

# Towards the Unification of Patterns and Profiles in UML

Petri Selonen, Kai Koskimies, Tommi Mikkonen

Institute of Software Systems  
Tampere University of Technology  
P.O. Box 553, FIN-33101 Tampere, Finland  
{petri.selonen, kai.koskimies, tommi.mikkonen}@tut.fi  
<http://practise.cs.tut.fi>

**Abstract.** Patterns have become a popular means to express recurring software solutions, as exemplified by design patterns. On the other hand, so-called profiles are used in UML to define various kinds of domain-specific architectural concepts and conventions as extensions of the UML metamodel. In this paper, we show how patterns and profiles can be unified in the UML context, using the UML metamodel as the common basis. We argue that this result has far-reaching implications on tool development, and helps us to understand the relationships of these two central software architecting concepts.

## 1 Introduction

The concept of a pattern has emerged as a response to the need to specify generic solutions to certain recurring software construction problems. In this context, a pattern is a description of an organization of abstract, system-independent software elements that provide a general solution to a problem in a context. A pattern can be instantiated in a particular system, which effectively binds the abstract software elements (so-called *roles*) of the pattern to concrete software elements in a particular system. Examples of patterns include analysis patterns [6], architectural patterns [1], design patterns [7] and coding patterns [3]. In a sense, a pattern can be regarded as a generic software unit comparable to, say, templates in C++, with roles corresponding to the template parameters. However, unlike templates a pattern is liberated from the primary decomposition of the system: a pattern can crosscut the components of the system in an arbitrary way.

In this paper we assume that a pattern is instantiated in a UML [10] model; thus, the roles of a pattern are bound to UML model elements. Unfortunately, standard UML provides only modest support for patterns: a special model element (collaboration symbol) can be used to mark an instance of a pattern in a model, with lines indicating the concrete model elements playing roles in the pattern. Consequently, efforts have been made to improve support for patterns in UML ([2], [4], [12]).

However, patterns are not always the most obvious vehicle for expressing architectural specifications. There is often a need to introduce various kinds of general conventions or rules, which are assumed to hold throughout the system. In particular, such rules are frequently imposed by the architectural decisions made in a particular software system, platform or domain. For example, a rule might state that because of a chosen architectural style, certain kinds of components may depend on each other only in a certain restricted, predefined way. In practice, such rules are typically enforced by coding standards given to system developers. In contrast to patterns, these rules are

system-wide, and their purpose is to preserve the architectural hygiene of the system rather than to offer a local solution to a specific problem.

In UML, a set of architectural rules can be presented as a so-called *profile*. A profile is an extension of the UML metamodel, specifying a subset of UML models that can be considered “legal” in a particular context (say, in a particular domain or software platform). In [13] a technique is presented for specifying a set of architectural rules as a UML profile, to be used for checking a given UML model against such rules. The characterization of a set of architectural rules as a UML profile (that is, as a metamodel extension) suggests that such a set of rules can in fact be viewed as a more restrictive modeling language the designer must follow. This is in sharp contrast with the notion of a pattern, which is seen as a specific arrangement of software elements that is voluntarily created by the designer.

In spite of their different character and purposes, patterns and profiles have much in common. Both are based on identifying certain model element categories, and on specifying relationships between these categories. In profiles a category consists of the instances of a new model element type added to the metamodel (so-called stereotype), while in patterns a category consists of the model elements playing a particular role in a pattern. In both cases, various kinds of semantic (or behavioral) information can be associated with the categories, originating from the problem context of a pattern or from the domain of a profile (like, say, the interaction protocol associated with a role or the domain-specific “meaning” of a stereotype).

Our thesis is that patterns and profiles can (and should) be unified. The main benefit of such a unification is that it would allow for a single tool that can be used to define patterns and profiles in UML modeling, supporting both the building of a model according to a pattern/profile and the checking of the conformance of a model with a pattern/profile. Moreover, we feel that it is important to find the common ground of software architecting concepts originating from rather different motivations, paving the way for conceptual convergence.

In this paper we take the first step towards the unification of patterns and profiles by demonstrating how a pattern can be interpreted as a profile and vice versa, assuming a particular (but reasonable) interpretation of these concepts. We outline the basic idea in terms of examples; we will not present a fully elaborated technique here but leave this for future work. In the scope of this paper, we will restrict ourselves to the structural properties of patterns and profiles, leaving behavioral issues out of the discussion.

The rest of this paper is structured as follows. Sections 2 and 3 introduce the concepts of profiles and patterns as we will treat them in this paper. Section 4 shows how to derive profiles from patterns, and Section 5 discusses how to create patterns out of profiles. Section 6 introduces some related work, and Section 7 concludes the paper with a discussion. Throughout the paper, we assume familiarity with UML.

## 1. UML and Profiles

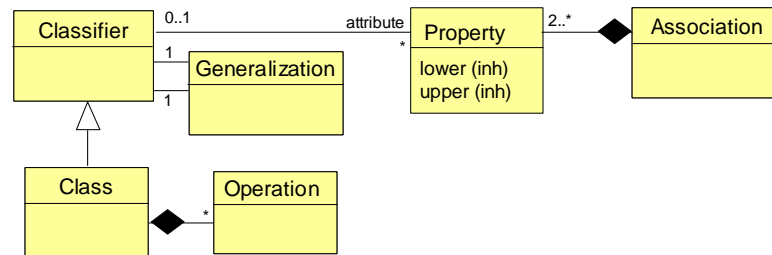
UML<sup>1</sup> has quickly emerged as lingua franca of software architecting. Although UML has been criticized for its lack of formal definition, its structural aspects are specified formally by the UML metamodel. This specification is given as UML class diagrams (strictly speaking, these class diagrams are given using the facilities provided by the

---

<sup>1</sup> While the basic idea does not rely on a particular version of UML, we exploit some of the notations of UML 2.0 to facilitate the presentation.

metamodel, including a subset of UML class diagram notation), associated with a set of well-formedness rules. From now on we will refer to these class diagrams as the UML metamodel. The classes appearing in the metamodel are called metaclasses, and the associations in the metamodel are called meta-associations.

As an example of the UML metamodel, Figure 1 depicts a simplified fragment of the metamodel, defining how classes, associations, and operations can be combined. Note that the metamodel requires that each association (class Association) has at least two association ends (class Property), and that each association end is attached to exactly one classifier (class Classifier, a generalization of class Class). Each class may in turn contain an arbitrary number of operations (class Operation). Two classifiers can be in a generalization relationship (class Generalization) with each other.



**Fig. 1.** A fragment of the UML metamodel

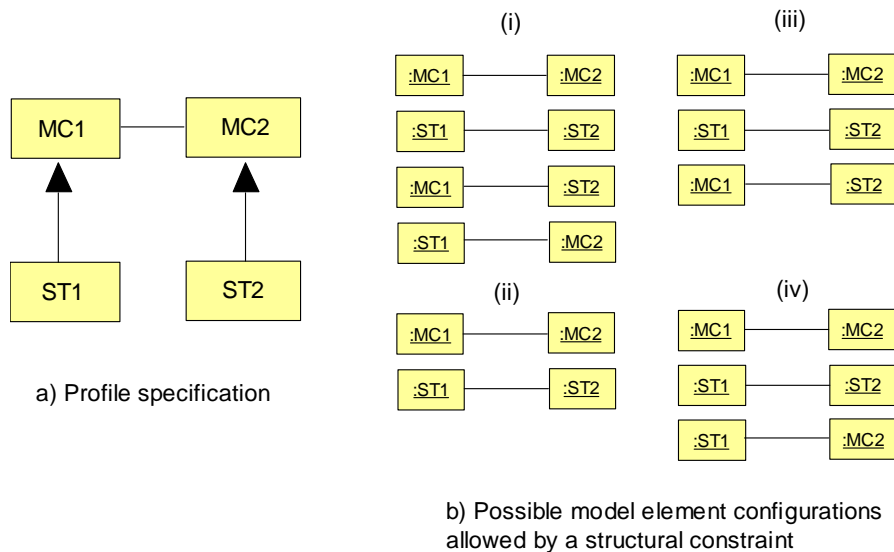
A profile is, by definition, an extension of the UML metamodel. Typically profiles are used to customize UML for a particular domain by introducing additional domain specific modeling concepts and constraints. The main mechanisms available for specifying profiles are stereotypes, constraints and tagged values. UML defines rather vaguely that a stereotype is “a limited kind of metaclass that cannot be used by itself, but must always be used in conjunction with one of the metaclasses it extends” [10]. A stereotype can be added to the metamodel as a new class symbol which *extends* (a black-headed arrow, see Figure 2) an existing metaclass. Extension is a kind of an aggregation where the properties of the metaclass are extended through the stereotype, allowing a stereotype instance to be treated also as an instance of the metaclass without the stereotype.

The idea of stereotypes is to allow the introduction of slight extensions of the concepts of the metamodel, rather than entirely new kinds of modeling elements (which should be defined using MOF, the Meta Object Facility). Thus, a profile cannot change the existing properties of the model elements, but only add extended element types (stereotypes), their new properties (tagged values) and constraints. Consequently, tools can always manipulate instances of stereotypes as instances of their base element types, although some tools may exploit the additional properties defined for the stereotype.

Constraints can be given formally as OCL (Object Constraint Language) expressions over instances of the stereotypes, giving additional requirements for those instances. Here we will use constrained associations between stereotypes to express certain structural restrictions in a natural way. Strictly speaking, new associations are allowed in a profile only if the associations are subsets of existing meta-associations [10]. However, we use only additional associations that specialize an existing meta-association for a pair of stereotypes. Thus, the new associations do not introduce any new structural

relationships; they merely stand for existing relationships when applied to certain stereotypes. In our view, this does not conflict with the idea of a profile in UML, since any model that is an instance of the profile is also an instance of the metamodel. The profile only rules out certain configurations of (stereotyped) model elements.

Let us study the required structural constraints in more detail. Assume two stereotypes ST1 and ST2 have been introduced as extensions of two metaclasses MC1 and MC2 connected by a meta-association, as depicted in Figure 2a. With respect to allowed configurations of model elements there are four choices for structural constraints, given in Figure 2b: (i) there are no restrictions on the structural composition of the instances of the involved metaclasses and stereotypes beyond those implied by the metamodel, (ii) the structural relationship represented by the meta-association can exist for two metaclass instances or two stereotype instances, but not for mixed combinations, (iii) in addition to (ii), an instance of the first metaclass can be linked with an instance of the second stereotype, and (iv) in addition to (ii), an instance of the first stereotype can be linked with an instance of the second metaclass.



**Fig. 2.** Profile specification and possible model element configurations

To express the structural constraints given above, we use an OCL template of the form<sup>2</sup>:

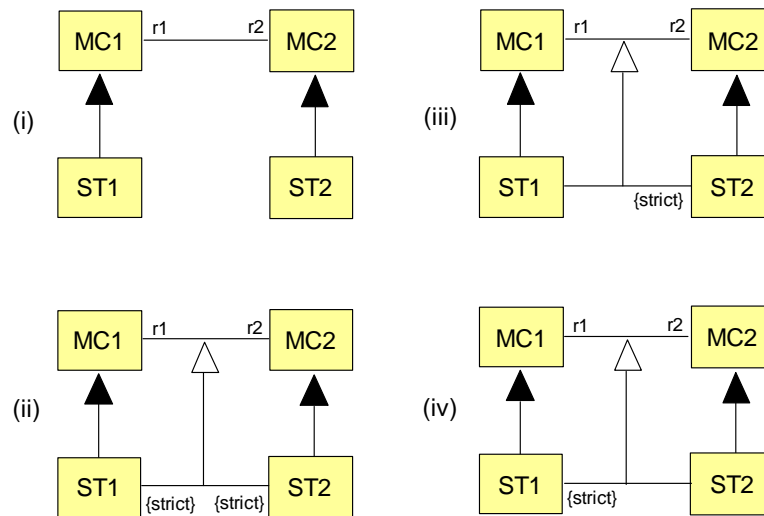
```
template <metaclass MC, navigation NAV, stereotype ST>
context MC
inv:
  self.NAV ->foreach(stereotype->includes(ST))
```

The constraint template is instantiated (by replacing the template parameters MC, NAV, and ST2 with actual names) and attached to a stereotype appearing in a profile. Thus, the context of the constraint (that is, “self”) is actually an instance of a stereotype (extended from metaclass MC), since the constraint will be applied only to the instances of the stereotype. The constraint simply says that the model element at the NAV-end of a link must be an instance of stereotype ST.

<sup>2</sup> Actually OCL does not allow templates, but we use here a C++ -like template notation to facilitate the presentation.

We will name this constraint “strict”. We assume that the location of this constraint in the profile model determines the template parameters and thus instantiates a concrete form of the constraint with actual names taken from the profile model. Instead of attaching this constraint visually to a stereotype symbol itself, we attach it to an association end owned by that stereotype. In this way the template parameters get fixed: NAV will be the role name (or target class) at the other end of the association, ST is the stereotype whose instance should be found at that end of the link, and MC is the metaclass of the stereotype at the constraint end.

Using constraint {strict} in the way described above, we can now express a profile specification yielding one of the structural constraints (i) – (iv) above, as depicted in Figure 3. Informally, {strict} means that only this kind of model element (that is, a model element of the associated stereotype) can appear at this end of the link.



**Fig. 3.** Specifying different structural constraints

In cases (ii) – (iv), we require that the specialized association between stereotypes does not introduce new names for the association or its end roles. Further, we require that the multiplicities of the original meta-association are followed in the instantiation in all cases (i) – (iv). If several specialized associations share the same base meta-association, the sums of the multiplicities of the “subassociations” must be subsets of the multiplicities of the base meta-association.

In a model, an instance of stereotype *S* in a model is denoted by <<*S*>> preceding the name of the model element. Whenever such an instance appears in a model, we assume that the constraints given for the stereotype are checked. We also assume that only defined stereotypes are used in the model.

As an example, consider a situation where the architect has chosen a client-server architectural style, and wants to enforce this style in the design. A profile specifying the rules implied by this decision is given in Figure 4. The profile states that if a class stereotyped as <<Client>> is used in a model, there must be another class stereotyped as <<Server>>, and there must be exactly one association between these classes stereotyped as <<service>>. The profile states also that a <<service>> association may appear only between a <<Client>> and a <<Server>> (note the use of the {strict} constraint at the Client and Server ends of the associations). However, the profile allows

other, arbitrary associations between a <<Client>> and a <<Server>> as well, and associations between a <<Client>> (or a <<Server>>) and some other classes, since the ServerEnd and ClientEnd ends of the associations are not constrained.

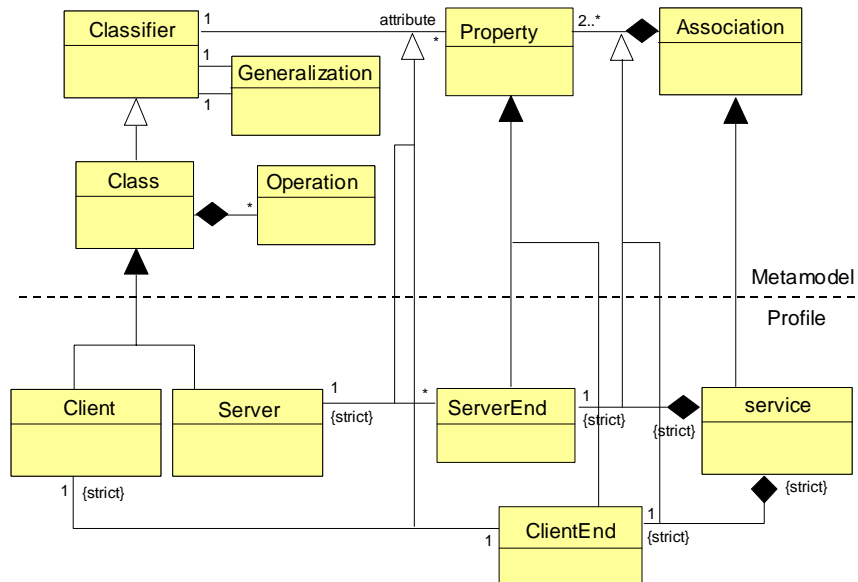


Fig. 4. A UML profile specification for client-server architectural style

Checking a design model against architectural profiles such as the ones in Figure 4 has turned out to be useful in practice [13]. Often it makes sense to introduce a fairly large set of system or domain specific class categories as stereotypes, and specify the allowed dependencies between instances of these categories. If a model consists of hundreds of classes, which is often the case in real life, manual checking of the model would become very cumbersome.

## 2. Patterns

We define a pattern as an arrangement of software elements for solving a particular problem. Depending on the nature of the problem, we may speak of analysis patterns, architectural patterns, design patterns etc. In the sequel we will give a simple technical characterization of a generic pattern concept from the purely structural viewpoint. Thus, we ignore aspects like behavior or purpose which are of course important ingredients of, say, design patterns, but which are not an issue here.

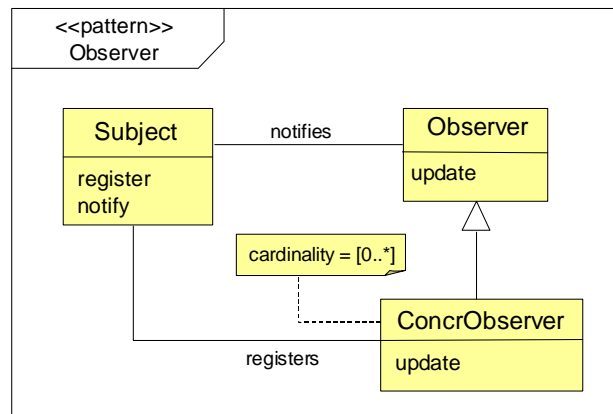
To be able to define a pattern independently of any particular system, a pattern is defined in terms of element *roles* rather than concrete elements; a pattern is instantiated in a particular context by binding the roles to concrete elements. A role has a type, which determines the kind of software elements that can be bound to the role; the set of all the role types is called the *domain* of the pattern. Here we assume that the domain of a pattern is UML; that is, all the roles are bound to UML model elements.

Each role may have a set of *constraints*. Constraints are structural conditions that must be satisfied by the model element bound to a role. For example, a constraint of

association role  $P$  may require that the association bound to  $P$  must appear between the classes bound to certain other roles  $Q$  and  $R$ . A *containment constraint* requires that the element bound to the role must be contained by the element bound to another role. For example, a containment constraint of operation role  $S$  may require that the operation bound to the role is contained by the class bound to class role  $T$ .

A *cardinality* is defined for each role. The cardinality of a role gives the lower and upper limits for the number of the instances of the role in the pattern. For example, if an operation role has cardinality  $[0..1]$ , the operation is optional in the pattern, because the lower limit is 0. The default cardinality is  $[1..1]$ .

As an example of a pattern, consider Observer [7]. Slightly simplifying, the structure of this design pattern could be presented in UML as depicted in Figure 5. We assume there is a standard stereotype `<<pattern>>`: a diagram (rectangle with a name in the upper left corner) stereotyped as `<<pattern>>` should be interpreted as a pattern specification rather than as part of a concrete model. The pattern interpretation of the diagram implies that every model element in the diagram represents a role; that is, a placeholder for a concrete model element. Thus, in Figure 5 there are roles for classes, operations, associations etc. A role can be unnamed, like the generalization (inheritance) relationship role. The type of a role is the metaclass of the corresponding model element. For example, the type of role “notify” is the metaclass Operation.



**Fig. 5.** Observer design pattern (simplified)

The class diagram in Figure 5 looks like an ordinary class diagram. The only place where we need additional notation (except for the `<<pattern>>` stereotype) is the specification of the cardinalities of roles. In the Observer pattern, we want to be able to provide several concrete observers derived from the same abstract observer. However, this is difficult to express naturally in UML: multiplicities have different meaning in UML. Therefore we specify the cardinality of a role (ConcrObserver) with a separate comment.

The pattern interpretation of a class diagram implies certain structural constraints for the roles:

- Each operation role has a containment constraint forcing the operation bound to the role to appear within the class bound to the enclosing class role.
- An inheritance role appearing between two class roles has a constraint requiring that the classes appearing at the ends of an association bound to this role must be bound to the two class roles.

- Similarly, if an association role appears between two class roles, the association role has a constraint requiring that this role must be bound to an association between classes bound to the class roles.
- If a class role inherits another class role, and both class roles contain an operation role with the same name, the operation role in the inheriting class role has a constraint requiring that the operation bound to this role must have the same signature as the operation bound to the role contained by the superclass role.

In the example, the syntactical consequences of these constraints are that “update” operations can appear only in classes playing the Observer and ConcrObserver roles, and that “register” and “notify” operations appear only in a class playing the Subject role. Further, “notifies” association may appear only between classes playing the Subject and Observer roles, and “registers” association only between classes playing the Subject and ConcrObserver roles. In each instance of the pattern, there can be an arbitrary number of classes class playing the ConcrObserver role. Finally, each class playing the ConcrObserver role must inherit a class playing the Observer role.

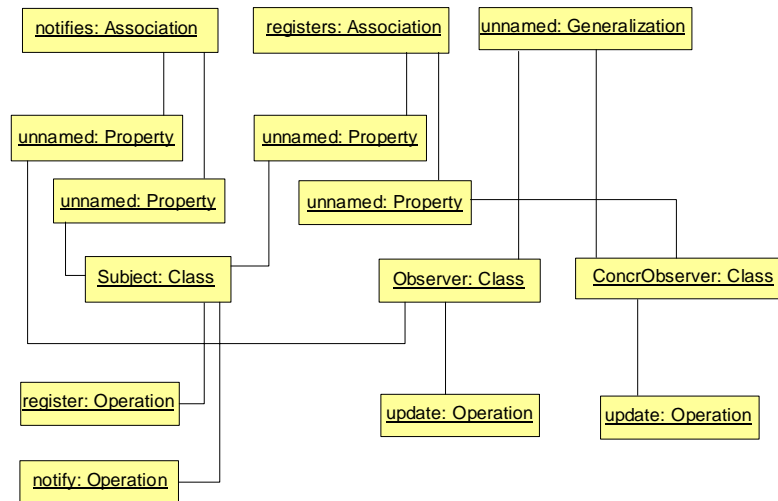
This kind of structural pattern concept can be used as a basis of fairly powerful tool support for pattern-driven software development. In [8], the general idea is that the specialization interface of an application framework (in Java) is described using such patterns, with some of the roles bound to the framework’s elements. The tool is able to maintain a task list for binding the application dependent roles. The task list displays a task for each role that can be instantiated and bound in the present situation, taking into account the dependencies implied by the constraints and the multiplicities of the roles. The tool keeps track of the constraints and generates additional tasks to correct constraint violations. The patterns can be augmented with practically useful information, like templates for default elements to be bound to a role (allowing the generation of the bound element), intuitive task prompts, informal explanations of the tasks etc. Recently, the tool has been extended to support UML based design in Rational Rose [11]. The patterns, however, are specified in that tool using a dedicated graphical presentation rather than UML.

In the following two sections, we shift the focus to the actual unification of profiles and patterns, and show how to translate patterns to profiles and vice versa using UML metamodel as the common language.

### 3. From Patterns to Profiles

To understand the presentation of a pattern as a profile, consider again the pattern in Figure 5. Every model element in the diagram represents a role, which can be played by several model elements in an actual system model. Thus, each model element in the pattern represents a category of concrete model elements. This suggests that each model element (or role) in the pattern should be viewed as a stereotype. The base class of the stereotype is the element type of the role, e.g., for a class role, the base class is Class, for an association role, the base class is Association etc.

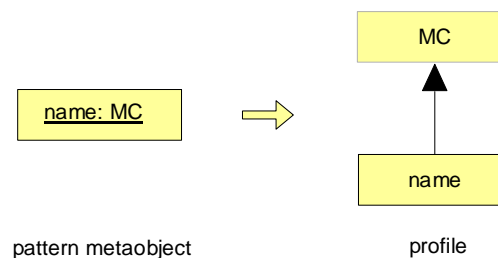
However, there are three kinds of model elements in the model of Figure 5: named explicit model elements (e.g., Subject), unnamed explicit model elements (e.g., the inheritance relationship), and implicit model elements (e.g., the association end elements of the associations). To really see all the model elements and their relationships we must construct the object representation of the model, according to the metamodel. This is shown in Figure 6. Note that the rectangles are now instances of the metaclasses (*metaobjects*), and the connecting lines are links (instances of associations).



**Fig. 6.** Object representation of the pattern specification in Figure 5

To turn this pattern into a profile, we must transform the metaobject representation into an extension of the metamodel. This can be done in a relatively simple way. We already noted that each model element in the pattern should give rise to a stereotype. Thus, we introduce a stereotype for each metaobject appearing in the object representation of the pattern. The base class of the stereotype is the class of the metaobject, and the name of the stereotype is the name of the metaobject. To guarantee unique names of the stereotypes we consider class roles as namespaces for the contained roles (e.g., operation roles). The transformation rule is depicted in Figure 7.

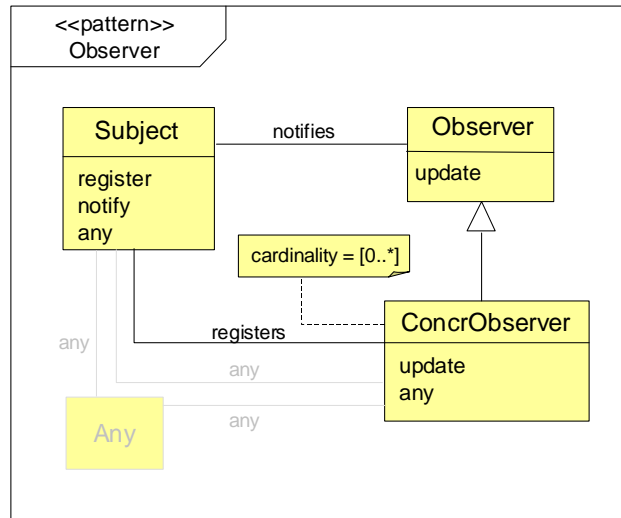
Each link in the metaobject representation of the pattern is an instance of a distinct meta-association or its specialization. Thus, for each link there should be either a distinct, existing meta-association between the corresponding metaclasses, or a specialization of such an association between two stereotypes. The latter is needed only if the related pattern rule excludes some structural combinations that are allowed by the pure metamodel. In that case we need additional transformation rules to create the specialized (constrained) associations in the profile specification.



**Fig. 7.** Transformation rule for stereotypes

To allow the pattern provider to give the information concerning the structural constraints to be attached to the profile, we propose the following convention. A pattern model may include elements which are named as “any”. These could be, for instance, classes, operations, attributes, associations etc. The general rule is then that anything not explicitly allowed by the profile is forbidden. If the pattern provider wants to relax

certain structural constraints imposed by the pure pattern description, she can add “any” elements meaning that in any legal UML element can appear in the structural position of the “any” element, not just the particular kinds of elements denoted with a specific name (and later transformed into stereotypes).



**Fig. 8.** A revised representation of the Observer pattern, with information relaxing structural constraints

Using this convention, the Observer pattern could be given as presented in Figure 8. This specification implies that both a subject and a concrete observer may have any other types of operations in addition to those specific to the pattern, and that these classes may also have any types of association with each other and with other classes, in addition to the given ones. However, the pattern does require that “notifies” and “registers” associations may appear only between the participants of the pattern in the specified way.

The additional information relaxing the strict interpretation of the structural constraints implied by a pattern description is of course out of the scope of the pattern concept itself: here a pattern description clearly gets some of the flavor of a profile. We argue that this is mostly a tool issue: a tool can hide this information, when a pure pattern view is desired. When the information is shown, it should be visually distinguishable from the rest of the pattern, as in Figure 8.

Assuming a pattern description following this convention, the metaobject representation will include the corresponding “any”-objects as well. Based on this, we can express the transformation rules producing the desired constraints in the profile. These rules are given in Figure 9. The crossed model elements denote missing elements.

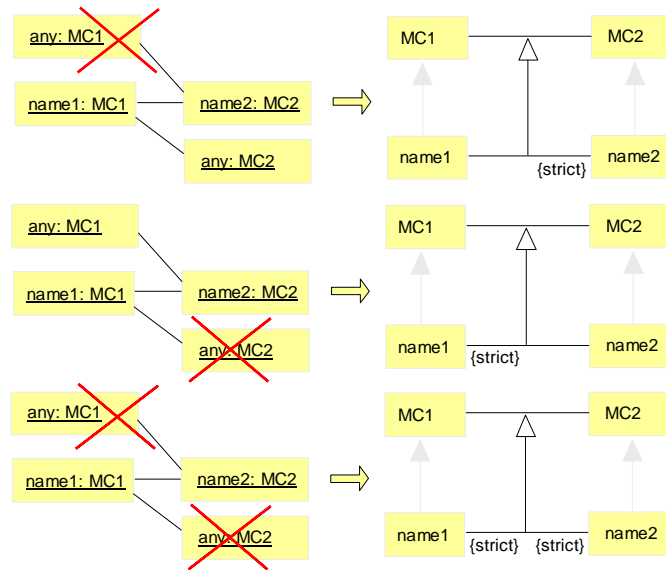


Fig. 9. Transformation rules for constrained associations

As an example of the application of the transformation rules for associations, consider roles update (of Observer) and Observer. Their presentation in a profile is depicted in Figure 10.

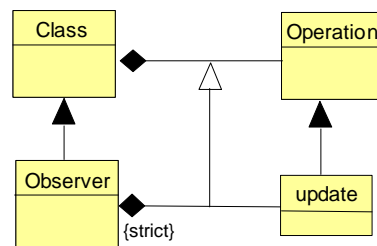


Fig. 10. Result of the application of the transformation rule for associations

Cardinalities of roles are somewhat trickier to transform into a profile representation, and we will not give any general rule here. The principle is illustrated by the example pattern, where role ConcrObserver has cardinality  $[0..*]$ . Cardinalities are given to class roles with respect to the relationships the role has to other class roles. In this case, ConcrObserver has an association relationship to Subject and an inheritance relationship to Observer. Thus, for each class playing the role Subject and for each class playing the role Observer, there can be an arbitrary number of classes playing the role ConcrObserver. Naturally, the relationship instances should be multiplied as well. Therefore we should locate the stereotypes representing the relationships, and attach the multiplicity to the associations leading to the stereotypes Subject and Observer in the profile.

Finally, additional conventions used in a pattern description have to be included in a profile in the form of additional dependencies between the stereotypes. For example, the operation overriding convention (see previous section) can be expressed as a dependency from ConcrObserver's "update" to Observer's "update", with a constraint that enforces the matching of the signatures of the operations bound to these roles.

The entire profile extracted from the Observer pattern becomes a fairly complex class diagram, and we will not present it here. However, it should be clear that we can express all the structural requirements of the pattern, as discussed in the previous section.

An instance of the pattern as it would appear in a model is presented in Figure 11. The use of the conventional pattern instance symbol in UML (collaboration ellipsis) is not necessary here since the stereotypes indicate both the existence of a pattern instance and the roles different parts in the model are playing in the pattern. However, in a real model where stereotypes are used for other purposes as well and where there may be several instances of a pattern, a separate symbol marking a pattern instance is needed.

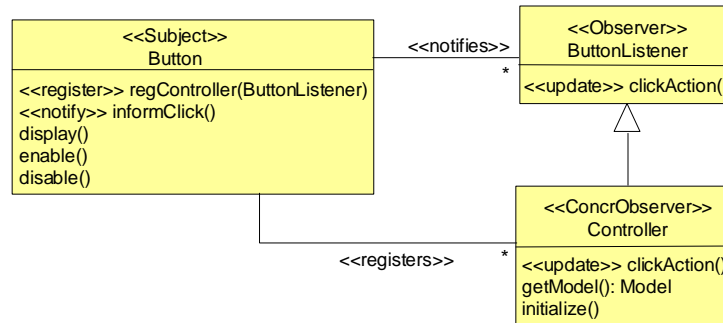
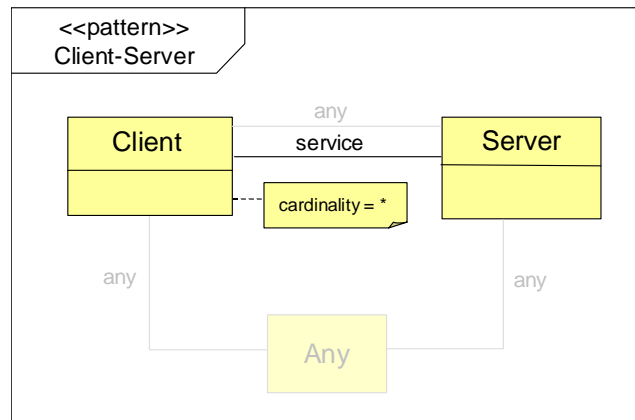


Fig. 11. An instance of the Observer pattern

#### 4. From Profiles to Patterns

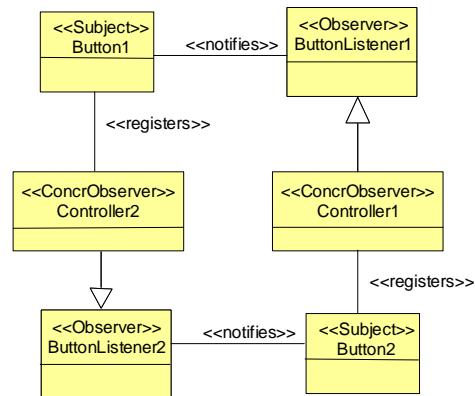
The transformation process described in the previous section can be reversed, allowing us to produce a pattern model from every profile constructed according to our principles. According to these principles, stereotypes can be introduced for any metaclass, and structural constraints can be expressed by specializing existing meta-associations. The idea is simple: if we interpret every stereotype class in the profile as an instance of the corresponding metaclass, and consider the associations between the stereotype classes as links between these instances, we get a valid instance of the metamodel. This follows from the fact that the associations between stereotype classes are only specializations of the meta-associations between the corresponding metaclasses. Thus, there must be a visual presentation of this model instance. If we put this visual presentation inside a <<pattern>> diagram rectangle, we have the proper patternized description of the profile. Note that the transformation rules of Figure 9 must be reversed to generate the “any” instances according to the use of the {strict} constraint in the profile. For example, the result of transforming the profile in Figure 4 into a pattern is presented in Figure 12.

Again, the handling of multiplicities appearing in the profile specification is less straightforward. However, assuming that multiplicities are used only in a restricted manner, which guarantees that the multiplicities can be transformed into cardinalities of certain roles in the profile, multiplicities can be allowed in a profile without hampering the transformation process. Similarly, the use of special dependencies with constraints must be limited to some standard, known set.



**Fig. 12.** Patternized profile

It should be emphasized that this transformation produces only a model fragment, and we assume that the `<<pattern>>` stereotype gives this fragment the correct interpretation of a pattern (to be used, for example, by a tool for instantiating the pattern). Similarly, the reverse transformation from patterns to profiles produces only a profile which can be used as a basis for checking certain structural constraints implied by a pattern, but not for locating instances of patterns. The notion of a pattern instance cannot be captured by the simple profile specification mechanism we use. For example, the model of figure 13 would be accepted as a legal model according to the Observer profile, although there are no valid instances of pattern Observer in the model (we have omitted the operations).



**Fig. 13.** An accepted model without legal pattern instances

## 5. Related Work

Patterns and profiles have been previously discussed in the same context in [12], where a UML profile is derived to support the presentation of design patterns. However, their approach is different from ours in the sense that they propose a general profile for all possible patterns, while we consider a single pattern as a profile. Their purpose is not to unify patterns and profiles, but rather to show what kind of a profile is needed to support patterns in UML based software development in general. Similarly, Fontoura and Lucena [4] propose UML extensions (mostly tagged values and constraints) to support the presentation of flexibility points of design patterns. They also present a technique to specify the supposed instantiation process for such a pattern as an activity diagram. A general profile for specifying the extension interface of an application framework has been presented in [5], exploiting tagged values.

Perhaps closest to this work is the technique described in [13], where special notational conventions are used to express certain structural requirements for a UML model. Using these conventions, the architect can specify structural rules as a profile. Integrated with Rational Rose, the technique allows the checking of a design model against the architectural rules of a, say, platform or product-line. The rules in a profile are given in the style of examples of the required structure. In some cases such examples come close to a pattern-like representation, although the forms allowed in [13] are more restricted and do not rely on the pattern concept. From this viewpoint, our work could be seen as a generalization and systematic derivation of a profile description language.

## 6. Discussion

From the methodological viewpoint, the unification of patterns and profiles opens up new possibilities. In particular, the architect can conveniently describe a profile as a collection of patterns, to be followed by the designers. Indeed, the technique described here allows, at least in principle, the interpretation of *any* model fragment as a profile, taken from any diagram type (ignoring the need for different levels of strictness of the rules). For example, a sequence diagram or a state diagram can be interpreted as a behavioral profile. We anticipate that this kind of “profilization” of models can be very useful especially in the context of product-lines: the architect can give a prototypical model of an application based on the product-line, and turn this into a profile to be followed by the application developers. Presenting a profile as a set of patterns is attractive from the comprehensibility viewpoint as well: an example-like pattern is much easier to understand than the fairly cryptic metamodel extension form.

From the viewpoint of UML tools, our work allows the integration of the tool support for patterns and profiles, which has been the primary motivation for this work. On one hand, we have developed an environment for pattern-driven software development, where the patterns guide a system developer and generate software artifacts ([8], [9]). On the other hand, our team has experimented with a tool set for specifying structural profiles for the purpose of model checking [13]. We anticipate that the results presented here allow us to combine the benefits of these different tools: a single tool environment, based on a unified pattern-profile concept, can both assist in the production of models according to predefined rules, and in the checking of the models against profiles. Furthermore, this approach allows UML itself to be used in a natural way as the description language of pattern-profiles.

We believe our existing tools reflect the purposes of patterns and profiles according to their original intentions. Patterns have been the means to describe individual design

solutions, whereas profiles have been understood as domain specific refinements of the modeling language. Being able to express patterns and profiles in a unified fashion is the first step in our attempt to integrate the pattern and profile features of our tools for a true unified architecting environment.

*Acknowledgements.* This work is supported financially by the Academy of Finland (projects 51005 and 51528) and by the National Technology Agency of Finland (project 40226/04). We wish to thank Mika Siikarla for suggesting the use of the OCL template in profiles, and professor Tarja Systä for many useful discussions.

## References

1. Buschmann et al.: *Pattern-Oriented Software Architecture*, Vol 1: A System of Patterns. Wiley 1996.
2. Dong J., Yang S.: Visualizing Design Patterns with a UML Profile. *Proc. of the IEEE Symposium on Visual/Multimedia Languages (VL)*, Auckland, New Zealand, October 2003, 123-125.
3. Coplien, J. O.: Idioms and patterns as architectural literature. *IEEE Software*, 14(1):36-42, January 1997.
4. Fontoura, M. F., and Lucena, C. J.: Extending UML to improve the representation of design patterns. 12-19, *Journal of Object-Oriented Programming (JOOP)*, 13(11), March 2001.
5. Fontoura, M.F., Pree W., Rumpe B.: UML-F: A Modeling Language for Object-Oriented Frameworks. *Proc. 14<sup>th</sup> European Conference on Object-Oriented Programming (ECOOP 2000)*, Springer LNCS 1850, 2000, 63-82.
6. Fowler M.: *Analysis Patterns: Reusable Object Models*. Addison.Wesley 1996.
7. Gamma, E., Helm, R., Johnson, R., and Vlissides J.: *Design Patterns*. Addison Wesley, 1995.
8. Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., and Viljamaa J.: Generating application development environments for Java frameworks. *Proc. 3rd International Conference on Generative and Component-Based Software Engineering (GCSE'01)*. Springer, LNCS 2186, 2001, 163-176.
9. Hammouda I., Koskinen J., Pussinen M., Katara M., and Mikkonen T.: Adaptable Concern-based Framework Specialization in UML. To appear in *Proc. ASE'04*, Linz.
10. Object Management Group: UML 2.0 Specification, On-line at <http://www.omg.org/uml>.
11. Rational Software Corporation: Rational Rose, 2001. On-line at <http://www.rational.com>.
12. Sanada Y., Adams R.: Representing Design Patterns and Frameworks in UML – Towards a Comprehensive Approach. *Journal of Object Technology*, 1(2), July-August 2002, 143-154.
13. Selonen, P. and Xu, J.: Validating UML models against architectural profiles. *Proc. 9th European Software Engineering Conference (ESEC 2003)*, 2003.