Precise Modeling of Design Patterns

Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel

IRISA/CNRS, Campus de Beaulieu, F-35042 Rennes Cedex, FRANCE
email: aleguenn,sunye,jezequel@irisa.fr

Abstract. Design Patterns are now widely accepted as a useful concept for guiding and documenting the design of object-oriented software systems. Still the UML is ill-equipped for precisely representing design patterns. It is true that some graphical annotations related to parameterized collaborations can be drawn on a UML model, but even the most classical GoF patterns, such as Observer, Composite or Visitor cannot be modeled precisely this way. We thus propose a minimal set of modifications to the UML 1.3 meta-model to make it possible to model design patterns and represent their occurrences in UML, opening the way for some automatic processing of pattern applications within CASE tools. We illustrate our proposal by showing how the Visitor and Observer patterns can be precisely modeled and combined together using our UMLAUT tool. We conclude on the generality of our approach, as well as its perspectives in the context of the definition of UML 2.0.

1 Introduction

From the designer point of view, a modeling construct allowing design pattern [8] participant classes to be explicitly pointed out in a UML class diagram can be very useful. Besides the direct advantage of a better documentation and the subsequent better understandability of a model, pointing out an occurrence of a design pattern allows designers to abstract known design details (e.g. associations, methods) and concentrate on more important tasks.

We can also foresee tool support for design patterns in UML as a help to designers in overcoming some adversities [2][7][15]. More precisely, a tool can ensure that pattern constraints are respected, relieve the designer of some implementation burdens, and even recognize pattern occurrences within source code, preventing them from getting lost after they are implemented. In this context, we are not attempting to detect the need of a design pattern application but to help designers to explicitly manifest this need and therefore abstract intricate details. Neither are we trying to discover which implementation variant is the most adequate to a particular situation, but we would like to discharge programmers from the implementation of recurrent trivial operations introduced by design patterns. According to James Coplien [3] p. 30 - *patterns should not, can not and will not replace programmers* - , our goal is not to replace programmers nor designers but to support them.

But in its current incarnation as of version 1.3 from the OMG, the UML is ill-equipped for precisely representing design patterns. It is true that some graphical

annotations related to parameterized collaborations can be drawn on a UML model, but even the most classical GoF patterns, such as Observer, Composite or Visitor cannot be modeled precisely this way (see Sect. 1.1). Ideas to overcome the shortcomings of collaborations are sketched in Sect. 1.2, providing some guidelines to model the "essence" of design patterns more accurately. An example showing how the Visitor and Observer patterns can be precisely modeled and combined together using our UMLAUT tool is presented in Sect. 2. Related approaches are then discussed in Sect. 4. We then conclude with a discussion of the generality of our approach, as well as its perspectives in the context of the definition of UML 2.0. To alleviate the reading of the paper, we have moved to an appendix some complementary support material needed to understand the extensions to UML that we propose.

1.1 Problem Outline

The current official proposal for representing design patterns in the Unified Modeling Language is to use the *collaboration* design construct. Indeed, the two conceptual levels provided by collaborations (i.e. parameterized collaboration and collaboration usage) seem to be appropriate to model design patterns.

At the general level, a parameterized collaboration is able to represent the structure of the solution proposed by a pattern, which is enounced in generic terms. Here patterns are represented in terms of classifier and association roles. The application of this solution, i.e. the terminology and structure specification into a particular context (so called instance or occurrence of a pattern) are represented by expansions of template collaborations. This design construct allows designers to explicitly point out participant classes of a pattern occurrence.

Parameterized collaborations are rendered in UML in a way similar to template classes [1], p.384. Thus, roles represented in theses collaborations are actually template parameters to other classifiers. More precisely, each role has an associated *base*, which serves as the actual template parameter (the template parameter and the argument of a binding must be of the same kind [14] p.2-46.)

However, there are severe limitations for modeling design patterns as parameterized collaborations:

First, the use of generic templates is not fully adapted to represent the associations between pattern roles and participant classes. More precisely, as each classifier role (actually its base class) is used as a template parameter, it can be bound to at most one participant class. Therefore, design patterns having a variable number of participant classes (e.g. Visitor, Composite) cannot be precisely bound. Also, if the use of base classes in a template collaboration is necessary to allow the binding (bindings can only be done between elements having the same meta-type), its utility and its underlying representation are unclear.

Second, some constraints inherent to design patterns cannot be represented by collaborations, since they involve concepts that cannot be directly included as OCL constraints. For instance, in the Visitor [8] pattern, the number of visit methods defined by the visitor class must be equal to the number of concrete

element classes. This constraint can not be written in OCL unless an access to the UML meta-model is provided.

Third, collaborations provide no support for feature roles. In design patterns, an operation (or an attribute) is not necessarily a real operation. It defines a behavior that must be accomplished by one or more actual operations. This kind of role cannot be defined in a collaboration, nor is it possible to describe behavioral constraints (e.g. operation A should call operation B).

These limitations were extensively discussed in previous work by the authors [16]. In this paper, we propose some solutions to overcome these problems.

A misunderstanding with the term *role* might be a possible source of the present inadequacy of collaborations to model design patterns. In a UML collaboration, roles represent placeholders for *objects* of the running system.

However, in the design pattern literature, the term role is often associated to participant *classes* and not only objects in a design model. There can also be roles for associations and inheritance relationships. In other words, pattern roles refer to an upper level. This subtle difference can be noted when binding a parameterized collaboration to represent an occurrence of a pattern: it is impossible to assign a single role to more than one class.

This difference is also observable when writing OCL constraints to better model a design pattern: frequently, this kind of constraints needs access to meta-level concepts, that cannot be directly accessed by OCL.

1.2 Patterns as sets of constraints: Leitmotiv in UML

Design patterns are described using a common template, which is organized in a set of sections, each one relating a particular aspect of the pattern. Before extending this description of how to model design patterns in UML, let us dispel some possible misunderstanding concerning the modeling of design patterns.

It is not our intention to model every aspect of design patterns, since some aspects are rather informal an cannot be modeled. We are interested in a particular facet of patterns, which is called *Leitmotiv* by Amnon Eden [5]: the generic solution indicated by a design pattern, which involves a set of participants and their collaborations.

Our intention is to model the leitmotiv of design patterns using structural and behavioral constraints. The goal of this approach is to provide a precise description of how pattern participants should collaborate, instead of specifying a common fixed solution. Design patterns can be expressed as constraints among various entities, such as classifiers, structural and behavioral features, instances of the respective classifiers, generalization relationships between classifiers, generalization relationships between behavioral features, etc.

All those entities are modeling constructs of the UML notation. That is, they can be thought of as instances of meta-classes from the UML meta-model. This suggests that patterns can be expressed with meta-level constraints.

The parameters of the constraints together form the *context* of the pattern, i.e. the set of participants collaborating in the pattern. Since the UML

meta-model is defined using a UML class diagram, we can make the reasonable assumption that it is not different than any other UML model.

Therefore, we propose to use meta-level collaborations to specify design patterns. However, to avoid any ambiguity in the sequel, we will explicitly use "M2" if necessary when referring to the UML meta-model and "M1" when referring to an ordinary UML model, following the conventions of the classical 4-layer metamodel architecture. The material presented in appendix explains in details how collaborations and OCL constraints can be used together in a complementary way, and we apply this principle to specify the structural constraints of patterns in Sect. 2. A different approach is needed to specify the behavioral properties associated with a pattern, and Sect. 2.4 presents how temporal logic could be used to that purpose. Finally, Sect. 3 shows how an appropriate redefinition of the mapping of dashed-ellipses permits to keep that familiar notation to represent occurrences of design patterns.

2 Modeling Design Patterns in Action

2.1 Presentation of the Visitor and Observer Patterns

Figure 1 shows the participants in the Visitor design pattern represented as a meta-level collaboration. It consists of a hierarchy of class representing concrete nodes of the structure to be visited, and a visitor class (or possibly a hierarchy thereof). Each element class should have an accept() routine with the right signature, for which there should be a corresponding visitElement() routine in the visitor class.

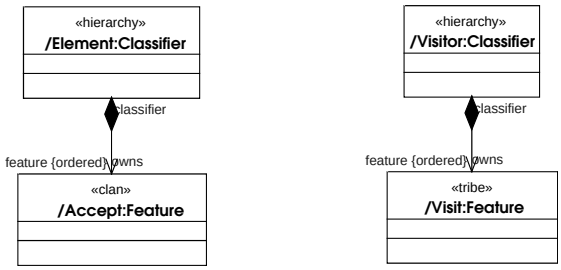


Fig. 1. Meta-level collaboration of the Visitor design pattern

Figure 2 shows a collaboration representing the participants in the Observer design pattern. It consists of a subject class (or hierarchy thereof) whose instances represent observed nodes, and a class (or hiearchy thereof) whose instances represent observers. The subject class should offer routines to attach or detach an observer object, and a routine to notify observer objects whenever the state of the subject changes. The observer class should offer an update() routine for notification purpose.

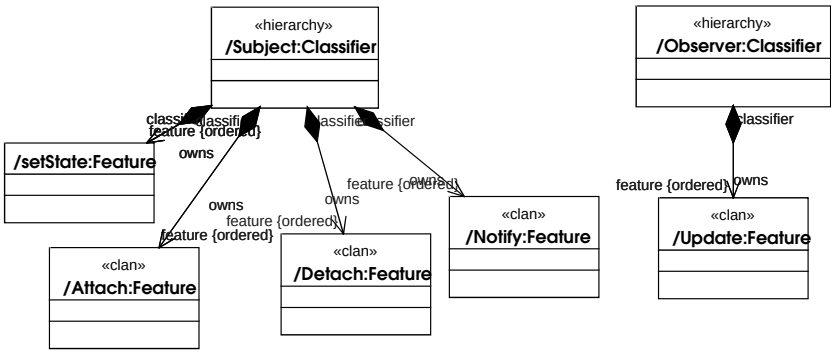


Fig. 2. Meta-level collaboration of the Observer design pattern

2.2 Structural constraints

When a behavioral (or a structural) feature appears in the specification of a design pattern, this does not mean that there must be an identical feature in a pattern occurrence. Actually, the specification describes *roles* that can be played by one or more features in the model. Using meta-level collaborations clears this possible confusion.

An example of a *feature role* is the Attach() feature of the Observer pattern. It represents a simple behavioral feature that adds an observer into a list. This does not mean that this feature cannot perform other actions, nor that it should be named "Attach" and have exactly the same parameters.

Some feature roles are more complex than the above example is, since they represent the use of dynamic dispatch by a family of behavior features. An example of this is the Accept() feature role of the Visitor design pattern. It represents a feature that should be implemented by concrete elements. Such a family of features is named a *Clan* by Eden [5] (p.60).

Finally, some other feature roles represent a set of clans, i.e. a set of features that should be redefined in a class hierarchy. The feature role Visit() of the Visitor design pattern is an example of this particular role. It designates a set of features (one for each concrete element) that should be implemented by concrete visitors. Such a family of features is named a *Tribe* by Eden [5] (p.61).

2.3 Factoring recurring constraints with stereotypes

These kinds of structural constraints among features and classes are recurring in pattern specifications, and factoring them would significantly ease the pattern designer's task.

The natural means provided by the UML to group a set of constraints for later reuse is the *stereotype* construct. Figure 3 recalls how constraints can be attached to a stereotype. These constraints will later transitively apply to any elements to which the stereotype is applied (OCL rule number 3, page 2-71 of the UML1.3 documentation [14]).

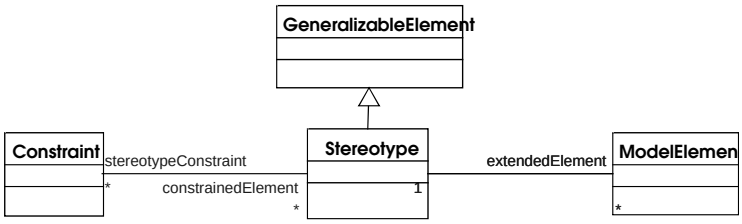


Fig. 3. How to attach recurring constraints to a stereotype

Clans The first stereotype presented here is called <<Clan>>. A clan is a set of behavioral features that share a same signature and are defined by different classes of a same hierarchy. In OCL, clans are defined as follows:

```
isClan(head     : BehaviorFeature,
       features : Sequence(BehaviorFeature)) inv:
features->forAll(f | f.sameSignature(head))
```

Other examples of clans are the AlgorithmInterface() feature role of the Strategy pattern or the notify() feature role of the Observer pattern (see Fig.2).

Tribes The second stereotype is called <<Tribe>> and is somewhat similar to the first one. A tribe is set of behavioral features that consists of other sets of behavioral features each of which is a clan. A tribe is defined in OCL as follows:

```
isTribe(heads, features:Sequence(BehaviorFeature)) inv:
features->forAll(f|heads->exists(head| head.sameSignature(f)))
```

Elements of a tribe do not necessarily have the same signature, as elements of a clan do. Other examples of tribes are the setState() feature role of the Observer pattern (see Fig. 2) or the Handle() feature role of the State pattern.

Auxiliary operations The above OCL constraints both use an operation that compares behavioral feature signatures. Two features share the same signature if they have the same set of parameters (including the "return" parameter):

```
sameSignature(featA, featB : BehaviorFeature) : boolean;

sameSignature =
        featA.parameter()->collect(par | par.type()) =
        featB.parameter()->collect(par | par.type())
```

2.4 Behavioral properties and temporal logic

Behavioral properties of the Visitor pattern

[1] A given call of anElement.accept(aVisitor) is always followed by a call of aVisitor.visitAnElement(anElement). If we want the call to visit to be synchronous ("nested flow of control"), we need a second constraint:
[2] A given call of aVisitor.visitAnElement(anElement) always preceeds the return of a call of anElement.accept(aVisitor).

Behavioral properties of the Observer pattern

[1] After a given call of aSubject.attach(anObserver), and before any subsequent call of aSubject.notify() or of aSubject.detach(anObserver), the set of observers known by aSubject must contain anObserver.
[2] A given call of aSubject.detach(anObserver) if any must follow a corresponding call of aSubject.attach() and no other call of aSubject.attach(anObserver) should appear in between.
[3] After a given call of aSubject.detach(anObserver), and before any subsequent call of aSubject.notify() or of aSubject.attach(anObserver), the set of observers known by aSubject must not contain anObserver.
[4] A given call of aSubject.notify() must be followed by calls of observers.update(), and all these calls must precede any subsequent call of aSubject.attach(anObserver), of aSubject.detach(anObserver) or of aSubject.notify(). Note that we do not require the notification to be synchronous, just that all the observers which are known when notification starts will be eventually notified. We could allow for collapsing of pending notification events by allowing another call of aSubject.notify() before all calls of update: pending calls would then not have to occur twice to satisfy the constraints (this can be very useful in GUI design notably, to improve rendering speed).

Using temporal logic Note how general and declarative the constraints are. For instance, it is not written down that update() shall contain a loop calling notify(), because the pattern *does not have* to be implemented like that. We do not want to proscribe correct alternative solutions. The constraints just ensure that some events shall occur if some others do, and prevent erroneous orders.

A form of temporal logic would provide the right level of abstraction to express the behavioral properties expected of all pattern occurrences. Some recent research efforts [12, 4] have begun to investigate the integration of temporal operators within OCL, reusing the current part of OCL for atoms of temporal logic formulas. Although this work is very valuable and necessary, we cannot reuse it directly in our context, because the resulting OCL expressions belong to the model level (M1): They lack quantification over modeling elements and therefore cannot capture the "essence" of a pattern's behavior.

Behavioral properties do rely on quantification over operations, their target, their parameters, and various other entities which will later be bound to elements

of the M1 level. This suggests that they are at the same level as the OCL expressions we used to specify the structural constraints of patterns. However, OCL is not really appropriate to formally specify the behavioral properties of a pattern: such OCL expressions would have to express very complex properties on a model of all possible execution traces (such a "runtime model" is actually under work at the OMG.) Making these OCL expressions completely explicit would amount to building a model checker: A simulator would produce execution traces following the rules of a semantics, and the OCL expressions would represent the properties to be checked. However, special OCL operations could be defined to simplify complex OCL expressions. To the extent that the designer could use these predefined operations to completely abstract away from the actual details of the runtime model, they would provide a formal definition for a set of OCL temporal operators.

Another interesting topic for future research is how to adapt UML sequence diagrams so that they could describe behavioral constraints at the more general level needed for design patterns.

3 Representing pattern occurrences

3.1 Bridging the gap between the two levels of modelisation

An occurrence of a pattern can be represented by a collaboration occurrence (see more details in the appendix) at the meta-model level (M2) connecting (meta-level) roles in the collaboration with instances *representing* modeling elements of the M1 model (instances in M2 represent modeling elements of M1). The bindings would belong to the M2 model, not to the M1 model. They link instances in M2 that are *representations* of modeling elements of M1.

The problem is similar to expressing that a class in a given (M1) model is an "instance" of a <<meta>> class of the same model. Normally the "is an instance of" dependency is between an Instance and a Classifier, and not between a Classifier and another Classifier. So this dependency would appear in the model of the model, where the normal class would appear as an instance while the <<meta>> class would appear as a classifier.

The standard <<meta>> stereotype acts as an inter-level "bridge", making up for the fact that UML is not a fully reflexive language and therefore avoiding a very significant extension. Using <<meta>> allows for representing or *transposing* M2 entities into M1. An appropriate M1 dependency can then be used to relate M1 entities to entities "transposed" from M2.

Now we can define a pattern using a M2 collaboration and still represent it and access it from an ordinary M1 model by using the <<meta>> stereotype. A pattern occurrence is then represented by a composite dependency between arbitrary model elements and classifier roles of the collaboration transposed from M2 (see Fig. 4), in a way similar to a CollaborationOccurrence (see Fig. 6 in appendix), except for the fact that a real collaboration occurrence connects instances to classifier roles, while a pattern occurrence connects any model elements to classifier roles of a <<meta>> collaboration.

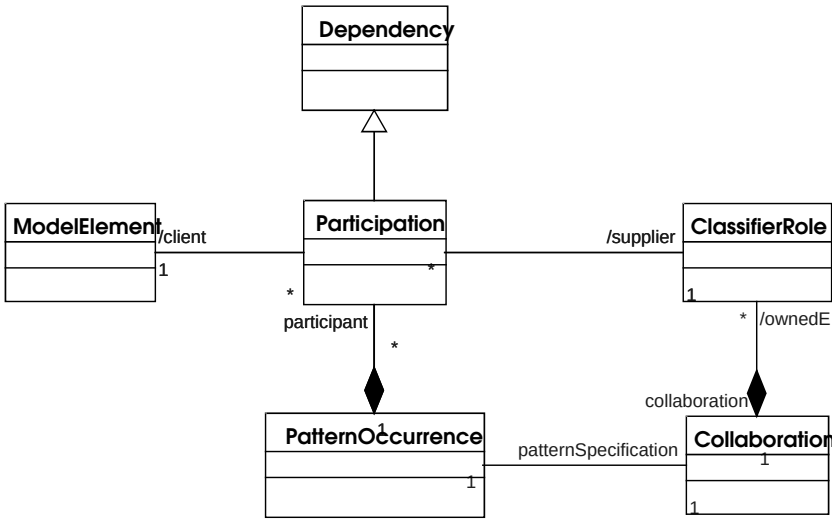


Fig. 4. Meta-model extensions to represent pattern occurrences

Additional well-formedness rules associated to pattern occurrences

[1] The pattern specification of the pattern occurrence must be a <<meta>> collaboration.

```
context PatternOccurrence inv:
        self.patternSpecification.stereotype->
           exists(s | s.name = 'meta')
```

[2] The number of participants must not violate multiplicity constraints of the roles in the <<meta>> collaboration.

```
context PatternOccurrence inv:
   (self.patternSpecification.ownedElement->
      select(cr | cr.oclIsKindOf(ClassifierRole)))
   ->forall(cr : ClassifierRole |
      let nbOfParticipants =
         cr.supplierDependency->select(p |
         p.oclIsKindOf(Participation))->size()
      in (cr.multiplicity.ranges->forall(r |
         r.lower <= nbOfParticipants
                        and nbOfParticipants <= r.upper)))
```

Additional well-formedness rules associated to participations

[1] The supplier must be a classifier role.

```
context Participation inv:
        self.supplier.oclIsKindOf(ClassifierRole)
```

[2] The supplier role of the participation must be a role of the collaboration specifying the corresponding pattern occurrence.

```
context Participation inv:
        self.supplier.namespace
                = self.patternOccurrence.patternSpecification
```

[3] The client element of the participation (the participant) must be of a kind whose name matches the name of the base of the role or any sub-class of the base.

```
context Participation inv:
   supplier.oclAsType(ClassifierRole).base.allSubtypes()->
      exists(c | c.name = self.client.type.name)
```

The last rule is the most significant one in the inter-level bridging context. It ensures that a pattern occurrence could be represented at the M2 level directly by a collaboration occurrence binding roles to *conforming* instances representing modeling elements of M1. Note that the type cast realized with oclAsType is always valid because of rule [1].

3.2 Graphical representation of pattern occurrences

Figure 5 presents a class diagram in which two pattern occurrences are used (we assume that all visit() operations call the markNode() operation, whose effect is in turn notified to the observer that can count marked nodes).

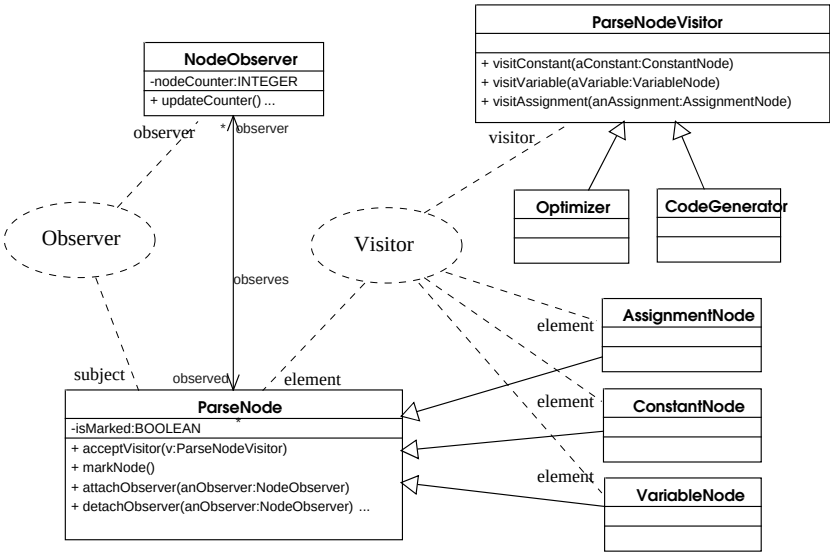


Fig. 5. A model combining two pattern occurrences

We chose to keep the familiar ellipse notation to represent both pattern occurrences as defined in the previous section and collaboration occurrences as defined in the appendix, in order not to disrupt designers accustomed to the current UML notation for design patterns.

Note that Fig. 5 does not show all participation relationships, because this would clutter the diagram for no good reason since there is no ambiguities. For the same reason, neither does it represent the relationships between pattern occurrences and the corresponding <<meta>> collaborations. However, a tool should provide the option of showing all participations, even those involving behavioral features or generalizations, possibly using dialog boxes. Also, as there may be many behavioral features participating in a pattern such as the observer (potentially all those that change the state of the subject), a good tool should also propose a default list of matching participants to ease the designer's task.

4 Conclusion and related work

4.1 Related work

PatternWizard is one of the most extensive projects of design pattern specification, and has influenced our research work in several points. PatternWizard proposes LePUS [6] a declarative, higher order language, designed to represent the generic solution, or *leitmotif*, indicated by design patterns. Our work differs from PatternWizard in two aspects. First, we use UML and OCL to specify patterns. We believe that a UML collaboration and OCL rules can be easier to understand than LePUS formulae and the associated visual notation. Second, PatternWizard works at the code level and is not integrated to any design model.

An approach to the validation of design patterns through their precise representation is proposed by Görel Hedin [9]. She uses attribute grammars to precisely model a pattern and explicit markers in a program to distinguish a pattern occurrence and validate it. Patterns are represented as a set of class, attribute and method roles, related by *rules* which have the same goal as the OCL constraints in our proposal. Using attribute extension is a way to extend the static semantics with new rules, while leaving the original syntax unchanged.

In [11], a dedicated logic called MMM for "Model and MetaModel logic" is used to express constraints and patterns. This logic can express *causal obligations* and can manipulate entities from both M1 and M2, but it is not based on OCL. The authors give a MMM specification of a Subject/Observer cooperation that includes both structural and behavioral aspects. However, the notion of role does not seem to be supported. Without roles, the generic form of a pattern cannot be completely represented, limiting this interesting approach to the specification of particular occurrences of design patterns.

Another research effort in precise representation of design patterns was presented by Tommi Mikkonen [10]. He proposes to formalize temporal behaviors of patterns using a specification method named DisCo. An interesting aspect of his work is that its formalism allows pattern occurrences to be combined through refinements between pattern definitions.

4.2 Conclusion

The use of meta-level collaborations and constraints, instead of the suggested parameterized collaborations, allows a more precise representation of design pattern structural and behavioral constraints. However, one may argue that this approach is not appropriate since we accomplish changes in the UML meta-model which is supposed to be standardized and static. Although this observation is true, it is also true that our approach does not change the UML abstract syntax. This is because the representation of a design pattern as a meta-model collaboration does not add new modeling constructs to UML. It only adds a way to enforce particular constraints among existing constructs.

Our approach also fits quite well with the Profile mechanism. A collection of design patterns, modeled with meta-level collaborations, could be provided as a Profile to a UML CASE tool which could reuse the pattern definitions.

In its current incarnation at the OMG, the UML is ill-equipped for precisely representing design patterns: even the most classical GoF patterns, such as Observer, Composite or Visitor cannot be modeled precisely with parameterized collaborations in UML 1.3. In this paper, we have proposed a minimal set of modifications to the UML 1.3 meta-model to make it possible to model design patterns and represent their occurrences in UML, opening the way for some automatic processing of pattern applications within CASE tools.

We are implementing these ideas in the UMLAUT tool (freely available from `http://www.irisa.fr/pampa/UMLAUT/`) in order to better document the occurrences of design patterns in UML models, as well as to help the designer to abstract away from the gory details of pattern application. Because in our proposal the "essence" of design patterns can be represented at the meta-level with sets of constraints, we can also foresee the availability of design pattern libraries that could be imported into UML tools for application into the designer's UML models. We are starting to build such a library as an open source initiative.

References

1. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
2. Jan Bosch. Language support for design patterns. In *TOOLS Europe'96*, pages 197–210. Prentice-Hall, 1996.
3. James O. Coplien. *Software Patterns*. SIGS Management Briefings. SIGS Books & Multimedia, 1996.
4. Dino Distefano, J.-P. Katoen, and Arend Rensink. On a temporal logic for object-based systems. Technical report CTIT 00-06, University of Twente, 2000.
5. Amnom H. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, University of Tel Aviv, 1999.
6. Amnon H. Eden, Amiram Yehudai, and Joseph Gil. Patterns of the agenda. In *LSDF97: Workshop in conjunction with ECOOP'97*, 1997.
7. Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In *ECOOP'97*, volume 1241 of *LNCS*, pages 472–495. Springer-Verlag, Jyväskylä, Finland, June 1997.

8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, MA, 1995.
9. Görel Hedin. Language support for design patterns using attribute extension. In *Workshop on Language Support for Design Patterns and Object Oriented Frameworks (LSDF), ECOOP'97*, pages 209–231, 1997.
10. Tommi Mikkonen. Formalizing design patterns. In *ICSE'98*, pages 115–124. IEEE CS Press, 1998.
11. Claudia Pons, Gabriel Baum, and Miguel Felder. Integrating object-oriented model with object-oriented metamodel into a single formalism. In *Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, pages 155–167. TUM-I9813, 1998.
12. Sita Ramakrishnan and John McGregor. Extending OCL to support temporal operators. In *Proceedings of the 21st International Conference on Software Engineering (ICSE99) Workshop on Testing Distributed Component-Based Systems, LA, May 16 - 22, 1999*, 1999.
13. Trygve Reenskaug. UML collaboration and OOram semantics, November 1999. http://www.ifi.uio.no/~trygver/documents/991108-UML/uml-collaboration.ps.
14. UML RTF. *OMG Unified Modeling Language Specification, Version 1.3, UML RTF proposed final revision*. OMG, June 1999.
15. Jiri Soukup. Implementing patterns. In *Pattern Languages of Program Design*, pages 395–412. Addison-Wesley, Reading, MA, 1995.
16. G. Sunyé, A. Le Guennec, and J.-M. Jézéquel. Design pattern application in UML. In E. Bertino, editor, *ECOOP'2000 proceedings*, number 1850, pages 44–62. Lecture Notes in Computer Science, Springer Verlag, June 2000.

A Appendix: Defining the *context* of OCL expressions with Collaborations

A.1 Collaborations as context for reusable OCL expressions

Section 7.3 of the UML documentation [14] defines what the *context* of an OCL expression is. The context specifies how the expression is "connected" to the UML model, that is, it declares the names that can be used within the expression.

UML proposes two kinds of context for OCL constraints:

Classifiers to which an <<invariant>> can be attached. The OCL expression can refer to "self" and to all subexpressions reachable from "self" using the navigation rules defined in Sect. 7.5.

Behavioral Features (Operations or Methods) to which a <<precondition>> and/or a <<postcondition>> can be attached. The OCL expression can refer to "self" and to all formal parameters of the behavioral feature. Note that although OCL postconditions can express that a new object has been created (using oclIsNew()), the context does not provide any way to declare a local variable that will denote the newly created object.

A Collaboration can also be attached to a Classifier or to a Behavioral Feature. The collaboration can described how the behavioral feature is realized,

in terms of roles played by "self" and the various parameters. One can also use the predefined stereotypes <<self>>, <<parameter>>, <<local>>, or <<global>> to make the roles more explicit within the collaboration.

Note that a given role can sometimes be played by several instances in a collaboration occurrence, in the limits imposed by the multiplicy of the role, and is then represented with a "multi-object" box, resembling stacked objects.

This suggests that Collaborations could well be used systematically as a precise graphical representation of the context of a OCL expression. Each parameter maps to a role having the parameter's type as base. A parameter which would be an OCL sequence of objects of a given type would map to a role with a multiplicity greater than one. Auxiliary variables introduced by "let expressions" (see 7.4.3 of [14]) can also be represented by corresponding roles in the collaboration.

A.2 Binding a parameterized OCL expression to the model

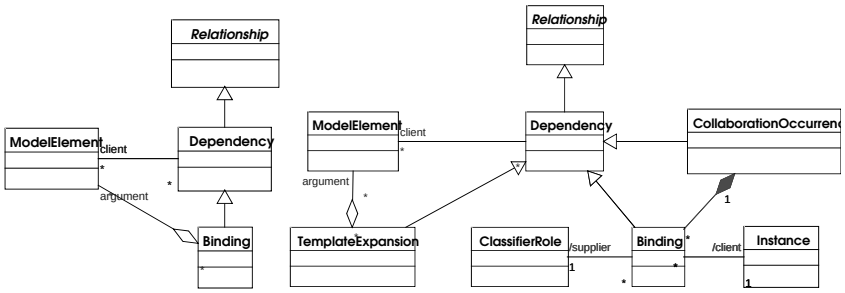
The way particular instances are attached to the respective roles is not very clear in the UML documentation: When a behavioral feature is called, the target of the call action and its effective arguments are supposed to be bound to the corresponding roles, but this is left implicit.

UML is apparently missing a construct to *bind* an instance to a role. The UML notation describes "instance level collaborations" (see Fig. 3-52 p314) as object diagrams representing snapshots of the system, where the roles of the objects are indicated in the object-box. But the mapping onto the UML abstract syntax is not described: how are Instances bound to ClassifierRoles?

Without further information, the only plausible mapping we found is to add the ClassifierRole (which is a kind of Classifier) to the set of current types of the Instance playing this role. Although this correctly reflects the dynamic nature of roles, this mapping explicitly relies on multiple and dynamic classification, which might be deemed too sophisticated, and is not well-defined in the UML context. Another disadvantage of this mapping is that it is too fine-grained: There is no way to *group* individual bindings together and say that they all belong to the same collaboration. This can cause confusion if a snapshot presents a set of instances participating in more than one collaboration at the same time.

Note also that the existing Binding construction of UML relates to generic template instantiations (see Sect. 1.1), which is a completely different matter altogether. We suggest that it be renamed as TemplateExpansion, which would better reflect its semantics, and reserve the name Binding for a new construct whose purpose is to bind an instance to a role, with a semantics equivalent to the dynamic classification alternative presented above. Note that it is desirable that individual Bindings be grouped together to form the whole effective context of a collaboration occurrence. UML1.1 offered the possibility to have *composite dependencies*, but this valuable capability has apparently been removed during the transition to UML1.3. We propose that it be brought back in.

A very similar approach is proposed in [13]: an InstanceCollaboration construct is used to group a set of Instances, while the relation between role and instance is expressed using classification instead of an explicit dependency.



(a) Current UML 1.3 (b) Proposed modifications

Fig. 6. Meta-model modifications for collaboration occurrences

Additional well-formedness rules associated to roles and bindings

[1] The number of instances playing a given role must not violate the multiplicity constraint of the role.

```
context ClassifierRole inv:
        let nbOfInstances =
                self.supplierDependency->select(b |
                        b.oclIsKindOf(Binding))->size()
        in (self.multiplicity.ranges->forall(r |
                        r.lower <= nbOfInstances
                        and nbOfInstances <= r.upper))
```

A.3 Graphical representation of bindings

There could be several ways of representing bindings between instances and roles.

1. Using the "instance level collaboration" idea of putting the role in the object-box. This solution is appropriate in some circumstances and is already known, and so is worth keeping.
2. Using a dependency arrow with a <<bind>> stereotype connecting the instance on the object diagram and the role on the collaboration diagram. This solution is not very attractive because it is too fine-grained and also requires both diagrams to be present together.
3. Reusing the dashed-ellipse notation originally proposed to represent instantiations of template collaborations, while changing the mapping of the ellipse onto the abstract syntax: The ellipse would represent the composition of all individual bindings, while each line going out of the ellipse would map to an individual binding dependency between the instance at the end of the line and the role whose name is given by the line label. This notation actually is generalizable to any composite dependency.