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, 12]) 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.

The remainder of the paper is organized as follows. In Section 2 we describe how Role Models can be used to precisely express pattern specifications. In Section 3 we illustrate the specification of the Abstract Factory pattern using Role Models. In Section 4 we describe the pattern-based model refactoring technique and give an example of its application using the Abstract Factory pattern. In Section 5, we present an overview of related work on pattern specification and model refactoring, and compare them with our approach. We conclude in Section 6 with an overview of how we plan to evolve this work.

2. 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
A SRM consists of roles and relationships between roles. In this subsection we describe SRM roles and the relationships that can exist between them.

2.1 Static Role Models (SRMs)
A SRM consists of roles and relationships between roles. In this subsection we describe SRM roles and the relationships that can exist between them.

2.1.1 SRM Roles
A SRM role characterizes a set of UML static modeling constructs (e.g., class, and association constructs). For example, 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 shown in Fig. 1(a). The top compartment has three parts: a role base declaration of the form `<< BaseRole >>`, where `BaseName` 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. The second compartment contains metamodel-level constraints and the third compartment contains properties expressed as feature roles.

Metamodel-level constraints are well-formedness rules, expressed in the Object Constraint Language (OCL) [11], 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.

Feature roles 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. An example of a structural role that can be realized by class attributes is given below:

   `/CurrentValue 1..1
   {[[CurrentValue]] <= [[Threshold]]}`

   In the above example, `CurrentValue` is the feature role name and the realization multiplicity following it (1..1) indicates that there must be exactly one realization of this role in a realization of the SRM role containing this feature role. The constraint template enclosed in the brackets `{}` (parameters are surrounded by `[[ ]]`) states that realizations of the `CurrentValue` feature role must always have a value less than or equal to the value of a realization of another feature role named `Threshold`.

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. An example of a behavioral role is `CreateProdBeh` shown in Fig. 1(b) and (c). A realization of `CreateProdBeh` creates a new instance of a realization of `Product`. Substituting the names of

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].

OCL: 

/CurrentValue 1..1
{[[CurrentValue]] <= [[Threshold]]}
realizations for the role names enclosed in the double square brackets ([[,]]) results in an application-specific property, called a **model-level constraint**, expressed in the OCL. For example, substituting Wall, the name of a Product role realization, for Product in the CreateProdBeh constraint template results in the following model-level constraint:

```
makeWall():Wall
{pre: true
 post: result = p and p.oclIsNew() = true }
```

Establishing that a model element realizes a SRM role involves proving that the constraints associated with the model element imply the model-level constraints obtained by suitably instantiating the role’s constraint templates, and determining that the realization multiplicities associated with the feature roles are not violated. Feature roles are detailed in [6] and [7].

### 2.1.2 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. For example, a Class role can be directly associated with an AssociationEnd role, but not with an Association role because the UML metamodel does not directly associate the Association metamodel element with the Class metamodel element. 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. 2 contains two associations, has-child and has-parent, between the roles Factory-Generalization and Factory. These associations indicate that realizations of Factory can be related to each other via generalizations (realizations of FactoryGeneralization).

Fig. 2 also shows two generalization relationships between roles. 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). This can be accomplished by further restricting the properties inherited from the parent role, for example, by restricting the multiplicities of inherited associations between roles, or by adding metamodel-level constraints that further constrain the form of realizations [6].

In Fig. 2, the Factory role (the generalization) captures properties that are common to AbstractFactory and ConcreteFactory roles (the specializations). The specializations inherit the associations has-child, has-parent, is-supplier, and is-client, as well as the metamodel-level constraints and feature roles in the generalization Factory (not shown). 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.

2.1.3 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).

2.1.4 Abbreviated SRMs
An abbreviated SRM allows one to gain an understanding of a pattern at a particular level of abstraction without being distracted by detailed information. An abbreviated SRM for is shown in Fig. 3(b). The abbreviated SRM contains an abstraction of a generalization role structure as shown in Fig. 3(a).

Figure 3: An Abbreviated SRM

2.2 Roles at the UML Metamodel level
Fig. 4 shows how Role Models relate to the UML metamodel. The realizations of the SRM shown in the figure are Class Diagrams. The two class diagrams shown at level M1 are realizations of the SRM. Each role in the SRM determines a subset of UML model constructs, for example, the association role Assc1 in the SRM shown at level M2 determines a subset of UML association constructs (i.e., it determines a specialization of the metamodel class Association). At the model level (M1), ClassB and ClassC are both realizations of the $R1$ role.

Fig. 4 also illustrates the difference between our roles and UML collaboration role: The UML classifier role RoleA (at level M1) is a view of ClassA, that is, RoleA consists of a subset of ClassA properties (attributes, operations, associations). Objects of ClassA (at level M0) can play the role of RoleA.

3. AN ABSTRACT FACTORY (AF) SRM
Fig. 5 shows an abbreviated SRM characterizing a well-defined subset of Abstract Factory (AF) pattern realizations, and a more detailed SRM is shown in Fig. 6. The relationship roles shown in Fig. 6 are abstracted in the abbreviated SRM shown in Fig. 5.

Figure 5: Abbreviated Abstract Factory SRM

The AF SRM consists of two major role structures: the Factory and the Product role structures where the Factory and the Product 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 \(\text{realize} \) 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 \(\text{realize} \) relationships. In a \(\text{realize} \) 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 Product 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. 5 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 ClientFactoryDep), that is, the Client realization is connected to each Factory realization via an association or via a usage dependency. Similarly, a Client realization is connected to each Product realization via an association or a usage dependency.

AbstractFactory and ConcreteFactory realizations have one or more behaviors that realize CreateProd-Beh. Realizations of ConcreteFactory are connected to one or more realizations of ConcreteProduct via usage dependencies (realizations of ConcreteDep).

### 3.1 Role Models and the UML MetaModel

![UML Metamodel view of the factory part of Abstract Factory SRM](image)

Fig. 7 gives the metamodel corresponding to the factory part of the AF SRM. Classes that can play the ConcreteFactory role are instances of the metamodel Class specialization ConcreteFactory, and classes that can play the AbstractFactory role are instances of the metamodel Classifier specialization AbstractFactory. Similarly, the generalizations that can play the FactoryGeneralization role are instances of a specialization of the Generalization metaclass called FactoryGeneralization, and the \(< \text{realize} \rangle\) relationships that can play the FactoryRealization role are instances of a specialization of the Abstraction metaclass called FactoryRealization. Instances of this specialized metamodel are realizations of the AF SRM.

### 3.2 Realizations of the AF SRM

The smallest models that can realize our AF SRM consist of a Factory realization (i.e., a realization of AbstractFactory or ConcreteFactory), a Product realization, a Client realization, a relationship between the Client realization and the Factory realization that is either an association or a usage dependency, and a relationship between the Client realization and the Product realization that is either an association or a usage dependency. If the Factory and Product realizations are concrete classes, then they are connected by a usage dependency (a realization of ConcreteDep).

Two realizations of the AF SRM are shown in Fig. 8 (bold stereotypes are used in model constructs to indicate that the constructs realize the roles named in the stereotype). The realization shown in Fig. 8(a) shows non-hierarchical realizations of Factory and Product. The derived associations (not shown) between MazeGame and the other products Room, Wall and Door (obtained by traversing the associations between Maze and Room and between Room and its sides) are all realizations of the ClientProductAssoc.

Fig. 8(b) shows a hierarchical realization of Factory (the generalization hierarchy with MazeFactory as the root superclass), a realization of ConcreteProduct called Maze, and three product hierarchical realizations of Product: Door is an interface that has \(< \text{realize} \rangle\) relationships with PlainDoor and EnchantedDoor, Wall is a concrete class with a specialization named BombedWall, and Room is an abstract class that is specialized by PlainRoom and RoomWithBomb. The derived associations (not shown) between MazeGame and the specializations of MapSite are also realizations of ClientProductAssoc. The dependency between MazeGame and MazeFactory is inherited by the specializations, thus there is a realization of ClientFactoryDep for each Factory realization as required by the pattern.

### 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

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. 11), is defined below:

**Source set characterization:** The abbreviated SRM characterizing the source models for the refactoring is shown in Fig. 9. 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. 11 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.

**Target set characterization:** The abbreviated SRM characterizing the target set is shown in Fig. 10. This SRM is a specialization of the AF SRM given
in the previous section in that it determines a proper subset of the models characterized by the AF SRM. The target model SRM defines only association relationships between the client and the product hierarchy, and distinguishes aggregate products from their parts.

Transformation set characterization: The subtransformation sequence that accomplishes the pattern-based transformation can be defined constructively as follows:

1. Create the factory hierarchy. This is 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.
2. Migrate the create operations from the realization of the *Client* role in the source model to the appropriate factory classes. The allocation of create operations to factories 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.

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

Fig. 11 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. 11 realizes the source model SRM shown in Fig. 9, thus we can apply the refactoring defined above to the model. 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. 12. 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.

5. **RELATED WORK**

There has been considerable work on object-based notions of roles (e.g., see [13, 14]). 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 struc-
tures 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 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).

7. REFERENCES
[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.