Precise Modeling of Design Patterns in UML

Jeffrey K. H. Mak
Department of Electronic and Information Engineering
The Hong Kong Polytechnic University
jeffreym@eie.polyu.edu.hk

Clifford S. T. Choy
The Multimedia Innovation Center
The Hong Kong Polytechnic University
mccliff@polyu.edu.hk

Daniel P. K. Lun
Department of Electronic and Information Engineering
The Hong Kong Polytechnic University
enpklun@polyu.edu.hk

Abstract

Prior research attempts to formalize the structure of object-oriented design patterns for a more precise specification of design patterns. It also allows automation support to be developed for user-defined design patterns in the future CASE tools. Targeting to a particular type of automation (e.g. verification of pattern instances), previous specification approaches over-specify pattern structures to a certain extend. Over-specification makes pattern specification ambiguous and disallows the specification language to be used for specifying compound patterns. In this paper, we present the structural properties of design patterns which reveal the true abstract nature of pattern structures. To support these properties so as to solve the over-specification problem, we propose an extension to UML 1.5 (basically UML 1.4 with Action semantics). The specialization and refining mechanism of UML provides also a smooth support for the instantiation, refinement and integration of pattern structures specified in UML. Our work makes no significant extension to the UML 1.5 meta-model but more in a UML Profile approach to ease the migration of our work to UML 2.0, which has not yet officially released by OMG during this work.

1. Introduction

Design patterns have increasingly gained acceptance not only as reusable constructs for software development but also the documentation and comprehension of the architectural design of a software system. While software teams and companies maintain their own set of design patterns, automation support to the utilization of design patterns is still very limited. Current researchers seek automation support to different pattern activities in three main aspects. As stated by Florijn et al. [16], CASE tools may provide assistance to (1) apply design patterns; (2) validate pattern implementations; and (3) discover pattern instances for system comprehension and documentation.

Recent research [13,15,20,21] also suggest using the pattern discovering technique for locating AntiPatterns [14] and code '*smells*'. To provide automation support to user-defined design patterns, CASE tools must be able to capture precisely the recurrent structure and behavior of design patterns, which is often referred as the pattern *leitmotifs* [5]. This requires a modeling language that can precisely specify the invariants of pattern leitmotifs. Unfortunately, a precise modeling approach to pattern leitmotifs is still absent. While the very few pattern experts feel satisfactory with the intuitive approach to design patterns, this brought a big obstacle for the common practitioners to fully understand the invariants of the design patterns. It increases the difficulties in learning and discussing patterns and their relationships.

To enable a more vigorous approach to specify, apply and analyze pattern leitmotifs, previous researchers [3-8,12] attempted to suggest some more precise specification languages to pattern leitmotifs. However, their suggestions erroneously impose excessive constraints to the leitmotif specifications. In other words, the languages are inadequate to convey the abstract nature (flexibility) of pattern leitmotifs that we claim it imprecise in describing the high level constraints of pattern leitmotifs.

Our work aims at providing a modeling language that can truly reveal the abstract nature of design patterns yet be precise enough for all practitioners to comprehend and agree on the specified invariants of leitmotifs. We believe that the development of pattern automation should be based on such a model rather than the other way round as the prior approaches do. In order to achieve this, we reviewed the properties of design patterns. It results in a list of properties [17] of pattern leitmotifs which distinguishes leitmotif structures from the conventional object-oriented (OO) models, class templates and frameworks.

A number of modeling languages have been proposed in the past. In particular, A. Lauder et al. [6] proposed the integration of constraint diagram with UML class diagram to allow pattern roles to be played by an uncertain number of ModelElements. It improved the inflexibility found in

Proceedings of the 26th International Conference on Software Engineering (ICSE'04)
0270-5257/04 $20.00 © 2004 **IEEE**

the LayOM[18] and attribute extension technique [19]. A. H. Eden et al. [5] defined a subset of higher order monadic logics called *LePUS*. It provided accounts for set descriptions and relationships among set elements. This approach is found deficient and highly complex for specifying compound patterns and higher order participants. We will leave this discussion in later part of this paper. A.L. Guennec et al. [4] realized LePUS concept in UML by applying the collaboration diagram to specify the collaboration among pattern roles. This approach inherited the shortfall from the LePUS approach and unable to define a certain types of invariants in pattern leitmotifs. However, their work clarified a certain abstract nature of pattern leitmotifs and inspired us about how *meta-level collaborations* [4] can be used to prevent a certain deficiency found in prior approaches. DPML [7] is one of the most recent works in design pattern modeling. This work introduced a concept of dimension stack, which prevented serious increasing of complexity when modeling complex design patterns while the higher-order logic approach in LePUS does. Also, it suggested the separation of definition and implementation aspects into two roles for a model that is more consistent with the pattern-level abstractions. This concept had inspired us in two aspects. Firstly, new kinds of building blocks are missing to encapsulate the structural knowledge of pattern leitmotifs. In particular, we agree their proposition that the responsibility to declare and implement operations should be considered separately in the context of design patterns. Secondly, these building blocks must precisely define the boundary of their realizations in design level models.

In the next section, we present a list of distinguishing properties of pattern leitmotifs that we consolidated in our study. It illustrates the required language support for modeling pattern leitmotifs. We point out how previous approaches fail to convey these properties. In section 3, we propose an extension to UML 1.5 that makes UML sufficient to model the invariants of pattern leitmotifs. Section 4 provides a case study to illustrate how a GoF design pattern is modeled. In Section 5, we conclude our works and propose future research directions.

2. Structural Properties of Pattern Leitmotifs

Pattern leitmotifs are abstract design models in designers' mind. It captures the most essential invariants that generate concrete solutions for specific design problems. As stated by J. O. Coplien [1], '*the structure of patterns are not themselves solutions, but they generate solutions*'. Such flexibility distinguishes it from OO design models, class templates and frameworks. In this section, we present these distinguishing features in terms of leitmotif structures. These are the features that we extracted from the existing design patterns and therefore must be supported in order to allow the specification of pattern leitmotifs at the correct level of abstraction yet remains its precision in stating the constraints that must be enforced in the pattern instances.

2.1 Role Properties

Design patterns define the structural and behavioral properties that must be fulfilled by classes and objects. It is a partial description which shows a view (the invariants) of the participating classes and objects. The concept of role fits well to this property. In order words, a leitmotif model defines a set of essential roles taken by structural entities (e.g. classes and objects) and behavioral entities (e.g. operations and methods), as well as the collaboration among them. This is more general than the object role model [2,3]. This property suggests that pattern leitmotifs can be specified as a UML collaboration of meta-level entities [4]. We hereinafter refer the entities of pattern leitmotifs as roles and the participants of the entities as actors.

2.2 Sets of actors and set relationships among actors

Each role in a leitmotif model may be taken by a non-fixed number of actors while some may not. All actors taking the same role shares the same available features defined by the role. While some roles may be taken by a set of actors, the specification language must, therefore, support the specification of the generalized relationships between these roles.

We do not go into detail as this property has long been recognized in the previous literature [3-7]. However, it should be noted that the number of actors taking different roles may be constrained by a certain ratio. For example, in Visitor pattern [8], the number of classes taking the *Element* role must be equal to the number of operations taking the *visitConcreteElement* role.

2.3 Role Dimensions

A role can be viewed through different dimensions. Each dimension represents a particular categorization of actors. For example, in the Abstract Factory pattern, the *ConcreteProduct* actors can be categorized by their product type or their product family. Being a particular type of product, all *ConcreteProducts* actors of the same product type are instantiated through the same interface. Belonging to a particular product family, all *ConcreteProducts* actors must be instantiated by the same factory object.

[4,5] realize role dimension with higher order variables. For example, the *ConcreteProducts* role, which has two dimensions, is defined as a 2-dimensional class. Higher order role relationships are defined by generalized relationships, namely, regular and total. This approach unnecessarily imposes orders to different dimensions. Such order increases the complexity of defining role relationships of higher order roles.

2.4 Abstract Relationships among Roles

Relationships among roles do not one-to-one map to design model constructs. This is one of the crucial properties distinguishing pattern leitmotifs from generic class templates and frameworks.

Firstly, some role relationships (which represent high level relations among pattern-level entities) may be reified into a set of design level relationships (which represent object-oriented relations among model level entities). For example, in the Abstract Factory pattern [8], each *ConcreteFactory* actor has the instantiation relationships to *Products* actors. This relationship defines its responsibility to instantiate a particular set of *Products* actors. However, whether the instantiation takes place locally as the factory method in Factory Method pattern [8] or delegates to other methods such as the clone method in the Pluggable Factory pattern [9] is opened. Previous approach including DPML does not allow such flexibility. In the contrary, DPML [7] confined that all instances of the same design pattern must share the same structures and that "*participants (including association role) with no dimension can only be linked with a single UML model elements*". This violates the above fact that Pluggable Factory pattern contains Abstract Factory pattern.

Secondly, relationships among patterns may be reified into different types of design level relationships. Figure 1 shows a simplified OMT diagram of Observer pattern given in the GoF catalog [8]. Previous approaches [4,5,6,7] directly follow the definition of OMT diagram that the relationship between the *Observer* role and *ConcreteObserver* role must be reified by inheritance. They prematurely commit that there must exist an inheritance relationship between the actor of the Observer role and the actors of the ConcreteObserver role. In fact, their relationship only restricts that the ConcreteObserver actors must "*implement the updating interface (of the Observer actor)*" [8]. UML provides a similar abstraction by modeling it as a realization of an interface which "*does not imply inheritance of structure (attributes or associations)*" [10]. DPML declares that "*Each proxy (a role in pattern specification) in the design pattern instance model is linked to a UML design element*". This is not the case for some design patterns including Observer pattern.

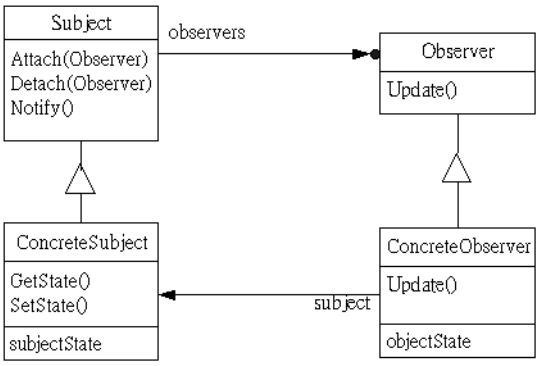

Figure 1. Canonical structure of Observer [8]

Considering the relationship between the Observer role and the ConcreteObserver role, it could be reified into no explicit model level relationships. This happens if the Observer role and the ConcreteObserver role are reified into one single class. It is reasonable because conceptually, a single class may provide the Observer interface as well as the implementation of the interface at the same time. If there is only one class of objects acting as the observer, it is not necessary to have a class only to take the Observer role and another class to take the ConcreteObserver role. Figure 2 shows an OMT diagram illustrating a possible instantiation of the Observer pattern. It is known as an instance of the Observer-Mediator pattern [11]. Figure 3 shows three possible types of reification of the relationship between the Observer role and the ConcreteObserver role.

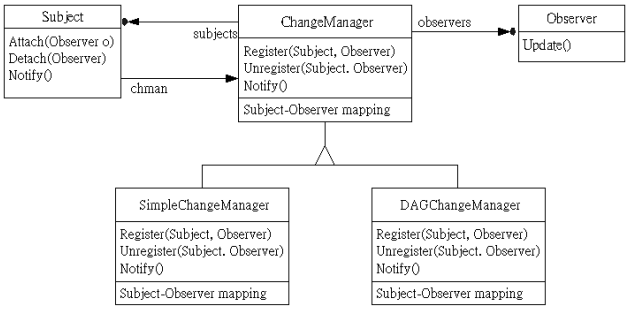

Figure 2. An instance of Observer-Mediator[8]

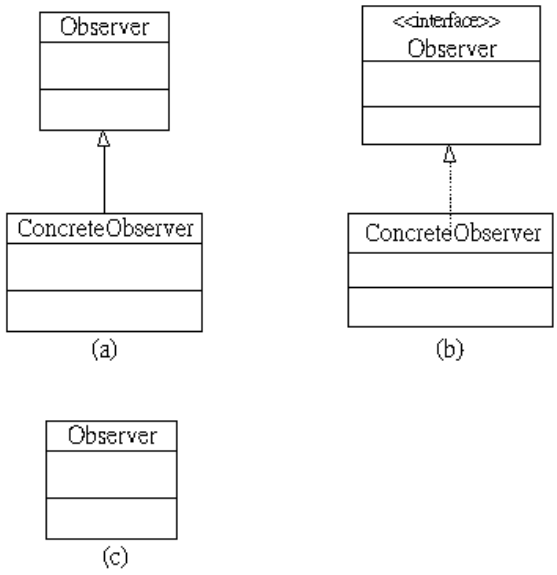

Figure 3. Reification Alternatives of Role Realization

3. Modeling Leitmotif Structure with Meta-level Collaborations and Stereotypes

We take the meta-modeling approach suggested in [4,7] to model the structures of pattern leitmotifs. In particular, we consider pattern leitmotifs as a collaboration of ModelElements [10]. A. L. Guennec at el. [4] took the same view in his approach. However, we have a major different from their works: we do not base our modeling technique on the higher order logics. As a result, our approach reduces the complexity of higher order logics considerably for complex design patterns. In this section, we propose our modeling approach with extensions to UML 1.5 which allows a precise specification to the generic structure of pattern leitmotifs.

3.1 Modeling Pattern Leitmotifs as Meta-level Collaborations

According to the properties discussed in Section 2.1, we consider the structure of pattern leitmotifs as UML collaboration among the elements in UML Metamodel. Roles in pattern leitmotifs can therefore be considered as a UML ClassifierRole with meta-level elements as its base class. However, it is illegal to assign meta-level elements to be the base of UML ClassifierRoles. This problem has been solved by A. L. Guennec et al. [4] using the standard <<meta>> stereotype. More precisely, the collaboration of meta-level elements, which specify the leitmotif structure, becomes accessible in model level by transposing it down to the model level with the <<meta>> stereotypes. We reuse this technique in our approach. Therefore, a pattern leitmotif is specified by a <<meta>> stereotyped UML collaboration diagram. Figure 4 gives the abstract syntax of the UML collaboration. Table 1 gives the mapping of terms from the pattern domain to the UML domain.

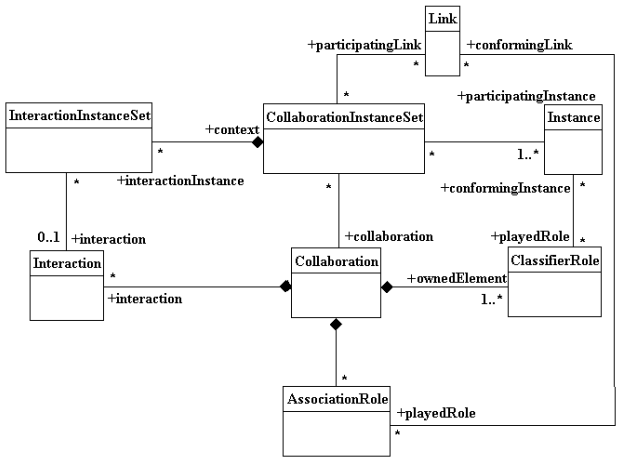

Figure 4. Abstract Syntax of Collaborations in UML 1.5 Specification [10]

Table 1. Synonyms between Pattern Domain and UML Meta-model Domain

Pattern domain	UML domain (all are meta-stereotyped)
Pattern specification	Collaboration
Pattern occurrence	CollaborationInstanceSet
Role	ClassifierRole
Actor	Instance
Role relationships	AssociationRole
Actor relationships	Link

Defining pattern leitmotifs with meta-level collaborations, the relationship between the roles and the actors are no longer Bind dependency but the playedRole-conformingInstance associations. Since all UML concepts are defined as Classes in the abstract syntax of UML Metamodel, a meta-stereotyped Instance can be any non-abstract meta-level elements. However, we expect only ModelElements to be used for modeling pattern leitmotifs. Constraints are therefore added to confine the use of meta-stereotyped Instance. These constraints are specified in the following subsection.

3.1.1 Collaborations.

[1] In a <<meta>> collaboration, the base classifier must be stereotyped with <<meta>> and its name must be that of a subtype of ModelElement.

Context Collaboration **inv**:

self.stereotype → exists(s | s.name = 'meta') **implies** self.ownedElement
→ select (cr | cr.oclIsKindOf(ClassifierRole))
→ forall (cr : ClassifierRole |
(ModelElement.allSubtypes()
→ exists(c | c.name = cr.base.name))
and
(cr.stereotype
→ exists(c | c.name = 'meta')))

[2] There is no Interaction in a <<meta>> collaboration. In fact, <<meta>> Interactions opens a good opportunity for us to define the behavior of a leitmotif model. However, this is outside the scope of this work. Therefore, we impose a temporary constraint to prohibit the use of <<meta>> interaction to avoid undefined semantics.

Context Collaboration **inv**:
self.stereotype → exists(s | s.name = 'meta') **implies** self.interaction->isEmpty()

3.1.2 CollaborationInstanceSet

[1] All *CollaborationInstanceSet* of <<meta>> Collaboration must also be sterotyped with <<meta>>.

Context CollaborationInstanceSet **inv**:
self.collaboration.stereotype
→ exists(s | s.name = 'meta') **implies**
self.stereotype → exists(s | s.name = 'meta')

[2] There is no InteractionInstanceSet in a <<meta>> CollaborationInstanceSet. The reason is the same as the case of <<meta>> Collaboration.

Context CollaborationInstanceSet **inv**:
self.stereotype → exists(s | s.name = 'meta') **implies** self.interactionInstance → isEmpty()

3.2 Modeling Role Relationships

AssociationRole in meta-level collaboration defines the abstract relationships required among actors (ModelElements) of leitmotif roles. To do so, different types of role relationships are defined as stereotypes. All these stereotypes are extended from the *<<implicit>>* association stereotype. *<<implicit>>* stereotype is a standard element in UML 1.5 for defining conceptual relationships in UML models. This agrees to the abstract properties of role relationships mentioned in Section 2.4. Figure 5 shows the stereotypes that we propose as the basic relationship among pattern roles in terms of UML stereotypes. Due to the space limitations, we only provide descriptions to three note-worthy relationships in Table 2 for the comprehension of the following discussion.

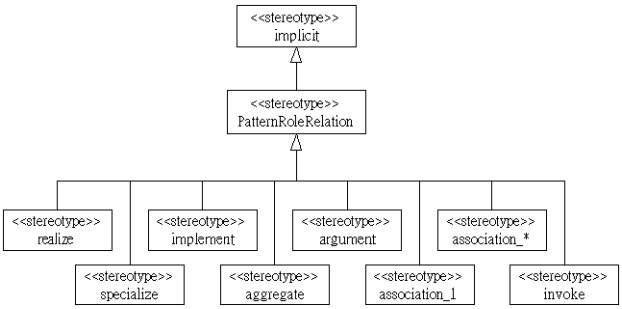

Figure 5. The stereotypes hierarchy for modeling role relationships

Table 2. The Definitions of Role Relationship Stereotypes

Stereotype	Description
Base element: Association Extended from: <<Implicit>>	
<<Realize>>	This stereotype defines that the source role (Class) provides implementation to the behavioral feature of the target role (Class/Interface). It can be reified into the Realization, Generalization or no explicit relationship in UML Model.
<<Implement>>	This stereotype defines that the source role (Method) implements the specification given by the target role (Operation). It can be reified into the polymorphism or the implementation of operations.
<<Invoke>>	This stereotype defines that the triggering of the source role (Operation/Method) will lead to the invocation of the target role (Operation/Method).

Note that all stereotypes can be reified into different model-level relationships. This is due to the abstract nature of role relationships mentioned in Section 2.4. Note also that while the stereotypes we defined can be reified into different model-level relationships, all possible reifications (i.e. the mapping from the stereotyped association to ModelElements that can reify the association) can be defined formally with OCL [23] and thus allow automatically checking whether a given UML model is an instance of a given pattern. We do not describe the mechanism of such automation due to the

scope and the space limitation of this paper. Interested reader may refer to [22].

Considering the <<Realize>> stereotype, it can be reified into Realization, Generalization or no explicit relationship in UML design model. The reason behind this is that this conceptual relationship can be achieved in three different structures.

Firstly, if the source role is taken by a class and the target role is taken by an interface, a standard UML Realization [10] will be used to define the realization of an interface by the class. However, a class (taking the target role) can also act as an interface to define behavioralFeature for subclasses (taking the source role) to implement. In this case, although the model level relationship is Generalization instead of Realization, the realization concept is achieved. In the extreme case, one class takes both roles if it defines and implements the behavioralFeature at the same time. In this case, the realization concept is achieved implicitly. This subsumes the idea of DPML [7] that separating the concern of behavioral definition and implementation into different roles results in a leitmotif model that is closer to the mind of pattern writers.

The <<invoke>> stereotype defines a high-level behavioral relationship among two behavioralFeatures. The reason to introduce behavioral description in the structural model is to specify the constraints which control the possible behavior among objects of different classes. This is a very important type of variants in pattern leitmotifs.

4. A Case Study

In this section, we illustrate how to model pattern leitmotifs with the meta-level collaborations we propose. The first case study provides the complete model of Visitor pattern to show all basic modeling techniques we propose and how the structural properties of leitmotifs are supported with our proposition. The second case study in Abstract Factory illustrates the importance of supporting abstract relationships in a pattern specification language.

4.1 Modeling Visitor Pattern

Figure 6 shows the meta-level collaboration of the Visitor pattern. Each pattern role is defined as a UML ClassifierRole. Some of the ClassifierRoles are taken by classifiers such as class (e.g. *ConcreteElement*), interface or both of them (e.g. *Element*) while the others are taken by behavioralFeatures such as operation (e.g. *Accept*), method (e.g. *VisitConcreteElement*) or both of them. In the following discussion, we refer to the set of instances which conforms a certain ClassifierRole (in a single collaboration instance) as the instance set of the ClassifierRole. In addition, each instance in the ClassifierRole's instance set is referred as the instances of that ClassifierRole.

ClassifierRoles are related to AssociationRoles which define the abstract relationships required among Instances. The composite relationship between the *Element* role and *Accept* role defines the required behavioralFeatures, *Accept*, of *Element* instances. Other AssociationRole can also be seen among the ClassifierRoles. The detail semantics (and constraints of the association stereotypes) can refer to section 3.2.

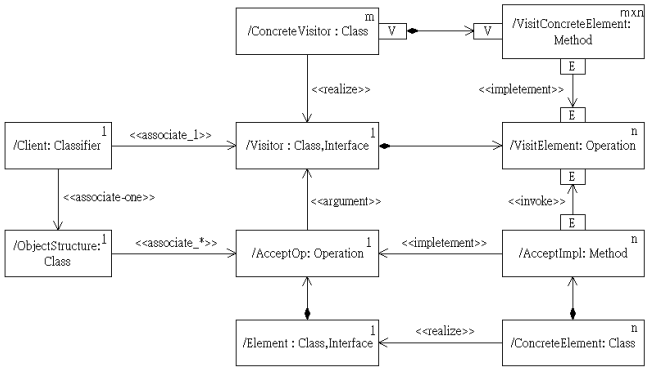

Figure 6. Meta-level Collaboration of Visitor Leitmotif

The number at the right upper corner of each ClassifierRole denotes the number of its instances in one collaboration instance (pattern instance). This is the standard UML notation. However, a difference is that for some ClassifierRole, the value is given in a mathematical expression with variables of integer type. While there are no restriction to the type of mathematical expression that can be used to specify the object multiplicity, the mathematical expression must give a positive integer in any circumstances in order to maintain the *well-formedness* of the collaboration diagram.

In this example, there are *m ConcreteVisitor* instances in each collaboration instance. We use this to maintain the instance ratio among the ClassifierRole. Therefore, in any collaboration instance,

$$\frac{\#of\ \ ConcreteVisitor\ instance}{\#of\ \ VisitConcreteElement\ instance} = m:mn$$

The filtering property of UML *quantifier* let us partition a large set of instances with different categorizations. This property let us specify what categorization of an instance set is related to another

instance set. Considering the *VisitConcreteElement*, there are *mn VisitConcreteElement* instances for each collaboration instance. Each has two dimensions, namely, the visitor dimension (*V*) and the element dimension (*E*). Figure 7 illustrates how the *VisitConcreteElement* instances are partitioned in the *V* and *E* dimension.

The *V*-dimension partitions the *mn VisitConcreteElement* instances into *m* sets. Each set is related to its corresponding *ConcreteVisitor* instance holding the same quantifier value. As a result, each *ConcreteVisitor* instance owns *n VisitConcreteElement* instances. Note that this number agrees to the ratio of their instances obtained before. On the contrary, the *E*-dimension partitions the *VisitConcreteElement* instance set into *n* subsets. Similarly, each subset is related to its corresponding *Element* holding the same quantifier value. The value of the quantifiers can be obtained by assessing the attribute values of the meta-level elements.

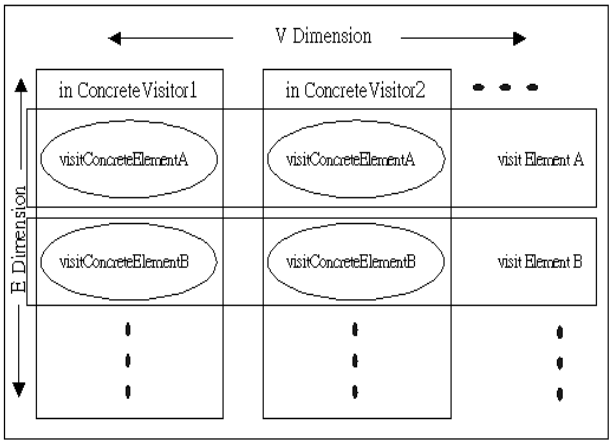

Figure 7. The partitioning of ConcreteVisitor actors in V and E dimensions

When the leitmotif model is instantiated, the abstract relationship can be reified into different model level relationships for different situations. Also, each instance may take any number of ClassifierRoles as long as it conforms to the required properties. As a result, a variety of concrete solutions can be obtained by varying these two factors. Taking the GoF Visitor as an example, the *Element* and *ConcreteElement*, as specified in the prior approach, must be instantiated into a class hierarchy. However, it is no longer mandatory in our approach. Instead, the *Element* role could be taken by an interface with a set of *ConcreteElement* class implementing it.

4.2 Modeling Abstract Factory Pattern

The use of the Abstract Factory pattern in the Pluggable Factory pattern illustrates the significance of conveying the true abstract nature of pattern leitmotifs. The Pluggable Factory pattern, shown in Figure 8, is a compound pattern composed by the Abstract Factory pattern and the Prototype pattern. However, the class hierarchy of the Factory role in the Abstract Factory pattern is eliminated in the Pluggable Factory pattern. Previous literature [9] explains that the *Abstract Factory* class and the *Abstract Factory* class hierarchy are removed because the hierarchy is no longer useful. However, the Abstract Factory role is a participant of the Abstract Factory pattern. It is a part of the invariants which should never be removed from any composition of this design pattern. Otherwise, the composition violates the invariants of the Abstract Factory pattern. This implies that the composition does not contain the Abstract Factory pattern indeed. In addition, according to the specification of Pluggable Factory, the *Client* "uses … the interface declared by *ConcreteFactory* …". Declaration of the creation interface in the Abstract Factory pattern is the responsibility of the Abstract Factory role according to the GoF Catalog. Therefore, it is clearly that the ConcreteFactory role in the Pluggable Factory pattern is actually not the ConcreteFactory role in the Abstract Factory pattern. Instead, it is a compound of the AbstractFactory role and the ConcreteFactory role of the Abstract Factory pattern. This can explain why the ConcreteFactory role in the Pluggable Factory pattern takes also the responsibility of the AbstractFactory role in the Abstract Factory pattern. It also supports the community's belief that Pluggable Factory pattern uses the Abstract Factory pattern.

Taking the previous approaches [4-7], the AbstractFactory role and ConcreteFactory role must be instantiated into a class hierarchy. It is apparently conflicting to the case of the Pluggable Factory pattern. On the contrary, in our approach, the AbstractFactory role and the ConcreteFactory role can be defined as two ClassifierRoles with a <<realize>> association. Therefore, the Abstract Factory pattern in the Pluggable Factory pattern is only a particular instance of the Abstract Factory pattern where the Abstract Factory role and the ConcreteFactory role are both taken by a single class.

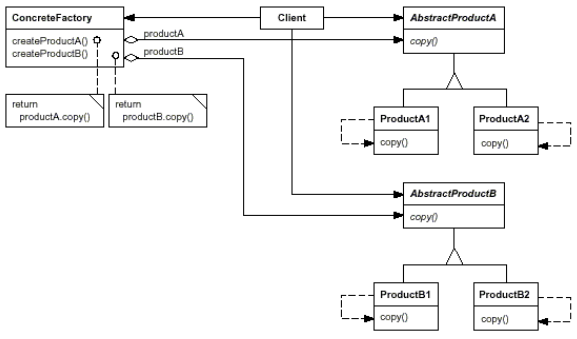

Figure 8. An instance of Pluggable Factory [9]

5. Modeling Pattern Occurrences

ModelElements participate in a meta-level collaboration through a *conformingInstance-playedRole* association between the *Instance* and *ClassifierRole* as shown in Figure 4. However, there is no explicit semantics in UML v1.5 to specifying how the roles in pattern leitmotifs are participated by ModelElements. The participation was only modeled as parameter binding dependency in UML 1.5 [10]. Since pattern leitmotifs cannot be modeled by parameterized collaborations but meta-collaborations that we are proposing, we need to define a new meta-level element to capture the semantic of role participations.

To minimize the work for migrating our proposition to UML 2.0, we propose to replace the original *conformingInstance-playedRole* association with a *RoleParticipation* association class. Figure 9 shows the extended meta-model after the replacement. Note that there is no modification required for the original specification of ClassifierRole and Instance.

5.1 Role Participation

A RoleParticipation specifies the participation of an Instance in a ClassifierRole. It is a kind of Dependency in which the playedRole (ClassifierRole) takes the server role to define the Features conformed by the conformingInstance (Instance) taking the client role.

Associations

playedRole: The classifierRole being played in the participation.

conformingInstance: The instance playing the classifierRole in the participation.

Well-formedness Rules

[1] The type of the conformingInstance must be the base of the playedRole or any subclass of it.

context RoleParticipation **inv**:
self.playedRole.base.allSubtypes()
⟶ exists (c | c.name =
self.conformingInstance.type.name)

Notation

As shown in Figure 9, RoleParticipation is a kind of Dependency. Therefore, the official notation for pattern occurrences can straightly be reused as a representation of pattern occurrences in UML models.

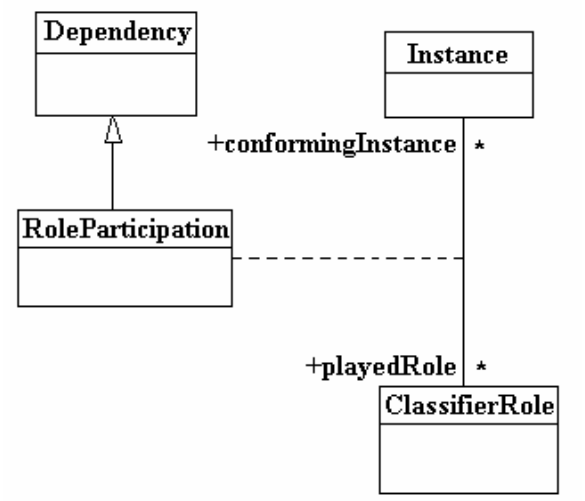

Figure 9. Extensions to UML Collaborations for RoleParticipation

6. Conclusion and Future works

In this paper, we suggest a list of essential properties of pattern leitmotifs. This set of properties distinguishes leitmotif structures from conventional OO models, class templates and frameworks. Based on this set of structural properties, we examine the prior pattern specification languages [3-7] and pinpointed their deficiency based on these essentials. In general, the previous specification languages [4-7] fall short in revealing the abstract nature of pattern leitmotifs. In other words, they can only be used to model some instances of a pattern leitmotif. We believe that obtaining precise models of the pattern leitmotifs is an important step towards CASE tool supports of user-defined design patterns. Capturing the true invariant of design patterns, engines and different packages of common mappings from the meta-level collaborations to model level structures can be suggested to facilitate the application, verification and recovery of design patterns. In that case, the automation support would be more complete and usable than the ad hoc and contemporary ones.

To provide a more complete and precise UML-based modeling of pattern leitmotifs, we suggested an extension to UML 1.5. In brief, we make use of the meta-modeling techniques as [4,7] by using collaboration diagram to specify the collaboration among ModelElements. To do so, we specify an additional set of well-formedness rules to the metamodel of UML Collaboration. One AssociationClass named RoleParticipation is introduced to provide semantics for the participation of roles. Since RoleParticipation is a subclass of Dependency, the participation of ModelElement in a leitmotif can be represented no difference from the original UML

specifications. In addition, a set of stereotypes is defined to encapsulate the high level relationships among roles. We claim that these relationships convey the true abstract nature of role associations that reveals designers' intuition to the invariants of pattern leitmotifs. The abstract property of design pattern can therefore be retained by avoiding premature commitments. Two case studies are given to illustrate how this is achieved in our approach.

We believe that modeling design patterns with UML have several significant advantages. Firstly, UML is a de facto standard of software modeling which we can reasonably foresee that it will become one of the mainstream languages in software industry. Towards a first class CASE tool support of user-defined design patterns, integrating design patterns with UML semantics ensures a more comprehensible and usable pattern specifications to the pattern community. Secondly, while the concept of components and frameworks has a significant role in the UML 2.0, our work opens the opportunity for a better collaboration between design patterns and other software reuse technology such as the components and the frameworks under a single modeling platform.

We are currently working on the specification of leitmotif behavior with the aid of Action Semantics. In terms of CASE tools support, we are testing a few mechanisms that allow generation of constraints for pattern verification as well as matching rules for pattern recovery given a UML design model. Study has also started to explore how the current proposal can be adapted to the UML 2.0 as a Profile for the modeling of Design Patterns and reusable high-level design concepts.

7. Acknowledgments

This work is supported by the Hong Kong Polytechnic University under grant no. G-W106.

8. Reference

[1] J. O. Coplien, *Software Patterns,* SIGS Management Briefings, SIGS Books, New York, 1996.

[2] T. Reenskaug, P. Wold and Odd A. Lehne, *Working With Objects*. Manning, Greenwich, 1995.

[3] D. Riehle, "Describing and Composing Patterns Using Role Diagram", In *Proc. of the 1st International Conference on Object-Orientation*. St. Peterburg Electrotechnical University. Russia. 1996. pp.137-152.

[4] G. Sunyé, A.L. Guennec, J-M Jézéquel. "Precise Modeling of Design Patterns", In *Proceedings of UML 2000*, volume 1939 of *LNCS*, pages 482--496. Springer Verlag, 2000.

[5] A. H. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, University of Tel Aviv, 1999.

[6] A. Lauder and S. Kent. "Precise visual specification of design patterns", *ECOOP'98 Proceedings*, Lecture Notes in Computer Science, Springer, 1998, vol. 1445, pp. 114-134.

[7] Mapelsden, D., Hosking, J. and Grundy, J. "Design Pattern Modelling and Instantiation using DPML". In *Proceeding of TOOLS Pacific 2002*, Sydney, Australia. Conferences in Research and Practice in Information Technology, 10. Noble, J. and Potter, J., Eds., ACS

[8] E.Gamma, Richard Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software. Professional Computing Series. Addison-Wesley, Reading*, MA, 1995.

[9] J. Vlissides, "Pluggable Factory Part II", *C++ Report*, Feb, 1999.

[10] OMG. *UML 1.5 Specification.* Formal/2003-03-01.

[11] D. Riehle. "Composite Design Patterns". In *OOPSLA '97 Conference Proceedings*, ACM SIGPLAN Notes, vol.32, no.10, pp.218-228, October 1997. ACM Press.

[12] A. P. Flores and R. Moore, "GoF Structural Patterns: A Formal Specification." *UNU/IIST Report No.207*, August, 2000.

[13] A. L. Correa, C. M. L. Werner, G. Zaverucha. *Object Oriented Design Expertise Reuse: An Approach Based on Heuristics, Design Patterns and Anti-patterns*. Software Reuse: Advances in Software Reusability. 6th International Conference, Vienna, Austria, June 2000. Proceedings, Springer.

[14] W. J. Brown, R. C. Malveau, H. W. McCormick III, T. J. Mowbray. *AntiPatterns: Refactoring Software,* Architectures, and Projects in Crisis. Wiley Computer Publishing, 1998.

[15] Jeffrey K. H. Mak, Clifford S. T. Choy and Daniel P. K. Lun, "Hierarchical Relationships among Good Design Patterns and Bad Design Patterns", In *Proc. of International Conference on Computer Science and Technology (CST'2003)*. ACTA Press, pp.7-13.

[16] G. Florijn, M.Meijers, and P. v. Winsen. "Tool support for object-oriented patterns", *ECOOP'97 Proceedings*, Lecture Notes in Computer Science, Springer, June 1997, vol. 1241, pp. 472-495.

[17] Doug Lea. "Christopher Alexander:an Introduction for Object-Oriented Designers". *The Patterns Handbook: Techniques, Strategies, and Applications*, Cambridge University Press, 1998, pp.407-422.

[18] J. Bosch. "Design Patterns as Language Constructs", *Journal of Object-oriented Programming*, vol.11, no. 12, 1998. pp.18-23.

[19] G. Hedin. "Language Support for Design Patterns using Attribute Extension", Workshop on Language

Suppport for Design Patterns and Object Oriented Frameworks (LSDF). In *Proceeding ECOOP' 97*, Springer Verlag, pp.209-231.

[20] O. Ciupke. "Automatic detection of design problems in object-oriented reengineering", In *Proceeding of TOOLS*. 30:18-32, 1999.

[21] Y-G Guéhéneuc, H. A-Amiot. "Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-class Design Defects", *Technology of Object-Oriented Languages*, 2001. TOOLS 39. 39th International Conference and Exhibition on, 2001.

[22] K. H. Mak, "Precise Specification of Design Patterns and Compound Patterns", *2nd Draft of MPhil. Thesis*, The Polytechnic University of Hong Kong, 2004.

[23] J. B. Warmer, *The Object Constraint Language : Precise Modeling with UML*, Addison Wesley Longman, 1999.