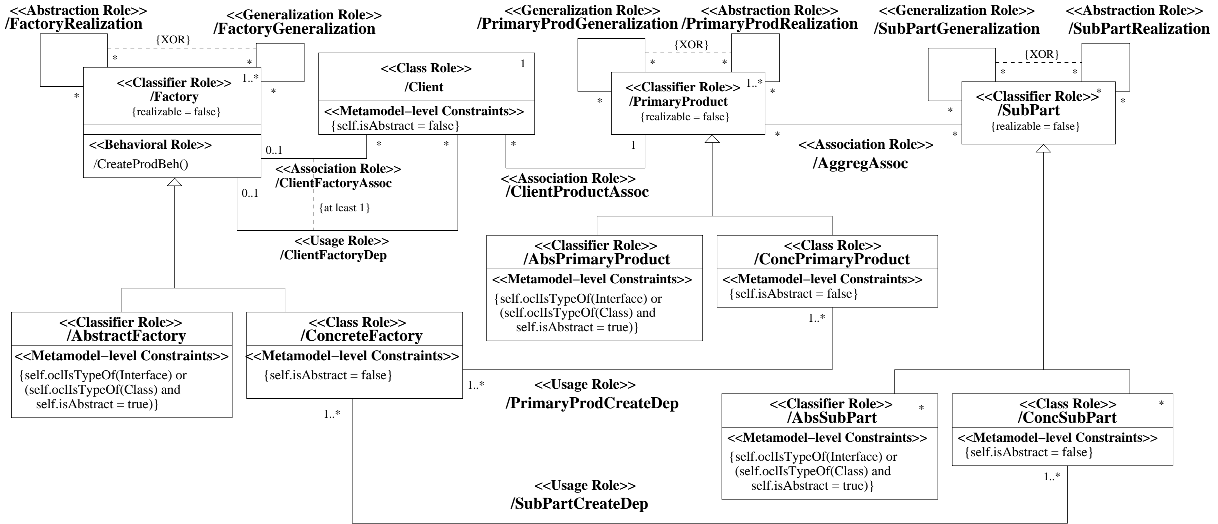

Fig. 2. Abbreviated Abstract Factory SRM for the Pattern-based Refactoring

The AF SRM consists of four major role structures: the *Factory*, the *PrimaryProduct*, *SubPart*, and *Client* role structures where the *Factory*, the *PrimaryProduct*, and the *SubPart* roles are abstract. There are two types of *Factory* roles: *AbstractFactory* characterizes factory interface constructs in the form of UML interfaces and abstract classes, and *ConcreteFactory* characterizes concrete factory classes. Realizations of the *Factory* specialization roles can be connected either by a realization of the *FactoryRealization* role (a UML $\ll realize \gg$ relationship) or a realization of the *FactoryGeneralization* role (a UML generalization relationship). This role structure permits hierarchical and non-hierarchical realizations. The hierarchical structures are created using generalization and UML $\ll realize \gg$ relationships. In a $\ll realize \gg$ relationship, the supplier must be an interface or a class that can have operations (but no methods), while the client must be a class. In a generalization relationship, the superclass and the subclass must both be interfaces or both be classes. The *PrimaryProduct* and *SubPart* role structure also permits hierarchical and non-hierarchical product classifier realizations.

A realization of the AF SRM consists of exactly one realization of *Client*. The pre-defined constraint $\{at\,least\; 1\}$ shown between the *ClientFactoryAssoc* and *ClientFactoryDep* relationship roles in Fig. 2 states that each pair of *Factory* and *Client* realizations must be connected by at least one relationship that is either an association (a realization of *ClientFactoryAssoc*) or a usage dependency (a realization of *ClientFactory-Dep*), that is, the *Client* realization is connected to each *Factory* realization via an association or via a usage dependency. *AbstractFactory* and *ConcreteFactory* realizations have one or more behaviors that realize *CreateProdBeh*. Realizations of *ConcreteFactory* are connected to one or more realizations of *ConcSubPart* via usage dependencies (realizations of *SubPartCreateDep*). More models realizing our AF SRM are illustrated in [6].

4 Pattern-Based Model Refactoring

Model refactoring occurs when a source model is transformed to a target model that is better than the source model with respect to particular quality attributes. The source and target models are at the same level of abstraction, that is, model refactoring is not a detailing of source models. In this section we illustrate how SRMs can be used to support systematic model refactoring.

4.1 Supporting Systematic Model Refactoring

A systematic, automatable approach to model refactoring is possible when the transformations needed to accomplish refactorings involving well-defined sets of source and target models, can be precisely characterized. In our approach, a characterization of a family of model refactorings consists of the following elements:

A source model set: This set consists of source models for the refactorings. The set is characterized by Role Models.

A target model set: This set consists of target models for the refactorings. The set is characterized by Role Models (as illustrated in section 3).

A transformation set: This set consists of transformations that each accomplish the refactoring goal. A transformation specifies how a source model is transformed to a target model. In our work, transformations are expressed in terms of sub-transformations a source model undergoes as it is transformed to a target model.

A transformation can be explicitly defined in terms of *transformation traces*, where a transformation trace is a sequence of models representing the sub-transformations a source model goes through in a refactoring. A transformation trace starts with a model from the source model set and ends with a model from the target model set. A single transformation is defined by the set of all transformation traces that have the same source and the same target models (a transformation can be accomplished in one or more ways). Creating explicit definitions of non-trivial transformations may not be an easy or feasible task. Implicit definitions of transformations, in which sub-transformations are specified in terms of their effects, may be more appropriate in these cases. In this paper, we use implicitly defined transformations (expressed in English) to illustrate our approach to pattern-based model refactoring.

4.2 Refactoring Using the Abstract Factory Pattern

Folded SRMs for an Abstract Factory(AF) refactoring are shown in Fig. 3. An AF refactoring characterization that specifies transformations to be carried out on models that may contain classes representing aggregate products of different types (e.g., see the $Maze$ product structure in Fig. 4), is defined below:

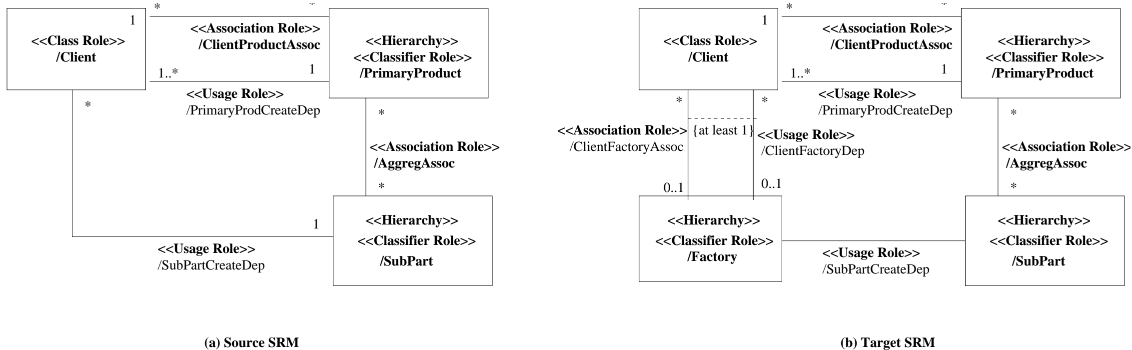

Fig. 3. Folded Source and Target SRMs for an Abstract Factory Refactoring

Source set characterization: The folded SRM characterizing the source models for the refactoring is shown in Fig. 3(a). The SRM characterizes models with the following form: (1) a client class (a realization of

Client) with operations that create product objects; (2) one or more primary product classifier hierarchies (realizations of the *PrimaryProduct* role structure) that are associated with the client class; and (3) zero or more sub-product classifier hierarchies (realizations of *SubPart* role structure) that are associated with primary product classes. A primary product class that is associated with a subpart class represents an aggregate product, with the subpart classes representing the parts. The Maze Game Class Diagram shown in Fig. 4 is a realization of this SRM: *MazeGame* realizes the *Client* role, *Maze* and its specializations each realize the *PrimaryProduct* role, and *Room*, *Door*, *Wall* and their specializations each realize the *SubPart* role. The abbreviated (unfolded) form of Fig. 3(a) is not shown in this paper.

Target set characterization: The abbreviated SRM characterizing the target set has been shown in Section 3 (see Fig. 2). Fig. 3(b) can be expanded (unfolded) into Fig. 2.

Transformation set characterization: The sub-transformation sequence that accomplishes the pattern-based transformation in the folded SRMs (Fig. 3) can be defined as follows:

Step 1. Create the factory hierarchy.

Step 2. Migrate the create operations from the realization of the *Client* role in the source model to the appropriate factory classes.

Step 3. Link the factory classes to the *Client* realization using associations or usage dependencies.

More detailed steps of sub-transformation sequence can be defined constructively from the abbreviated(unfolded) SRMs. Step 1 can be accomplished by (1) creating an abstract factory classifier (an abstract class or interface that is intended to realize the target SRM *AbstractFactory* role) for each realization of the source SRM *AbsPrimaryProduct* role in the source model, (2) creating a concrete factory class (an intended realization of the target SRM *ConcreteFactory* role) for each realization of *ConcPrimaryProduct* role in the source model, and (3) linking them (using generalization or realization relationships) in accordance with the product hierarchy. Step 2 can be done by allocating *create* operations to factories. Allocation is determined by the associations between the primary parts and their subparts: the *create* operations for each subpart linked to a primary part are placed in the factory corresponding to the primary part. This results in the removal of the create dependencies between the *Client* realization and the product classifiers, and creation of create dependencies between the factories and the product classifiers.

Fig. 4 shows a class diagram that reflects the static, structural aspects of a design for a maze game (this design is an adaptation of an example given in [8]). In this design, the *MazeGame* is responsible for creating the different types of mazes and their parts. If a new type of maze or maze part is added, the *MazeGame* class would have to undergo significant change. Incorporating the Abstract Factory pattern into this design will result in a more flexible design in which the maze creation aspects are localized in factories that can be accessed by the MazeGame. The model shown in Fig. 4 realizes the source model SRM shown in Fig. 3, thus we can apply the refactoring defined above to the model. (See [6] for the abbreviated SRM.) The result of applying the sub-transformations defined above to this problem is outline below:

1. Create factory hierarchies: In the source model there is only one realization of *AbsPrimaryProduct*, *Maze*, thus we create one abstract factory construct, an abstract class, with the name *MazeFactory*. A concrete factory specialization of *MazeFactory* is created for each concrete specialization of *Maze*, *BombedMazeFactory* and *EnchantedMazeFactory*.

2. Migrate create operations to the factories: The create operations are moved out of *MazeGame* and placed into 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 client to factories: The modeler has the choice to link the client to the factories using associations or dependencies. In this case we chose to use an association to link *MazeGame* to *MazeFactory*.

The result of the refactoring is shown in Fig. 5. The above manual refactoring illustrates that precise pattern forms can provide clear indicators of what needs to be changed in a design in order to incorporate pattern properties.

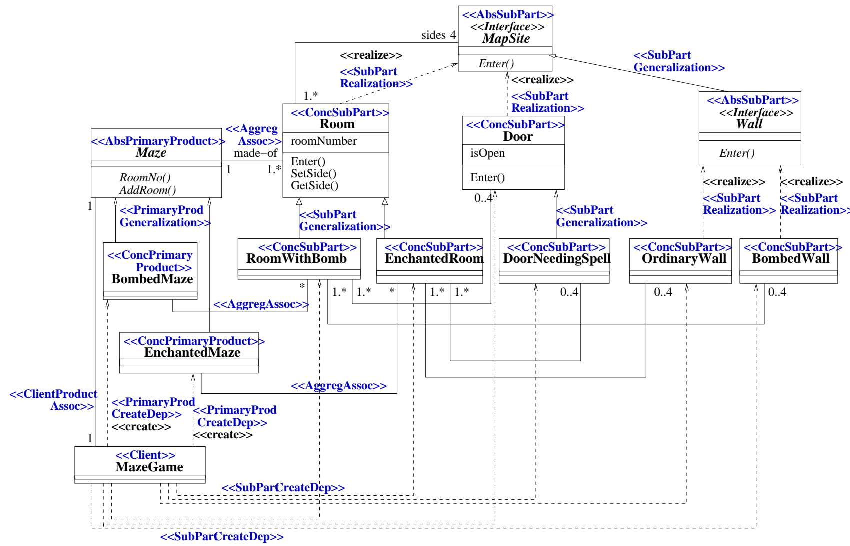

Fig. 4. Maze Game adapted from the GoF book

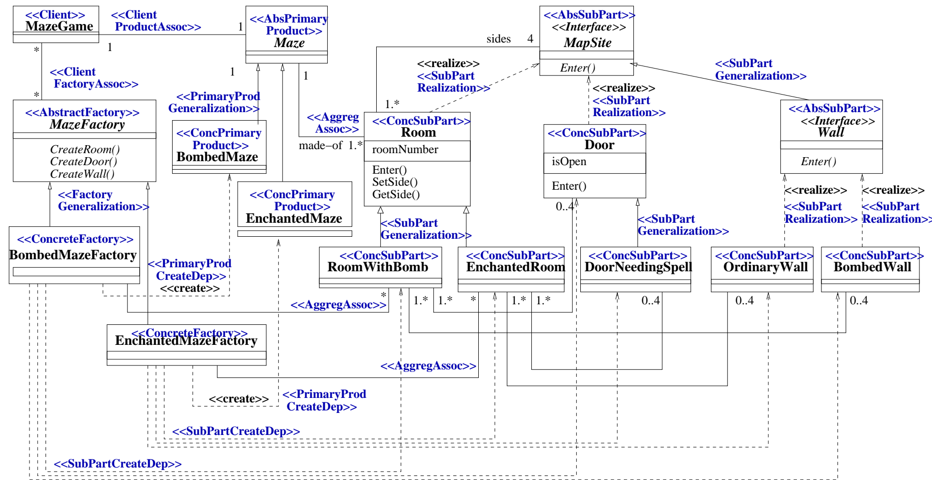

Fig. 5. Maze Game refactored with Abstract Factory pattern

5 Related Work

There has been considerable work on object-based notions of roles (e.g., see [12, 13]). An object-based role specifies properties that objects in the run-time environment must have if they are to play the role. Our Role Models, which are intended to support pattern-based model refactoring, require that roles be played by model elements (e.g., classes and associations), and not by objects.

There has also been some work on precisely defining pattern properties. Lauder and Kent [10] use graphical constraint diagrams for precise visual presentation of patterns. Guennec et al. [9] use a metamodeling approach that is based on the UML metamodel. Their approach provides an alternative representation in terms of meta-collaborations that utilize a family of recurring properties initially proposed by Eden [4]. The paper does not, however, describe how properties other than hierarchical structures of classifiers are specified, nor is there a clear notion of what it means for a model to realize a role model.

Refactoring code has been considered an important part of the evolution of object-oriented software and has gained importance particularly with the acceptance of eXtreme Programming techniques [1, 5]. Tokuda and Batory [15, 16] proposed a refactoring approach to support design patterns as target states for software restructuring efforts. Butler and Xu [3] extend the concept of refactoring to other models, such as the feature model, use case model, and architecture, of an object-oriented framework. Our refactoring approach uses precise forms of design patterns (expressed in UML terms) as the basis for refactoring models.

6 Conclusions and Future Work

A goal of our work on model-based software development is to develop systematic pattern-based model refactoring techniques for incorporating patterns into designs. In this paper we introduced the notion of Role Models as a means for precisely characterizing families of pattern problems (source models for refactoring) and solutions (target models for refactoring). For the creational patterns (like Abstract Factory pattern), SRMs are enough to characterize its pattern-busingd refactoring. However IRMs can be used for refactoring using any other categories of patterns (e.g., Visitor or Observer pattern)

For lack of space in the paper, we have illustrated the approach for the Abstract Factory pattern only. To date, we have developed full Role Models for the following GoF patterns: Abstract Factory, Singleton, Observer, Composite and Observer. We described the process of pattern-based model refactoring and illustrated it using the example of the Abstract Factory pattern.

We are currently developing tool support for the model refactoring technique described in this paper. A tool-supported model refactoring technique can be an essential part of a model-based software development environment in which models are used as the primary development artifacts. Such environments allow developers to raise the level of abstraction at which they develop systems (see http://www.omg.org/mda).

References

1. K. Beck. *eXtreme Programming eXplained: Embrace Change*. Addison-Wesley, 1999.
2. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. Wiley, 1996.
3. G. Butler and L. Xu. Cascaded refactoring for framework evolution. In *SSR'01*, 2001.
4. A. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, University of Tel Aviv, Israel, 1999.
5. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
6. R. B. France, D. K. Kim, and E. Song. Patterns as precise characterizations of designs. Technical Report 02-101, Computer Science Department, Colorado State University, 2002.
7. R. B. France, D. K. Kim, E. Song, and S. Ghosh. Using Roles to Characterize Model Families. In *Proceedings of the 10th OOPSLA Workshop on Behavioral Semantics: Back to Basics*, Seattle, Washington, November 2001.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
9. A.L. Guennec, G. Sunye, and J. Jezequel. Precise modeling of design patterns. In *Proceedings of UML'00*, 2000.
10. A. Lauder and S.Kent. Precise visual specification of deisgn patterns. In *Proceedings of ECOOP'98*, 1998.
11. W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995.
12. T. Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects: The OORAM Software Engineering Method*. Manning/Prentice Hall, 1996.
13. D. Riehle and T. Gross. Role Model Based Framework Design and Integration. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 117–133, Vancouver, Canada, October 1998. ACM Press.
14. The Object Management Group (OMG). Unified Modeling Language. Version 1.3, OMG, http://www.omg.org, June 1999.
15. L. Tokuda and D. Batory. Automated software evolution via design pattern transformations. In *3rd International Symposium on Applied Corporate Computing*, 1995.
16. L. Tokuda and D. Batory. Evolving object-oriented designs with refactorings. In *Proceedings of the 14th IEEE International Conference of Automated Software Engineering*. IEEE CS Press, 1999.