
Supporting Effective Software Modeling

Robert B. France* — **Sudipto Ghosh*** — **Daniel E. Turk****

** Computer Science Department
Colorado State University
Fort Collins, CO, 80523, USA
{france,ghosh}@cs.colostate.edu*

*** Computer Information Systems
Colorado State University
Fort Collins, CO, 80523, USA
Dan.Turk@colostate.edu*

RÉSUMÉ. Dans cet article nous identifions les raisons de la faible utilisation des techniques de modélisations dans l'industrie et nous donnons un aperçu d'une approche basée modèle qui peut améliorer l'utilisation des modèles pour le développement des systèmes complexes. Les mécanismes présentés ici reposent sur une approche du développement guidé par les modèles et qui met en avant (1) la réutilisation explicite des artefacts de modélisation, (2) la transformation systématique des modèles par l'utilisation de patrons et de restructurations, et (3) une séparation multi-dimensionnelle des aspects par la modélisation orientée aspect.

ABSTRACT. In this paper we identify some of the reasons behind the low use of modeling techniques in industry and give an overview of model-based development mechanisms that can enhance the use of models in the development of complex systems. The mechanisms discussed in this paper support a model-driven approach to software development that emphasizes (1) explicit reuse of modeling artifacts, (2) systematic transformation of models through pattern-based refactoring, and (3) multi-dimensional separation of concerns through aspect-oriented modeling.

MOTS-CLÉS : aspect, modèle, modélisation, MDA, patron, restructuration, UML

KEYWORDS: aspect, model, modeling, MDA, patron, model transformations, UML

1. Introduction

Advocates of software modeling techniques point to the growing complexity of software systems when highlighting the need for good abstractions during software development. Software systems are becoming increasingly complex as a result of (among other considerations) (1) efforts to build interoperable systems and to integrate disparate systems (resulting in systems of systems), (2) the desire to distribute functionality and data to improve quality of service, and (3) the need to address concerns such as security, fault tolerance, concurrent access to services and data, and run-time reconfigurability of systems. Rapid changes in technology and the desire to gain competitive advantage through the innovative use of information technologies has resulted in pressure to develop complex systems faster. This pressure has led developers of complex software to view modeling techniques warily.

Our interactions with software developers in industry indicate that models are seldom developed and used for the following reasons :

1) It is sometimes not easy to determine which diagrams and modeling tools are appropriate in a given situation : Modeling notations such as the Unified Modeling Language (UML) [The 99] provide a number of diagrams that can be used to model systems from a variety of perspectives. Unfortunately, it is sometimes not clear to developers what diagrams and modeling constructs can be used to capture useful abstractions of the systems. While papers and texts extoll the virtues of particular modeling techniques and languages, they seldom discuss weaknesses and the situations in which modeling techniques may not be suitable.

2) Considerable effort is spent building and evolving models with questionable value : Developers are sometimes mandated to use particular modeling techniques in their work. Unfortunately, in some cases the mandates are not accompanied by justifications that convince developers of the worth of the models they are building. The result can be the creation of models that do not provide significant insights that can help developers tackle complexity more effectively. Evolving these models as requirements and implementations change is labor intensive (given the current lack of good tool support for evolving models - see below). Without a clear appreciation of the value of models it is very likely that model creation and evolution will produce substandard results (which, in turn, can lead to further disillusionment on the value of modeling).

3) Lack of precise semantics for some modeling notations can impair effective communication : Without a precise semantics confusion over the meaning of constructs can hinder rather than facilitate communication. Furthermore, lack of precise semantics can hinder attempts at providing tool support for evaluating models. For example, tools that can animate or “execute” models for testing purposes require a precise interpretation of the models.

4) Current model development mechanisms do not facilitate timely creation and evolution of models : The learning curve associated with some of the more sophisticated modeling tools has discouraged some developers from applying modeling techniques. Furthermore, very few tools provide adequate support for (a) relating modeling concepts across different diagrams and for defining and maintaining trace, refinement

and realization relationships in models (e.g., see [TRA 01]), (b) transforming models (including transformation to code) and (c) pattern-based reuse of models.

Agile modeling (see <http://www.agilemodeling.com>) targets concerns 1 and 2. In an agile modeling approach¹, developers use only those models that provide clear value to the project. The choice of modeling concepts, notations and techniques can vary across projects within the same organization, and is based primarily on (1) the particular characteristics of the project problem, (2) the experiences of the developers, and (3) the available tools. Clearly, use of an agile modeling process requires a good understanding of the applicability of modeling concepts and notations.

Work on developing technologies supporting the Object Management Group (OMG) Model-Driven Architecture (MDA) initiative should be concerned with all of the above concerns, in particular, concerns 3 and 4. In an MDA approach, models are the primary artifacts of development. Technologies that support the MDA are intended to have good support for separating concerns, transforming models, relating concepts across abstraction levels and across modeling views, and for generating efficient and maintainable implementations from models.

In this paper we give an overview of our work that targets specific modeling concerns :

- Reuse of Modeling Experiences : The use of reusable models can improve the time and effort needed to build models during a project. We outline our work on developing domain specific modeling languages and a repository of reusable modeling experiences in section 2.

- Pattern-Based Model Refactoring : Patterns are a form of reusable design experiences that can be exploited during software modeling. We outline our work on developing support for the systematic incorporation of design patterns into UML models in section 3.

- Aspect-Oriented Modeling : Aspect-oriented modeling (AOM) supports the encapsulation of concerns that cross-cut the structural units of a model. An aspect-oriented model consists of a *primary* model and one or more aspects that capture concerns such as security, fault-tolerance, and distribution. The impact of these concerns on a primary model can be determined by weaving the aspects into the primary model. We outline our work on AOM in section 4.

2. Model-Based Reuse

The systematic reuse of artifacts across a development lifecycle can yield order-of-magnitude improvement in development productivity. Implemented reuse practices and experiences have yielded the following insights :

1. An approach that adheres to the principles, values, and practices of agile modeling as outlined on the agile modeling website.

– **Reuse of domain-specific experiences can significantly enhance development productivity and quality** [ARA 91]. Reusable experiences can be classified as vertical or horizontal. Horizontal reuse occurs when reusable artifacts are created for, and used in a variety of application domains. Design patterns (e.g., see [GAM 95, PRE 95]) and software architectures (e.g., see [BAS 98, BUS 96]) are examples of horizontal experiences packaged for reuse. Vertical reuse occurs when development experiences within a clearly defined application domain are reused. When compared to horizontal artifacts, domain-specific artifacts for a mature domain² are easier to build, and can cover a wider range of development phases, primarily because of their restricted scope. Domain-specific programming languages (DSPLs) [WIL 99] provide examples of vertical reuse. A DSPL provides a language interface for assembling domain-specific code components into programs.

– **Reuse of development artifacts above the code level can significantly reduce development cycle time** [BAS 91]. There is a growing realization that development cycle times can be significantly shortened if reuse opportunities are exploited in all phases of software development; not just in the coding phase [BAS 91, MOR 93, PRI 93]. In the past, a barrier to the reuse of experiences above the code level was the lack of widely-accepted notations for representing requirements and design artifacts. The emergence of the Unified Modeling Language (UML) [The 99] as a de-facto industry modeling standard has the potential to remove this barrier.

– **Effective reuse requires good support for finding, adapting, and incorporating reusable artifacts into project deliverables.** For reuse to be effective it is important that software developers understand when and how to apply the reusable development experiences. In a traditional repository-based approach to reuse, the tasks of selecting, tailoring and integrating reusable experiences into project deliverables are explicitly carried out by developers in conjunction with other development activities. These reuse tasks are nontrivial and, if not properly supported, can be time consuming. Novice “reusers” have the additional task of learning about the available artifacts in the repository.

The above observations motivate our work on developing techniques for building *Reuse-based System Development Environments* (ReSyDEs). Using a ReSyDE, an organization can build a reuse infrastructure that initially supports a repository-based reuse environment³, and evolves towards a development environment that uses *domain-specific modeling languages* (DSMLs) to build models. The reusable modeling experiences in the repository are expressed in terms of pattern templates from which application-specific models can be generated. The evolution of a ReSyDE is based on the premise that as development experiences in an application domain matures (i.e., changes in fundamental design concepts and structures occur infrequently and expert developers have a good understanding of the fundamental concepts) the opportunities for creating high-quality forms of reusable artifacts increases.

2. A mature domain is one that is stable (i.e., major changes in designs occur infrequently) and in which substantial codifiable experience exists

3. One in which developers access the repository directly

Reusable modeling artifacts are organized around an organization's *Enterprise Architecture* (EA) to facilitate cataloging and retrieval. An EA is a model that provides a map of an organization's computer-based systems and business functions, and defines mappings between systems and business functions and between business functions and business goals⁴. An EA can consist of the following models :

- An *Enterprise Business Model* (EBM) is a model of business processes and entities from an organizational perspective. Business models can also be defined for each of an organization's *Business Areas* (BAs) (e.g., Customer Care, Billing).

- The *Enterprise System Architecture* (ESA) consists of models describing (1) the static structure of an organization's systems, and (2) the interactions that take place among the systems. A *BA (Business Area) Architecture* is an elaboration of the parts of an ESA that pertain to the BA.

- The different models and diagrams in an EA are linked via trace mappings across different diagrams in a model, and refinement/abstraction and realization mappings between models at different levels of abstractions.

A ReSyDE exploits the structure imposed by an EA to identify, classify, deploy and manage reusable artifacts in a repository-based reuse infrastructure.

2.1. Representing Reusable Modeling Experiences

A ReSyDE supports two forms of reusable modeling experiences : template forms of models that can be used to generate application-specific models, and modeling languages in which domain-specific constructs are embedded (DSMLs). The model templates are stored in a repository and are organized around an organization's EA. In [FRA 01b] we describe the processes used to create model templates and DSMLs. In this paper we focus on the forms of model templates and DSMLs.

A packaged model template consists of two parts :

- 1) **Usage context** : This part consists of a narrative description of the situations in which the reusable model can be applied, and the consequences of using the reusable model.

- 2) **Structural and behavioral models** : This part consists of template forms of *Role Models*⁵ from which application-specific models can be generated. A Role Model is a structure of roles. In the template form of Role Models, a role is a parameterized UML model element. Generating a model from a template form of a Role Model involves substituting application-specific values for role parameters.

We have developed two types of Role Models [FRA 01a, KIM 02] :

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

4. This definition of EA is broader than those found in the business modeling literature

5. The specification form of a Role Model is used to define DSMLs as will be discussed in a later section.

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).

2.1.1. Template Form of SRMs

The template form of an SRM consists of roles expressed as parameterized static modeling constructs (e.g., a class role is a parametrized class and an association role is a parameterized association). An example of the template form of an SRM is shown in Fig. 1. This template can be used to stamp out models that conform to the Observer-

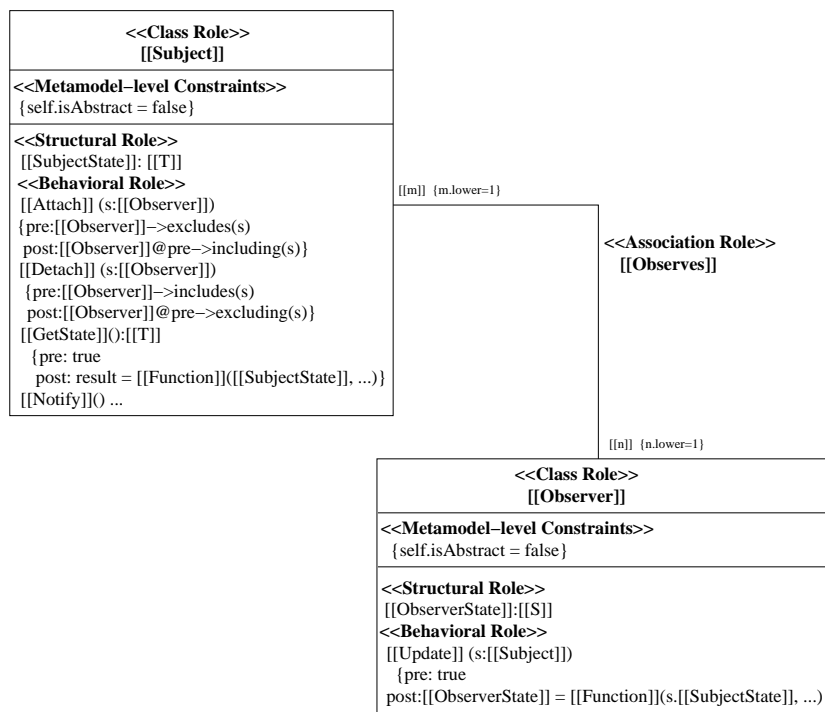


Figure 1. Example of a SRM in Template Form

Subject pattern. The Role Model consists of two class roles (*Observer*, *Subject*) and one association role (*Observes*). The roles are parameterized forms of their corresponding model constructs (parameters are enclosed in the brackets `[[,]]`). A class role consists of metamodel-level constraints and feature roles (structural and behavioral roles). *Metamodel-level constraints* are well-formedness rules, expressed in the OCL [The 99], that determine the form model constructs that can be generated from the roles. For example, only concrete (non-abstract) classes can be generated from the roles *Observer* and *Subject*. Features of classes are generated by instantiating *Feature roles*. Attributes

are generated from structural roles, while operations are generated from behavioral roles. Both structural and behavioral roles can be associated with template forms of constraints (e.g., the pre and post-condition templates shown for the behavioral roles in the example).

Association roles are parameterized associations. The association role *Observes* has two multiplicity parameters m and n that can only be substituted by ranges with a lower bound of 1.

2.1.2. Interaction Role Models (IRMs)

IRMs are parameterized forms of UML collaboration and sequence diagrams. A collaboration-styled IRM for the Observer template shown in Fig. 1 is shown in Fig. 2(a). The IRM describes an interaction pattern in which the *Notify* behavior involves invoking the *Update* behavior in each of its observers. During the execution of the *Update* behavior, an observer invokes the *GetState* behavior of the subject. The variable st represents the subject state that is returned as a result of the *GetState* behavior.

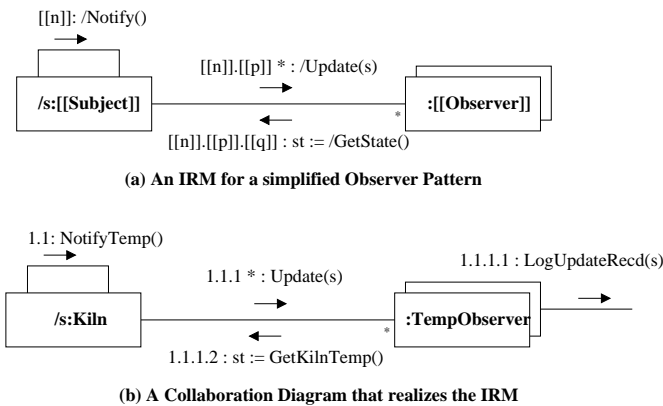


Figure 2. IRM for the simplified Observer pattern

The rectangular boxes in Fig. 2(a) are collaboration role templates. Substituting values for the parameters of a collaboration role template results in a UML classifier role (a UML classifier role is a projection of a UML class) [BOO 99]. The classifier roles obtained by instantiating collaboration role templates are projections of classes generated from the SRM roles indicated in the templates. For example, by substituting *Kiln* for *Subject* in $/s : [[Subject]]$ we get the UML classifier role $/s : Kiln$ shown in Fig. 2(b).

We place a “/” in front of the behavioral feature role names (e.g., $/Notify()$) to indicate they are *message roles* that represent calls to operations generated from corresponding behavioral roles in SRMs. The sequence labels in an IRM are generic (as indicated by surrounding them in $[[,]]$), for example, the expression $[[n]] : /Notify()$ indicates that the *Notify* stimulus is the n th interaction in the sequence, where a specific value for n

must be provided. Similarly, the expression $[[n]].[[p]].[[q]] : st := /GetState()$ indicates that a message realization of *GetState* is the q th nested interaction of the p th nested interaction of the n th outermost interaction. The IRM allows other interactions to occur between the */Update* and the */GetState* interactions as indicated by the sequence expressions.

Fig. 2(b) shows a collaboration diagram that was generated in part from the IRM in Fig. 2(a). Part of the collaboration diagram was created by substituting the names of the operations generated from corresponding behavioral roles and values for n, p, q as follows : $n = 1.1, p = 1$ and $q = 2$. An additional interaction was added to the generated collaboration diagram (interaction 1.1.1.1); this interaction logs the receipt of the *Update* stimulus before the observer obtains the state of the *Kiln*.

2.2. Towards DSML-based Reuse Environments

In our approach a DSML is defined by a *domain metamodel* expressed in terms of (specification-level) Role Models. The domain metamodel is used to build tool support for building models using the DSML.

When used to define a domain metamodel, a Role Model is a structure of (meta-)roles that determines a domain-specific sub-language of the UML. A role defines properties that determines a specialization of a UML metamodel class. For example, an association role determines a specialization of the UML metamodel class *Association*. The type of model elements characterized by a role is determined by its base, where a role *base* is a UML metamodel class (e.g., a class role is a role with the *Class* base). A UML model element conforms to, or plays (realizes) a role if it is an instance of the role's base and has the properties specified in the role. Such an element is also called a *realization* of the role.

An example of a (specification-level) SRM is shown in Fig. 3. In the specification form of SRMs, roles are named specifications of properties (thus role names are not parameterized). For example, *feature roles* characterize application-specific structural and behavioral 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 expressed in terms of class attributes and operations. *Structural roles* specify state-related properties that are realized by attributes or value-returning operations in a SRM role realization.

Substituting the names of realizations for the role names enclosed in the double square brackets ([[.]]) of constraint templates results in an application-specific property, called a *model-level constraint*, expressed in the OCL. 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

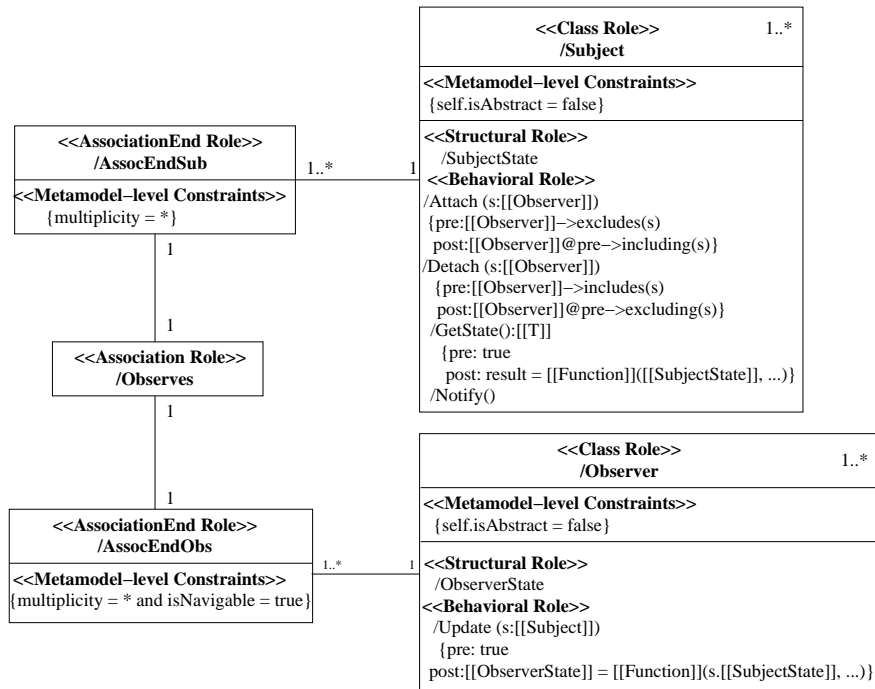


Figure 3. Example of a SRM in Template Form

role's constraint templates, and determining that the realization multiplicities associated with the feature roles are not violated [FRA 02b, FRA 01a].

An SRM determines a sub-language for UML static models. Each role in a SRM specifies a specialization of its base class in the UML metamodel. A UML static model (e.g., a Class Diagram) conforms to a SRM if static model elements that are intended to be instances of the UML metamodel class specializations defined by the roles in the SRMs have the properties defined in the SRM. This involves establishing the following :

- For each model element that is intended to play a role (i.e., is an intended instance of the UML metamodel class specialization defined by the role) the model element (i) satisfies the metamodel-level constraint (the constraint evaluates to true for the model element) and (ii) for classifier constructs, the feature roles are realized by attributes and behaviors.

- The model conforms to constraints expressed across roles in the SRM.

Pre- and post-condition templates expressed in a SRM constrain the effects of behaviors. Not all behavioral properties can be expressed in terms of pre- and post-conditions in a SRM, for example, one cannot use pre- and post-condition constraint templates to constrain how objects of realizations interact when performing a particular behavior.

A domain metamodel can also consist of IRMs that are used to constrain how objects of class role realizations interact when carrying out a realization of a behavioral role.

2.3. *ReSyDE Summary*

While repository-based approaches to reuse are common (e.g., see [GRI 93, LEW 95, ROG 97, TRA 93, WIL 99]), the reuse environment described in this paper differs from others in the following respects :

1) The development and resulting structure of the repository are based on an organization's EA. Given that an EA provides the context in which system development is carried out within an organization, its use as a basis for developing reuse infrastructures ensures that reuse activities are closely aligned with the organization's system evolution goals.

2) The UML is used as the basis for (1) representing reusable artifacts above the code level (model templates), and (3) developing DSMLs.

3) UML-based DSMLs are used to seamlessly incorporate reusable experiences into modeling notations. Use of a DSML allows a developer to build project deliverables without having to directly interface with the repository to obtain reusable artifacts. The DSML provides a language front-end to the repository that enables developers to assemble repository contents into models.

3. **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 Role Models (specifically, SRMs) can be used to support systematic pattern-based model refactoring.

3.1. *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.

A transformation algorithm : A transformation algorithm specifies how a source model is transformed to a target model.

3.2. Refactoring Using an Abstract Factory (AF) Pattern

The Abstract Factory (AF) pattern describes how one can encapsulate concerns related to creation of products with aggregate structures in classes referred to as abstract factories. The transformation outlined in this section takes a model in which complex products are created explicitly by clients and transforms it to a model that creates complex products using abstract factories. The transformation is characterized as follows :

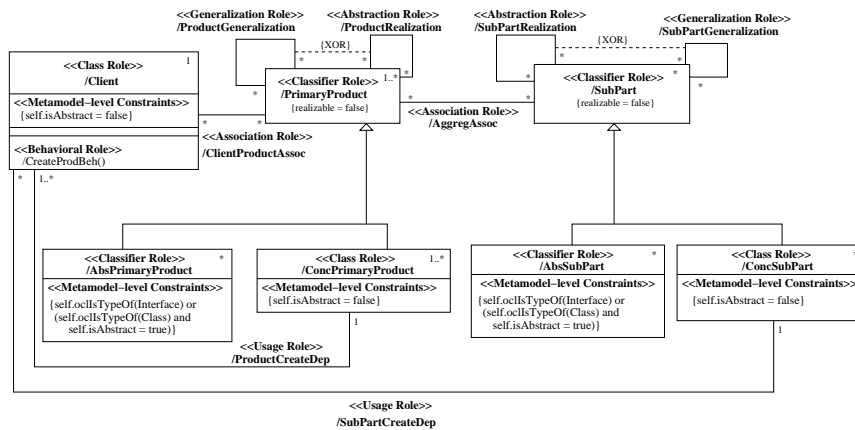


Figure 4. Source SRM for an Abstract Factory Refactoring

Source set characterization : The SRM characterizing the source models for AF refactoring we define is shown in Fig. 4. 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 can be classes in a generalization hierarchy or an interface/realization 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. 5 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 SRM characterizing the target models is shown in Fig. 6. The target model SRM defines association relationships between the client

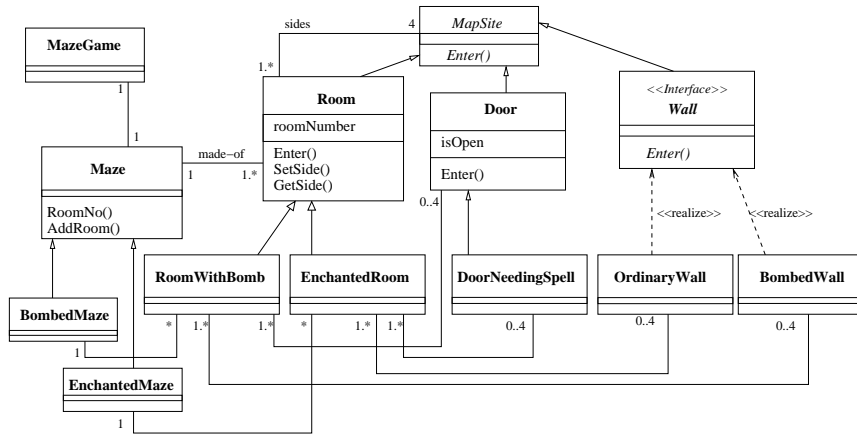


Figure 5. Maze Game adapted from the GoF book

and the product hierarchy. The relationship between the client and factory classifier hierarchies can take the form of associations or dependencies.

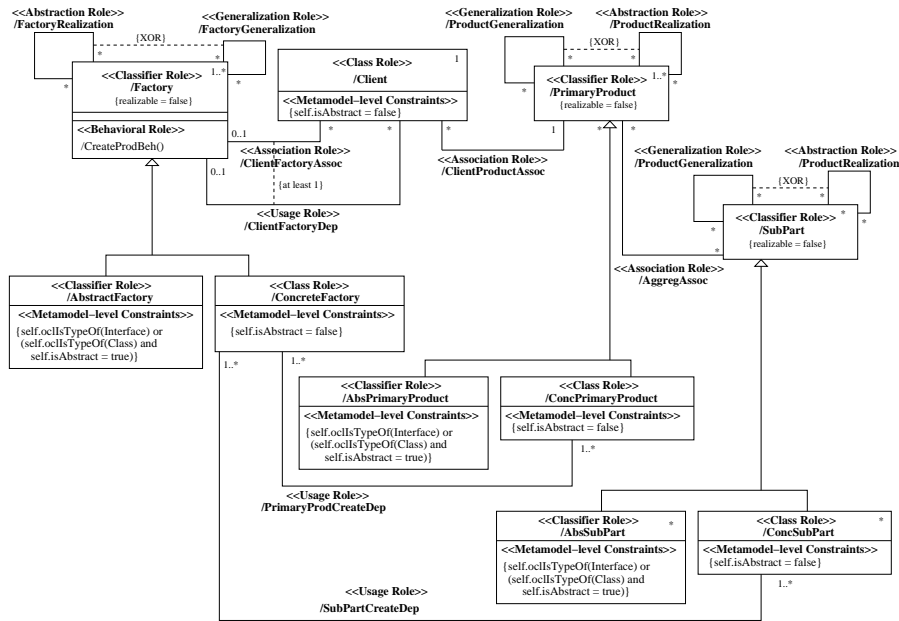


Figure 6. Target SRM for an Abstract Factory Refactoring

Transformation algorithm : The sequence of steps that accomplishes the pattern-based transformation can be defined as follows :

1) *Create the factory hierarchy.* This is accomplished by (a) 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, (b) 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 (c) 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. 5 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 [GAM 95]). In this design, the *MazeGame* (the client) 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*.

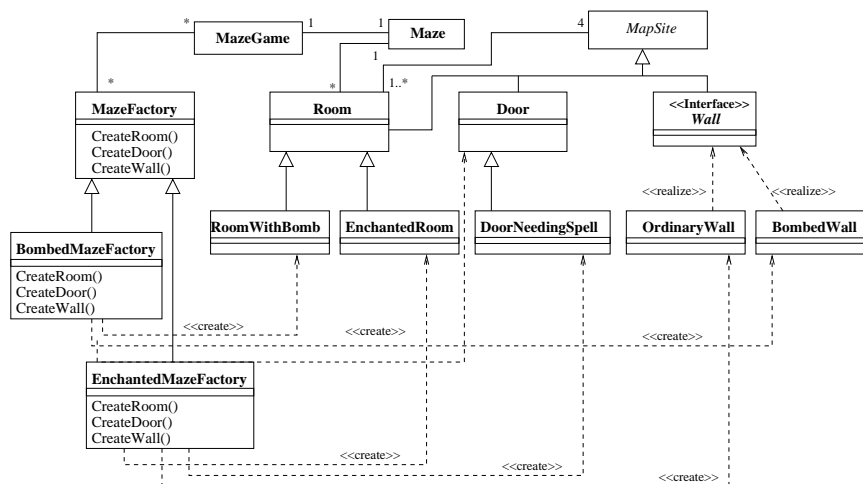


Figure 7. Maze Game refactored with Abstract Factory pattern

The model shown in Fig. 5 realizes the source model SRM shown in Fig. 4, thus we can apply the refactoring defined above to the model. The result of applying the transformation algorithm outlined above to this problem is shown in Fig. 7 :

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*.

4. Aspect-Oriented Modeling

Development approaches that support the separation of concerns software engineering principle are considered to be effective at tackling the complexity of developing software [C.G 91]. Traditional software engineering design approaches that provide guidelines and mechanisms for organizing designs around purely functional modules or classes of objects support separation of concerns along a single dimension (system functionality). Attempts at providing support for multi-dimensional separation of concerns at the programming level has given rise to aspect-oriented and subject-oriented programming [HAR 93, KIC 01, KIC 97]. An aspect is an implementation or design concern that cross-cuts the primary functional decomposition of a program. Researchers have also started to develop techniques and mechanisms that provide support for multi-dimensional separation of concerns in the design phase of software development (e.g., see [CLA 99, CLA 98, MEK 02a, SUZ 99]). The goal of our work on AOM is to provide a highly flexible environment for weaving aspects into models that leverages our work on pattern-based model refactoring. The AOM approach allows developers to encapsulate cross-cutting design concerns (e.g., a user authentication security concern) as design *aspects* [GEO 02b, GEO 02a]. In our AOM approach, a design model consists of one or more aspects and a model of system functionality called the *primary model*; the aspects represent concerns that cross-cut the design structure reflected in the primary model. A comprehensive system design model can be obtained by weaving the aspects into the primary model using a weaving strategy that is based on knowledge about dependability concerns and their interactions.

Encapsulating cross-cutting design concerns in aspects and weaving of aspects into primary models can be beneficial for the following reasons :

1) The localization of information related to a dependability concern in an aspect allows one to understand and communicate dependability concerns in their essential forms, rather than in terms of a specific system's behavior. This also facilitates the use of aspects for capturing properties and policies (e.g., security policies) that are applicable (reusable) across a family of systems.

2) High quality experiences related to defining dependability concerns and creating designs that meet dependability goals can be explicitly captured in aspects and strategies for weaving aspects into system designs. This enables reuse of expert experiences across projects.

3) Changes to a concern (e.g., changes to security mechanisms) are made in one place (the aspect), and effected by weaving the changed aspect into the system model. This eases change management.

4) The impact of dependability concerns on system functionality and other cross-cutting concerns can be analyzed by weaving aspects and system models and analyzing the resulting models. During weaving, conflicts can be identified and analysis of the weaving results can reveal undesirable emergent behaviors at the design level, thus leading to the early identification of design problems. Tool-supported weaving and evaluation enable investigation of alternative ways of defining dependability concerns and weaving that can yield insights into design trade-offs.

The primary and woven models are expressed in the UML [The 01]. Aspects are treated as patterns of structures and behaviors and are represented using the template form of Role Models. Applying (weaving) an aspect into a primary model involves modifying specified elements of the model such that they conform to the properties expressed in a Role Model. Weaving (1) adds model elements to, (2) deletes model elements from, and (3) modifies model elements in the primary model so that the resulting model conforms to the Role Model.

Weaving can be viewed as a special case of pattern-based model refactoring in which a non-conforming model is transformed to a conforming model (i.e., a model that incorporates the aspect). Weaving a Role Model into a target UML design model, M , can involve (1) merging roles with specified model elements in M , that is, modifying (including deleting) specific model elements in M so that they conform to the roles and (2) for roles that are not associated with any model element in M , generating new model elements from the roles and inserting them into M . Specifically, weaving of an aspect into a primary model is carried out as follows :

1) Map primary model elements to the roles they are intended to play : Before the weaver can incorporate the pattern (aspect) into a primary model the modeler must first indicate the parts of the model the aspect is to be woven into. This can be accomplished by the modeler explicitly indicating the model elements that are intended to play the roles. Alternatively, the aspect can characterize the points into which it will be woven (as is done with pointcuts in AspectJ). For this paper we assume that the former approach is being used. We are currently developing support for the second approach. Note that not all model elements need be mapped to roles. Also, not all roles need be associated with a primary model element. Roles not associated with primary model el-

ements indicate that new model elements must be created and added to the primary model as described later.

2) Merge roles with primary model elements : Each model element that is mapped to a role has its properties matched with the properties contained in the aspect, and additional properties are generated from the role if deficiencies in the model element are found. For example, a class that is mapped to a class role that does not have attributes or operations that play structural and behavioral roles defined in the mapped class role is extended with attributes and operations generated from the role.

3) Add new elements to the primary model : Each role that is not mapped to a model element is used to generate a model element that is then added to the model.

4) Delete existing elements from the primary model : If a model element is mapped to a *« delete »* role (a *« delete »* role indicates that conforming elements must not exist in the model), then the model element is removed from the primary model.

A system can be associated with multiple aspects : each aspect modeling a cross-cutting concern. There can be dependencies among the aspects and the order of weaving plays an important role. We are currently investigating types of dependencies among aspects in order to define weaving strategies.

We have used the AOD approach to model and weave some non-trivial fault-tolerance and security aspects into primary models [FRA 02a, MEK 02a]. We are currently developing a prototype tool that will support flexible weaving, by providing users with a language for describing (reusable) weaving strategies and weaving procedures ([MEK 02b]).

5. Conclusion

In this paper we outline some mechanisms that can enhance the creation and evolution of models. The mechanisms emphasize reuse of models in the form of template form of design patterns and DSMLs, and provides support for systematic refactoring of models and multi-dimensional separation of concerns through the use of modeling aspects.

In summary, the complexity of software development can be tackled using modeling approaches that allow developers to (1) gain insights into problems and solutions, (2) use a toolbox approach to selecting and using appropriate techniques and notations, and (3) manage complexity through support for the separation of concerns principle, and for the timely creation and evolution of models. Unfortunately, current technologies, notations, and techniques are not able to consistently support timely development of high quality models. The mechanisms outlined in this paper are intended to enhance the use of models during complex system development.

6. Bibliographie

[ARA 91] ARANGO G., PRIETO-DIAZ R., « Introduction and Overview : Domain Analysis Concepts and research directions », *Domain Analysis and Software Systems Modeling*, IEEE

- Press, 1991.
- [BAS 91] BASILI V. R., ROMBACH H. D., « Support for comprehensive reuse », rapport n° UMIACS-TR-91-23, CS-TR-2606, 1991, Department of Computer Science, University of Maryland at College Park.
- [BAS 98] BASS L., CLEMENTS P., KAZMAN R., *Software Architecture in Practice*, SEI Series in Software Engineering, Addison-Wesley, 1998.
- [BOO 99] BOOCH G., RUMBAUGH J., JACOBSON I., *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.
- [BUS 96] BUSCHMANN F., MEUNIER R., ROHNERT H., SOMMERLAD P., STAL M., *A System of Patterns : Pattern-Oriented Software Architecture*, Wiley, 1996.
- [C.G 91] C. GHEZZI M. J., MANDRIOLI D., *Fundamentals of Software Engineering*, Prentice Hall, 1991.
- [CLA 98] CLARKE S., MURPHY J., « Developing a tool to support the application of aspect-oriented programming principles to the design phase », *Proceedings of the International Conference on Software Engineering (ICSE '98)*, Kyoto, Japan, April 1998.
- [CLA 99] CLARKE S., HARRISON W., OSSHER H., TARR P., « Separating concerns throughout the development lifecycle », *Proceedings of the 3rd ECOOP Aspect-Oriented Programming Workshop*, Lisbon, Portugal, June 1999.
- [FRA 01a] FRANCE R., KIM D., SONG E., GHOSH S., « Using Roles to Characterize Model Families », KILOV H., Ed., *Practical foundations of business and system specifications*, Kluwer Academic Publishers, 2001.
- [FRA 01b] FRANCE R. B., GHOSH S., TURK D., « Towards a Model-Driven Approach to Reuse », *Proceedings of the 7th International Conference on Object-Oriented Information Systems (OOIS 2001)*, Springer, 2001.
- [FRA 02a] FRANCE R., GEORG G., « Modeling fault tolerant concerns using aspects », rapport n° 02-102, 2002, Computer Science Department, Colorado State University.
- [FRA 02b] FRANCE R. B., KIM D. K., SONG E., « Patterns as Precise Characterizations of Designs », rapport n° 02-101, 2002, Computer Science Department, Colorado State University.
- [GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.
- [GEO 02a] GEORG G., FRANCE R. B., RAY I., « Designing High Integrity Systems using Aspects », *Proceedings of the Fifth IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS 2002)*, 2002.
- [GEO 02b] GEORG G., RAY I., FRANCE R. B., « Using Aspects to Design a Secure System », *Proceedings of the International Conference on Engineering Complex Computing Systems (ICECCS) 2002*, ACM Press, 2002.
- [GRI 93] GRISS M. L., « Software Reuse : From library to factory », *IBM Systems Journal*, vol. 32, n° 4, 1993.
- [HAR 93] HARRISON W., OSSHER H., « Subject-Oriented Programming (A Critique of Pure Objects) », *Proceedings of the 8th Annual Conference on Object-Oriented Programming : Systems, Languages, and Applications (OOPSLA '93)*, Washington, D.C., September 1993, p. 411-428.

- [KIC 97] KICZALES G., LAMPING J., MENDHEKAR A., MAEDA C., LOPES C. V., LONGTIER J.-M., IRWIN J., « Aspect-Oriented Programming », *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, vol. 1241 de *Lecture Notes in Computer Science*, Jyvaskyla, Finland, June 1997, p. 220-242.
- [KIC 01] KICZALES G., HILSDALE E., HUGUNIN J., KERSTEN M., PALM J., GRISWOLD W. G., « An Overview of AspectJ », *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '01)*, Budapest, Hungary, June 2001, p. 327-353.
- [KIM 02] KIM D.-K., FRANCE R., GHOSH S., SONG E., « Using Role-Based Modeling Language (RBML) as Precise Characterizations of Model Families », *Proceedings of the Iterational Conference on Engineering Complex Computing Systems (ICECCS) 2002*, ACM press, 2002.
- [LEW 95] LEWIS T., ROSENSTEIN L., PREE W., WEINAND A., GAMMA E., CALDER P., ANDERT G., VLISSIDES J., SCHMUCKER K., *Object Oriented Application Frameworks*, Manning Publication Co., 1995.
- [MEK 02a] MEKERKE F., GEORG G., FRANCE R., ALEXANDER R., « Tool Support for Aspect-Oriented Design », *Proceedings of the OOIS Workshop on Model-Driven Approaches to Software Development*, 2002.
- [MEK 02b] MEKERKE F., GEORG G., FRANCE R., ALEXANDER R., « Tool Support for Aspect-Oriented Design », *Proceedings of the OOIS Workshop on Model-Driven Approaches to Software Development*, 2002.
- [MOR 93] MOREL J.-M., FAGET J., « The REBOOT Environment », *Advances in Software Reuse*, IEEE Computer Society Press, March 1993.
- [PRE 95] PREE W., *Design Patterns for Object-Oriented Software Development*, Addison Wesley, 1995.
- [PRI 93] PRIETO-DIAZ R., « Status Report : Software Reusability », *IEEE Software*, vol. 10, n° 3, 1993.
- [ROG 97] ROGERS G. F., *Framework-based software development in C++*, Prentice Hall, 1997.
- [SUZ 99] SUZUKI J., YAMAMOTO Y., « Extending UML with Aspects : Aspect Support in the Design Phase », *Proceedings of the 3rd ECOOP Aspect-Oriented Programming Workshop*, Lisbon, Portugal, June 1999.
- [The 99] THE OBJECT MANAGEMENT GROUP (OMG), « Unified Modeling Language », Version n° 1.3, June 1999, OMG, <http://www.omg.org>.
- [The 01] THE OBJECT MANAGEMENT GROUP (OMG), « Unified Modeling Language », Version n° 1.4, September 2001, OMG, <http://www.omg.org>.
- [TRA 93] TRACZ W., COGLIANESE L., YOUNG P., « Domain-specific SW architecture engineering », *Software Engineering Notes*, vol. 18, n° 2, 1993.
- [TRA 01] TRASK R., FRANCE R., « RIGR - A Repository Model Based Approach to Management », *Proceedings of UML Workshop on the Practical UML-Based Rigorous Development Methods*, GI-Edition, Lecture Notes in Informatics, 2001.
- [WIL 99] WILE D. S., RAMMING J. C., « Special Section : Domain Specific Languages », *IEEE Transactions on Software Engineering*, 25(3), IEEE, 1999.