Design pattern application in UML

Gerson Sunyé, Alain Le Guennec, Jean-Marc Jézéquel

Design Patterns Application in UML

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

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

Abstract. The Unified Modeling Language (UML) currently proposes a mechanism to model recurrent design structures: the parameterized collaborations. The main goal of this mechanism is to model the structure of Design Patterns. This is an interesting feature because it can help designers to point out pattern application without spending time with intricate design details. Moreover, it can also help designers to better document their systems and to manage their own design pattern library, which could be used in different systems or projects. However, from a tool perspective, the semantics associated to parameterized collaborations is still vague. To put it more precisely, the underlying representation of a design pattern and of its application, and the binding between these two levels is not exactly defined, and therefore, can be interpreted in different ways. This article has two purposes. First, we point out ambiguities and clarify some misunderstanding points concerning parameterized collaborations in the "official" UML literature. We also show the limits of this mechanism when effectively modeling design patterns. Second, we propose some workarounds for these limits and describe how a tool integrating this mechanism could help with the semi-automatic application of design patterns.

1 Introduction

Design patterns [11] integration into a modeling language is a tempting idea. A simple modeling construct allowing to explicitly point out participant classes of a design pattern could help designers in many ways. Besides the direct advantage of a better documentation and the consequent better understandability of a model, pointing out the existence of a design pattern allows designers to abstract known design details (e.g. associations, methods) and concentrate on more important tasks.

Another tempting idea, consequent to the first one, is to provide tool support to this modeling language and therefore, to design patterns. The automatic implementation of patterns can help overcome some adversity encountered by programmers [27] [4] [10]. More precisely, a tool can ensure that pattern constraints are respected (e.g. that a subject always notifies its observers when it is modified), avoid some implementation burden (e.g. creating several forwarding methods in the Composite pattern) and even recognize pattern within source code, avoiding them to get lost after their implementation.

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 44–62, 2000.

The UML community succumbed to the first temptation: the latest version of the Unified Modeling Language [25] has improved the *collaboration* design construct in order to provide better support for design patterns. Indeed, the two conceptual levels provided by collaborations (i.e. parameterized collaboration and collaboration usage) fit perfectly 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. The application of this solution i.e. the terminology and structure specification into a particular context (so called instance or occurrence of a pattern) can be represented by collaboration usages.

However, UML parameterized collaborations suffer from a lack of precision, which constrains the effective benefits of tool support and makes the second temptation less seductive.

Section 2 is dedicated to the representation of design patterns in UML, and tackle a number of ambiguities that hinder tool support: Pattern occurrences are presented in Sect. 2.1, and it is explained how the occurrences of a pattern are linked to its general description in a precise way. The general description of patterns is supported in UML through the notion of parameterized collaborations, and Sect. 2.2 is dedicated to the syntactic and semantic issues arising from the use of collaborations in the context of design pattern. The limits of UML collaborations are carefully analyzed in Sect. 2.3. Ideas to overcome the shortcomings of collaborations are sketched in Sect.2.4, providing some guide-lines to model the "essence" of design patterns more accurately.

Once the most important issues pertaining to the presentation of design patterns in UML have been pointed out, better modeling of design pattern becomes possible. Effective tool support for UML design patterns is proposed in Sect. 3. Section 3.1 presents the main features that a user is likely to expect from an effective design pattern tool. Relying on the transformation framework provided by the UMLAUT prototype (presented in Sect. 3.2), we show in Sect. 3.3 how a metaprogramming approach allows for powerful manipulations of design patterns, easing the task of designers significantly.

2 Design Patterns and UML Collaborations

2.1 Representing Occurrences of Design Pattern

Had it not provided some support for the notion of pattern, it would have been hardly possible for UML to sustain its role as a unifying notation for object-oriented modeling. Therefore the abundant documentation on UML has sections wholly dedicated to patterns. Figure 1 presents an example of what represents an occurrence of the Composite design pattern, as given in the UML Reference Guide [26]:

The UML 1.3 notation for occurrences of design patterns is in the form of a dashed ellipse connected by dashed lines to the classes that participate in the pattern.

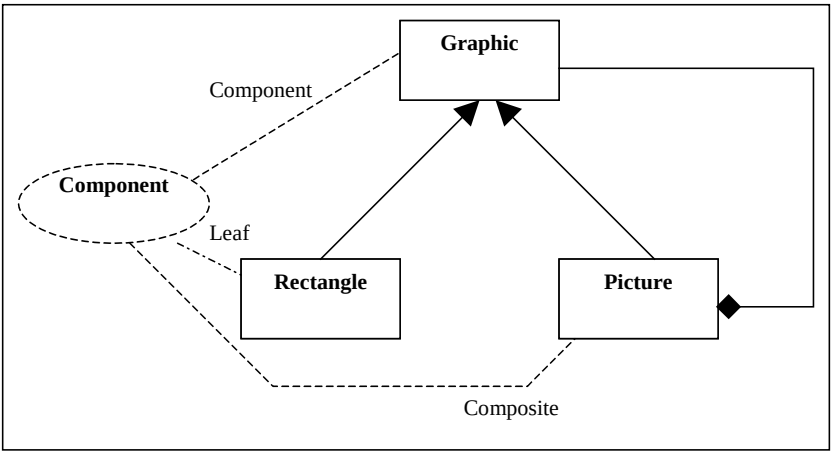

Fig. 1. An occurrence of the Composite Design Pattern

The interpretation of this model seems obvious, even though the Composite design pattern has been (erroneously) renamed as "Component": a Picture is composed of a set of Graphics (Rectangles or other Pictures). Objects instances of class Rectangle can only play the role of a Leaf, and hence cannot contain other Graphics. An experienced designer will also understand that Picture implements methods that will forward every message it receives to its components. In the context of this pattern occurrence, adding another class (say Circle) that also plays the Leaf role looks simple: Figure 28-4 in the UML User Manual [3] shows an example of a similar pattern occurrence, where a same role is assigned to different classes, by simply using as many dashed lines with the same label as necessary.

However, a small remark on that same page explains that "these [two] classes are bound a little differently than the others". This is an evidence that the apparent simplicity actually hides a lot of complex representation issues (see section 2.3 for a detailed discussion). Of course, the user should be shielded from those details, but the UML tool designer is not.

Note that Fig. 1 does not explain how implementation trade-offs are set and what the benefits of representing an occurrence of a design pattern are, other than for documentation purposes. Actually, UML does not support the representation of implementation trade-offs. The designer has no other choice but using comments to point these out.

Of course, the designer can reuse the pattern occurrence symbol for a given pattern any number of times, with a different binding for each new context in which the pattern appears (UML User Manual [3], p.388).

So far we have only seen how a pattern occurrence is made explicit in a model thanks to the dashed ellipse symbol. Two fundamental issues still remain to be solved:

1. Specifying how a pattern occurrence refers to the corresponding pattern specification. That is, we should give a precise meaning to dashed ellipses and dashed lines.
2. Specifying the pattern itself as formally as possible in UML. The UML actually provides a mechanism for this purpose, based on collaborations and genericity. It also provides a mapping of dashed ellipses and lines in this context. However, we will see in section 2.2 that this mechanism entails some confusion and suffers from many shortcomings.

2.2 The Official UML Proposal: Parameterized Collaborations

Design patterns are supposed to be modeled using parameterized collaborations, which are rendered in the UML in a way similar to template classes (UML User Manual [3], p.384). According to [3] p.387, three steps are needed to model a design pattern:

1. Identify the common solution to the common problem and reify it as a mechanism;
2. Model the mechanism as a collaboration, i.e. a namespace containing its structural, as well as its behavioral aspects;
3. Identify the elements of the design pattern that must be bound to elements in a specific context and render them as parameters of the collaboration;

The last two steps give an idea of how a design pattern is supposed to be modeled (and how a collaboration editor might work).

A collaboration is defined in terms of *roles*. The structural aspects of a collaboration are specified using ClassifierRoles, which are *placeholders* for objects that will interact to achieve the collaboration's goal. As a placeholder, a role is similar to a free variable or to a formal parameter of a routine. It will later be bound to an object that *conforms* to the ClassifierRole. Several objects can play one given role at run-time (constraints on the actual number are specified by the multiplicity of the classifier role) and each of them must conform to the classifier role. ClassifierRoles are connected by AssociationRoles, which are placeholders for associations among objects.

The way conformance of an object to a specific role is defined is particularly interesting, and this is where the notion of *base* of a role intervenes. A ClassifierRole does not specify exhaustively *all* the operations and attributes that an object conforming to this role must have. Only the features strictly necessary to the realization of the role are specified as features available in the ClassifierRole. Therefore, the ClassifierRole can be seen as a restriction (or projection) of a conforming object's "full" Classifier to the needed subset of features. Actually, UML imposes that roles be defined *only* as a restriction of existing classifiers and there are OCL rules in the meta-model (see [25] p.2-108) that enforce this view. The classifier(s) that a ClassifierRole is a restriction of is called the *base(s)* of the role.

An object is said to conform to a particular role if it provides all the features needed to play this role, that is, all the features declared in the ClassifierRole.

Although this is not strictly required (as explained in [25] p.2-113) any object that is an instance of a role's base classifier will by definition conform to this role.

On page 199 of [26], the authors propose a model for the Composite design pattern [11] (cf Fig. 2). This model is composed of three roles (Component,

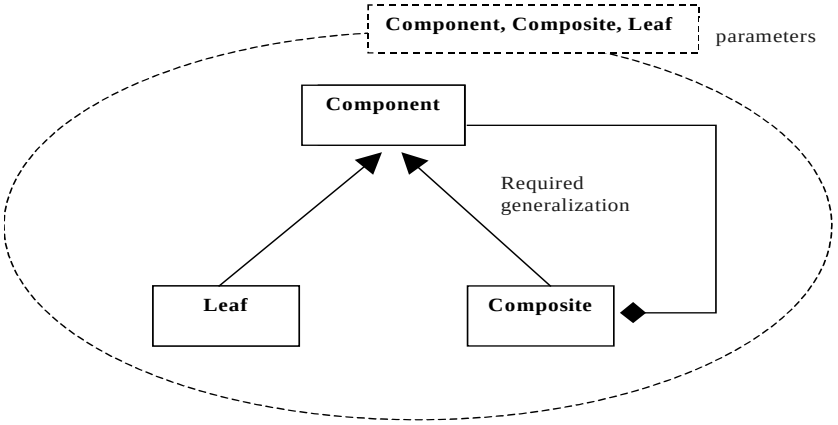

Fig. 2. Composite Design Pattern as a Parameterized Collaboration

Leaf and Composite). The reusability of a design pattern is expressed using the genericity mechanism: to make the above description of Composite context-independent and hence reusable in the context of geometric shapes, the base classifiers of each role are turned into template parameters of the collaboration. Putting the pattern in context simply consists in *binding* the template collaboration to the user model by providing actual arguments for template parameters. This is the official mapping given in [25] for the dashed ellipse symbol in Fig. 1, the actual arguments being inferred by the dashed lines. The label on each dashed line leaving the ellipse corresponds to the name of the role the actual argument will be a base of. With the conformance as defined above, the binding ensures that any instance of the Graphic classifier can play the role of a Component in the collaboration resulting from the template's instantiation represented in Fig. 1.

The parameterized collaborations cannot however be bound with just any actual classes. The participant classes must respect the *fundamental constraints* of the pattern. Several kinds of constraints can be imposed by the collaboration using a set of *constraining elements*:

– A generalization between two (formal generic) base classifiers means that a similar inheritance relation must also exist between the two corresponding actual classifiers used in any binding of the collaboration. This is just the

UML version of *constraint genericity*. For example, the classes acting as bases for the role Composite and Leaf must be specializations of the class acting as base for the role Component. The inheritance relationship between actual classifiers imposed by such a constraint need not be direct, though.
- An association among two (formal generic) base classifiers means that a similar association must also exist between the two corresponding actual classifiers used in any binding of the collaboration. Such associations among base classifiers are very likely to act as bases for association roles within the collaboration.

2.3 Limitations of Parameterized Collaborations

However, there are severe limitations to the expressive power of constraining elements associated to a collaboration:

Constraints on Generalizations: The graphical representation of collaboration superimposes classifier roles with their corresponding bases, which makes the use of generalization arrows ambiguous. Contrary to generalization relationships between their respective bases, generalization relationships between the classifiers roles themselves do not bring supplementary constraints: they just mean (as usual) that the child role inherits all features of its parent(s) (see [25] p2-105). This graphical ambiguity is the root of confusing descriptions in the UML books themselves [26]. Consider the following quotes:

> In a parameterized collaboration (a pattern) some of the classifier roles may be parameters. A generalization between two parameterized classifier roles indicates that any classifiers that are bound to the roles must satisfy the generalization relationship.
>
> (...) for the common case of parameterized roles, the template is bound by specifying a class for each role.

This can lead the readers to think that the template parameters are the classifier roles, when only base classifiers of the roles actually are template parameters (if a role were a template parameter, then the corresponding actual argument of the binding should also be a role, not simply a classifier. This is possible only if the binding is nested within a broader collaboration that will provide the actual role).

Constraints on Associations: The UML documentation (see e.g. [26]) does not consider it useful to make the bases of association roles template parameters of the collaboration. It is assumed that the bases of association roles can be deduced automatically from the existing associations among the base classifiers given when the template collaboration is bound. However, there are cases when this assumption does not hold: when there are several candidate associations,

or when there is no (direct) association. The former case forces an arbitrary choice, while the latter case requires the creation of a <<derived>> association to provide the necessary shortcut through available (indirect) associations. For instance, in the Observer pattern, there could be a mediator object between an observee and an observer, and the association between observee and observer would be computed by going through the mediator. The way an association can be derived (or computed) from existing but indirect ones is explained in [25] p2-19.

Constraints on Available Features: To take part in the collaboration, a role must dispose of a set of available features, which it gets from its base(s) classifier(s) (see [25] p2-108). But if the base is itself a generic parameter, where do the available features come from? Although this problem has not been raised so far in the UML literature, we can envision several possibilities:

1. Defining a special-purpose classifier associated with the template collaboration. This classifier would hold the features needed by a given role, and a constraining generalization relationship would ensure that the actual classifier used as a base for this role is a subclass of this special class. But then, since a ClassifierRole is a special kind of Classifier, why not instead simply define the needed features within the ClassifierRole itself and get rid of the base altogether? (see more on this alternative solution later on).
2. Defining *compound template parameters* and associated conformance rules for the corresponding actual argument in a binding. The available features would then come from *within* the template parameter. The number of so-called well-formedness rules involved in defining the conformance to such template parameters would probably be dissuasive.
3. Turning all needed features into template parameters (the UML does not impose any constraints on the kind of entity allowed as template parameters). We would then need a set of supplementary Constraints to ensure that the actual arguments for the features are actually owned by the appropriate actual arguments for the base classifiers. Constraints written in the Object Constraint Language (OCL [29]) would be added in the set of constraining elements of the collaboration (a Constraint with an upper case 'C' is a kind of UML modeling element, and is represented by an expression enclosed in curly braces.) Note that such a constraint needs access to the meta-level, which is not normally accessible from user-level models.

Constraints Involving any Number of Classifiers: In section 2.1, we have already recognized the need to somehow link several classes of the user model to a single role (or base thereof) of a design pattern. For example, one might want to show that several classes like Rectangle, Circle, Square, etc. all represent Leaves of the Composite design pattern in the context of geometric shapes. Figure 28-4 in the UML User Manual [3] similarly shows that several classes of the model are Commands of the Command design pattern.

Unfortunately, this is not as obvious as one can think. Indeed, the normal mapping of the pattern occurrence symbol is to bind exactly one actual class to exactly one template parameter (which is the base for a role). With bindings as defined in UML 1.3, it does not make any sense to provide several actual arguments for one template parameter. Moreover, templates have a fixed number of template parameters. As already mentioned in section 2.1, [3] avoids the problem by saying that the binding is different in this case: the dashed lines are supposed to map to generalization links which automatically turn all actual arguments into subclasses of a common class (respectively named Leaf and Command) provided by the collaboration itself. This exception to the normal mapping rule is rather confusing: note how the Leaf and Command classes are still represented in the upper-right corner of the collaboration symbol, as if they were still template parameters.

We argue that this proposed solution is more an ad-hoc workaround than a generalizable principle: it is applicable only because adding new *classes* during the binding for the Leaf or the Command roles does not really add any supplementary constraints on the structure of the design pattern solution. The only new information provided is that new kinds of *objects* can play these roles in the collaborations at run-time. It is remarkable that all examples of design patterns that we found in the UML literature are patterns for which this workaround is applicable.

But let us think about modeling the Visitor design pattern [11] in UML, an occurrence of which is given in Fig. 4. A parameterized collaboration will not be sufficient to represent "the" general solution, because the number of template parameters representing the nodes of the structure to be visited is frozen. But this number of nodes impacts the whole structure of the pattern, since the Visitor class must have the right number of accept operations.

We cannot address this problem with a simple workaround. We need to express constraints on the pattern that require full reflexive capability (if only to check the number of operations for example). Giving OCL expressions access to the meta-level (which is not normally accessible from user models) is a promising way to resolve the issue. The standard <<metaclass>> and <<powertype>> UML stereotypes might also prove useful as hooks into the meta-level.

Temporal or Behavioral Constraints: The mere existence of necessary operations or attributes for playing a role is of course not enough. Patterns also prescribe how operation calls, updates of attributes and other actions are organized to achieve a particular task. For instance, in the context of the Observer Design Pattern, any modification of the observee's state must eventually be followed by a call to `notify()`. UML Interactions can be attached to a collaboration to specify its behavioral aspects. An interaction is a partial order on messages and actions pertaining to the goal, and can be graphically represented as a sequence diagram. Such sequence diagrams often accompany the description of patterns by Gamma et al. in [11].

Being part of the parameterized collaboration, interactions are involved in the binding process, although the UML does not define how. There are two ways interactions can potentially affect the binding:

1. The interaction in the template might be transposed as-is in the resulting model, with template parameters substituted with actual arguments of the binding. This kind of macro expansion is the way UML genericity is supposed to work (see [25] p.2-26).
2. A more sensible approach is to consider the interactions as a new kind of constraint that must be respected by the actual arguments. Of course, the actual participants in the binding might satisfy the constraints as a more or less direct consequence of their own behavioral organization. For instance, a completely unrelated operation call might be inserted between two calls prescribed by the pattern. This of course should not invalidate the pattern usage, since such a situation is bound to happen in all but trivial situations.

It should now be clear that interactions attached to collaborations ought to be interpreted as behavioral constraints on participants, not unlike temporal logic formulas. Satisfaction of these constraints or formulas should be part of the conformance rules of pattern bindings, but UML 1.3 does not provide many hints on this issue. Some recent work [20] on formalizing UML collaborations and interactions shares this view of conformance of objects to collaboration roles.

To emphasize the formula-like nature of these constraints, an interesting approach worth investigating is to turn them into textual constraints written in a variant of OCL extended with temporal logic operators. Such an extension to OCL is proposed in [21].

The notion of base is ad hoc We have seen that the notion of base classifier of a role was at the heart of design pattern definition in UML, since roles are defined with respect to their base, and reusability is provided by making these bases formal parameters of the collaboration which then becomes a generic template.

We have also just seen that this use of bases as template parameters seriously complicates the way roles are specified in terms of available features, and suffers from the inherent limitations of template instantiations (e.g., fixed number of parameters in a UML Binding).

These problems suggest that the notion of base is probably not the right way to relate roles and classifiers of objects that will conform to the roles. Making roles dependent on pre-existing, "external", base classifiers when they could have stood on their own, looks suspicious. These dependencies impair the reusability of the collaborations, since they can't be reused without the corresponding bases, and they require clairvoyance on the part of the designer if several design patterns are to be combined (they might need shared bases).

The dependency between roles and classifiers should actually be in the opposite direction, as it is very likely that the classifier of an object will be designed by first carefully considering all the roles that the object may play in the various collaborations of the system. Hence the classifier of this object will be obtained by *merging* the roles' specifications, not the other way round.

The UML notion of *Realization* would be more appropriate than the notion of *Binding* of template parameters to express the fact that a given classifier realizes a set of roles. Moreover, realizations can specify a *mapping* expression to describe precisely how the classifier should realize the roles. We are investigating how the transformation functions proposed in section 3.3 would fit in this context.

2.4 Constraints as the Essence of Patterns

On the one hand, the various limitations listed above make it impossible for UML parameterized collaborations to precisely specify some of the more interesting constraints of design patterns. Their expressive power is limited to the prescription of associations and generalization links, and to a certain extent, the availability of features. Undoubtedly, more sophisticated constraints need access to the UML meta-model.

On the other hand, the static structure of collaborations (and of associated interactions if they are not considered as constraints) entails many choices that are not fundamental to a pattern itself, but are specific of some of its reified solutions. This prevents UML collaborations from representing only the *essence* of a pattern, free of any premature choices. All diagrams representing patterns or pattern occurrences in the UML literature fall short of this ambitious goal because of the over-specification side-effect of collaborations.

The essence of a pattern is what Eden [8] calls a "leitmotiv", that is, the intrinsic properties of a pattern *and nothing more*. These properties are common to all variants of a given pattern reification. Therefore, they should be expressed as general constraints over such reifications, which presumably involves meta-level OCL-like expressions and temporal logics.

How an enhanced OCL would fit together with UML collaborations and genericity to fully represent the essence of a pattern is at the heart of accurate UML specifications of patterns, and is still an active research topics of the authors.

A transformation tool would ideally provide this level of pattern modeling, and would help the user solve the corresponding constraints (or offer transformations implementing them, with meta-programs). The next section further elaborate on this idea of sophisticated tool support for UML design patterns.

3 Towards a UML Pattern Implementation Tool

The goal behind the above study of parameterized collaborations (and their limits) is to provide effective support for design patterns in a UML tool. But before extending the description of automatic design patterns implementation, let us dispel some possible misunderstanding concerning the integration of pattern in a CASE tool. According to James Coplien [7] 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. 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. We are also not trying to discover which

implementation variant is the most adequate to a particular situation, but to discharge programmers from the implementation of recurrent trivial operations (e.g. message forwarding) introduced by design patterns.

Consequently, our goal is to propose a tool that allows designers to model the structure of design patterns, to explicitly identify the participant classes of a pattern application and to map the structure of a pattern into any application of this pattern. Once this is possible, the tool can automate different approaches of patterns use. According to Florijn et al. [10], a pattern-based tool can follow three main approaches:

- Recognition: In this approach, the tool recognizes that a set of classes, methods and attributes corresponds to a design pattern application and points this out to the designer;
- Generation: Here, the designer chooses a pattern she wants to apply, the participant classes and some implementation trade-off and receives the corresponding source code;
- Reconstruction: The former approaches can be merged into this third approach. Here, the tool modifies a set of classes that looks like a pattern application into an effective pattern application. This modification implies the addition or modification of classes, attributes and methods.

Since our goal is to integrate design patterns into a UML tool and therefore adopt a generative approach, our present interest concerns only the last two approaches.

The idea behind the second approach is that the pattern solution could be seen as a sequence of steps. Therefore, a pattern implementation would be obtained if the tool follows these steps. Actually, the implementation is more complicated than that since the tool should verify if the pattern has not already been partially implemented. Moreover, the solution is not unique, the implementation may change according to certain trade-offs.

The third approach is very close to Opdyke's refactorings [19], i.e. operations that modify the source code of an application without changing its behavior. One of the many difficulties of this approach is that design patterns specify a set of solutions to a problem, but do not (or rarely do) specify a common situation to which the pattern should be applied. This seems to be reasonable, since the problem described by a pattern can be solved in several other manners and it would be impossible to catalog each manner.

Another approach, which was not enumerated by Florijn et al. concerns design patterns validation. More precisely, design patterns have implicit constraints that could be automatically verified. For instance, in the Observer pattern, a tool could verify if every method that modifies the subject also notifies its observers.

In our perspective, a design tool that support design patterns should, on one hand, provide high level transformations that help designers to apply a design pattern and, on the other hand, ensure that the applied solution remains consistent during the design process. In the next sections, we will further explain what exactly we want our tool to do.

3.1 A Pattern Tool in Action

In order to further describe the rationale behind our pattern tool, let us take the role of a designer using her favorite UML CASE tool. More specifically, she is in the middle of the design of her application and her model contains many classes and operations.

Suppose that she now decides to apply a specific design pattern to her model. This could be done in two different ways.

The first way is to select a class and choose the transformation she wants to apply to this class. In this case, the designer knows exactly what she wants to do and uses the pattern application as a macro. This transformation could be, for instance, the creation of a Composite class (i.e. the application of the Composite pattern). The tool will automatically create a new class, an association between the new class and the selected one and a set of forwarding methods in the new class. At the same time, the tool will create an usage of the Composite collaboration (defined previously) and assign the composite role to the new class and the component role to the selected class. This approach will be further explained in section 3.3.

The second way consists in creating an usage of a collaboration, which corresponds to the pattern she wants to use. In this case, she has already applied (partially or not) this pattern to her classes and wants to document it. She will manually determinate which classes participate to this pattern occurrence. Then, the tool will try to automatically bind non-specified participants (classes, features, associations, etc.) to the collaboration, and ask for the designer validation.

From then on, all the constraints inherent to the chosen pattern should be continuously checked as a background process. Any unsatisfied constraint should be clearly indicated: For each of them, an item would appear in a "to-do list", describing the action the user should take in order to make the pattern application complete and correct. To the extent that it is possible, the tool should propose some semi-automatic steps, to relief the user as much as possible. As soon as all constraints are satisfied, the user can proceed with code generation to obtain the final application.

3.2 UMLAUT's Transformation Framework

UMLAUT is a freely available tool dedicated to the manipulation of UML models. UMLAUT notably provides a UML transformation framework allowing complex manipulations to be applied to a UML model [13]. These manipulations are expressed as algebraic compositions of elementary transformations.

We propose the use of a mix of object-oriented and functional paradigm to develop a reusable toolbox of transformation operators. The general approach consists of two major steps. The first phase uses an iterator to traverse the UML meta-model instance into a linear sequence of model elements. The second phase maps a set of operators onto each element in this sequence.

In the context of the theory of lists, it has been shown that any operation can be expressed as the algebraic composition of a small number of polymorphic operations like *map*, *filter* and *reduce* [2].

The transformation process can be generalized into three stages: element selection, element processing and reductive validation. We can re-apply the first two stages repeatedly using composition of *map* and *filter* to achieve the desired results.

3.3 A Metaprogramming Approach

This metaprogramming approach consists in applying design patterns by means of successive transformation steps that should be applied starting from an initial situation up until a final situation is reached where the occurrence of the pattern is explicit. For instance, Fig. 3 presents a situation to which the Visitor pattern can be applied, i.e. a class hierarchy where several methods (optimize() and generate()) are defined by every class. Applying the pattern to this hierarchy means creating another class hierarchy to where these methods will be transferred. The final situation is presented in Fig. 4.

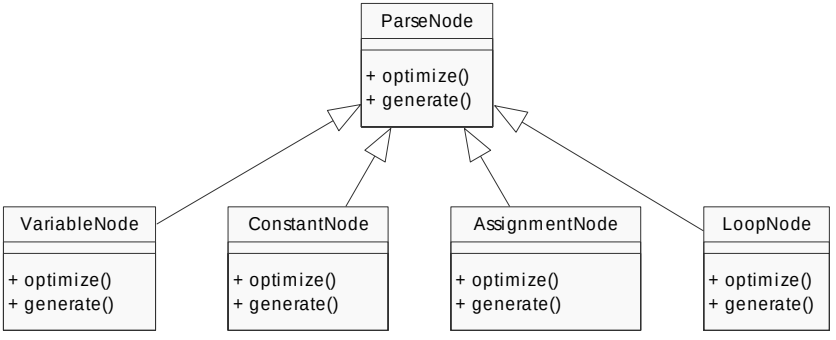

Fig. 3. The Visitor Design Pattern - Initial state

One may accurately notice that this transformation approach is not valid for every design pattern, since only a few patterns (e.g. Bridge, Proxy) mention an existing situation to which the pattern should be applied. This is true, this approach is not and does not intend to be universal. Our prime intent here is to provide UML designers meta-programming facilities that we believe every software designer should have [24].

The above reference to Smalltalk refactorings is not naive; we are strongly convinced that every development tool should have such facility and that this facility can help developers to apply design patterns. However, refactorings cannot be directly translated to UML for two reasons. First, refactorings were defined for programming languages and UML has several modeling constructs other than

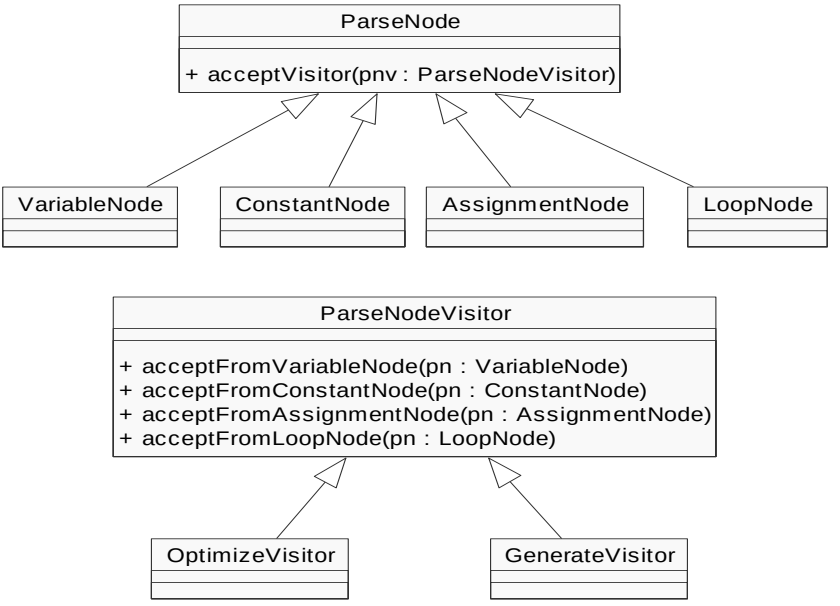

Fig. 4. The Visitor Design Pattern - Final state

classes, methods and variables. Second, the behavior of a UML method is described by a few diagrams and constraints whilst refactorings operate directly over Smalltalk code. Hence, some refactorings should be changed and new ones must be defined. Nevertheless, there is at least one advantage of using UML: OCL [29] can easily be used to define refactorings pre and post conditions. For instance, the addClass() refactoring [22] can be defined as:

```
Package::addClass(newClass:Name, superclasses,
                                  subclasses:Set(Class)):Class
pre:
not self.allClasses()->exists(name=newClass) and
subclasses->forAll(each | each.allSuperTypes()->
    includesAll(superclasses))
post:
result.name = newClass
self.allClasses()->includes(result) and
self.classReferences(result)->isEmpty and
result.allSuperTypes()->intersection(superclasses)->size() =
    result.allSuperTypes()->size() and
result.allSubTypes()->intersection(subclasses)->size() =
    result.allSubTypes()->size()
```

This function is interpreted as follows: a new class can be added to a package if this package does not already have a class named as the new one and if the future subclasses of the new class are subclasses of its future super-classes. After the class addition, the package will contain a class named as the new class, there

will be no references to the new class and the list of the new class super-classes will be exactly the same as the super-class list provided as a parameter.

We intend to allow designers to combine multiple refactorings in order to represent design pattern applications. UMLAUT currently provides a transformation framework [13], where UML model transformations can be specified. However, since this framework cannot be specialized at run-time (some compilation is needed), we expect to use a syntax close to OCL to describe refactorings and combine them.

To illustrate a transformation function, we present below a function which specifies the application of the Visitor pattern. This example was copied, with a few changes, from [23]. These changes were necessary since the original example applies to Smalltalk code whilst here it applies to UML models:

```
Class::addVisitor()
actions:
let abstractVisitor :=
    self.package.addClass(self.name+'Visitor', nil, nil).
self.allOperations()->forAll(operation |
    let concreteVisitor := self.package.addClass(
            operation.name+'Visitor', abstractVisitor, nil).
    self.allSubTypes()->forAll(subclass |
        subclass.allOperations()->select(subop |
            subop.hasSameSignature(operation)->forAll(op |
                op.move(concreteVisitor,'acceptFrom'+subclass.name).
                op.rename('acceptVisitor').
                op.pullUp())))).
```

This transformation applies to the class that is to play the role of element in the Visitor pattern. Its application to the ParseNode class operates as follows. At first, an abstract visitor class named ParseNodeVisitor is created. Then, for each method of the ParseNode class, the transformation will:

1. Create a concrete visitor, subclass of the abstract visitor;
2. Move all same signature methods (from subclasses) to this concrete visitor. This operation replaces the body of the original method by a forwarding method that simply calls the new method. The moved method is renamed and receives a new parameter allowing it to refer to the members of its original class.

A natural extension to this transformation would be the addition of a collaboration usage, specifying that ParseNode and ParseNodeVisitor are participant classes of an occurrence of the Visitor pattern.

4 Related Work

The rapid evolution of design patterns has been followed by several research efforts on pattern-based tool development. These efforts pursue different goals, such as: design pattern recognition [15] [5]); formal specification [18] [16] [1]; code reuse [27] [17]; and code generation [6] [28].

PatternWizard is one of the most extensive projects of design patterns specification, and has influenced our research work in several points. PatternWizard proposes LePUS [9] a declarative, higher order language, designed to represent the generic solution, or *leitmotif* , indicated by design patterns. In order to define the constructs of this language, the authors have analyzed the solution of all Gamma et al. design patterns and identified a set of common building block, called *tricks* (an extensive list of tricks is given by Amnon H. Eden in his PhD thesis [8]).

Tricks specify a sequence of operations over an abstract syntax language. Tricks are divided in three levels: Idioms, Micropatterns and Design Patterns. Idioms are the first level of tricks and operate over the abstract syntax language. They abstract language dependence. Tricks can be compared to refactorings in many ways, except in code generation: in opposition to refactorings, tricks can generate code. Micropatterns are the second level of tricks (they can be defined as a set of idioms). They represent simple mechanisms that appear repeatedly among design patterns such as, for instance, message forwarding. Finally, design patterns are the higher level of tricks and represent the *leitmotif* of design patterns.

The structure of design patterns can be more precisely defined by LePUS declarations, or formulae, than a simple class diagram. Indeed, a LePUS formula can define the multiplicity of each participant class in the design pattern application. Furthermore, the behavior of participant methods is precisely defined by tricks.

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 more intelligible than the LePUS formulae and its associated graphical language. Second, PatternWizard works at code level and is not integrated to any design model.

Such integration is proposed by Kim et Benner [14]. They propose to split design into two levels (both described in OMT). The *pattern* level is situated above the *design* level. Design level is composed by classes, their components and associations that represent together the result of design. Above this level, the pattern level defines additional semantics. The main idea is to link design constructs which participate in a pattern application to the structure representing the model itself. This rationale is very close to UML parameterized collaborations. However, unlike collaborations, a certain flexibility exists (and is defined). Pattern occurrences need not be totally isomorphic, they respect the concept of generalizable path, that takes into account the generalization links present in the design model.

Design and implementation integration is also provided by Fred [12], a development tool designed for framework development and specialization. Fred helps developers to specialize application frameworks, indicating hot-spots and inviting them to follow a sequence of steps, presented by a *working-list*. Doing this, Fred can reduce the time necessary to effectively use a framework. In Fred, developers can explicitly precise an occurrence of a pattern using links between

the pattern description and participant classes of this occurrence. After this, Fred proposes a set of template methods (also presented by a working-list) that should be completed to fully implement the pattern. Template methods describe constraints concerning the behavior of participant classes.

Finally, Jan Bosch proposes LayOM [4], a layered object model. LayOM intends to integrate design patterns, using an extended object model that supports the concept of layers. More specifically, the layers encapsulate a set of objects that intercept and treat sent messages. LayOM integrates two more concepts, category and state. A category is an expression that describes the characteristics of a set of possible clients, that should be treated likewise. A state is the abstraction of the internal state of an object. Message interception, provided by layers are appropriate to the implementation of some patterns, such as Adapter, for instance. LayOM generates C++ code.

5 Conclusion

The extensive study of the solutions proposed throughout the UML literature evidences many ambiguities in the use of parameterized collaborations and a lack of semantic foundation preventing systematic analysis and manipulation of design patterns in UML. We still consider however that sophisticated support for design patterns in a UML tool is not out of reach.

The knowledge that we acquired throughout the former development of a pattern implementation tool [28] and the implementation of the full UML meta-model in UMLAUT was extremely valuable when we started working on tool support for design patterns in UML. It helped us point out the crucial difficulties paving the way of automatic implementation of design pattern.

Getting a good grasp of the complexity underlying UML parameterized collaborations is definitely not a simple task, especially when they are used in the context of design patterns. We promptly searched for solutions in the abundant UML literature, but did not get complete, unambiguous, authoritative answers. Section 2 brings new insights on this topic, and also raises several issues for which there is no satisfying solution yet.

Since parameterized collaborations are not totally adapted to model design patterns, we now face a delicate choice between adapting the present semantics of collaborations or extending the UML meta-model with new constructs to bridge current semantic gaps. We are investigating this latter approach. in order to provide a more accurate way of binding Design Patterns and their occurrences, and a better support for implementation trade-offs and feature roles. In addition, we are specifying a set of UML specific refactorings and implementing them using our transformation framework.

In spite of our efforts, some questions are still left open:

First, keeping the binding between a general pattern description and the participants in a particular occurrence up-to-date might be problematic if the user is allowed to dynamically change the variant applied. Indeed, the set of

participants may also have to be changed, and so will the set of actual constraints to be satisfied.

Second, checking for constraint satisfaction is likely to be largely undecidable, or prohibitively expensive in term of computational resources. Model-checking techniques might however provide useful results to check some behavioral constraints.

Third, automation of the step needed to satisfy a constraint is possible in isolation, but when a modeling element takes part in more than one pattern occurrence, the number of possibilities the tool could suggest quickly becomes unmanageable.

At the present time, a working version of UMLAUT is freely available for download[1]. This version provides some model construction facilities, Eiffel code generation and evaluation of OCL constraints.

References

1. P. S. C. Alencar, D. D. Cowan, and C. J. P. Lucena. A formal approach to architectural design patterns. In J. Woodcock M. C. Gaudel, editor, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 576–594. Springer-Verlag LNCS 1051, 1996.
2. R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springler-Verlag, 1987.
3. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
4. Jan Bosch. Language support for design patterns. In *TOOLS Europe'96*, pages 197–210. Prentice-Hall, 1996.
5. K. Brown. Design reverse-engineering and automated design patterns detection in Smalltalk. *Technical Journal*, 1995.
6. F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2), 1996.
7. James O. Coplien. *Software Patterns*. SIGS Management Briefings. SIGS Books & Multimedia, 1996.
8. Amnom H. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, University of Tel Aviv, 1999.
9. Amnon H. Eden, Amiram Yehudai, and Joseph Gil. Patterns of the agenda. In *LSDF97: Workshop in conjunction with ECOOP'97*, 1997.
10. Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 472–495. Springer-Verlag, New York, N.Y., June 1997.
11. 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.
12. Markku Hakala, Juha Hautamäki, Jyrki Tuomi, Antti Viljamaa, and Jukka Voljamaa. Pattern-oriented framework engineering using FRED. In *ECOOP '98— Workshop reader on Object-Oriented Technology*, Lecture Notes in Computer Science, pages 105–109. Springer-Verlag, 1998.

[1] http://www.irisa.fr/pampa/UMLAUT/

13. Jean-Marc Jézéquel, Wai Ming Ho, Alain Le Guennec, and François Pennaneac'h. UMLAUT: an extendible UML transformation framework. In Robert J. Hall and Ernst Tyugu, editors, *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.
14. Jung J. Kim and Kevin M. Benner. A design patterns experience: Lessons learned and tool support. In *Position Paper, Workshop on Patterns, ECOOP'95*, Aarhus, Denmark, 1995.
15. C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Working Conference on Reverse Engineering (WCRE'96)*. IEEE CS Press, 1996.
16. Anthony Lauder and Stuart Kent. Precise visual specification of design patterns. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 114–134. Springer, 1998.
17. Marco Meijers. *Tool Support for Object-Oriented Design Patterns*. PhD thesis, Utrecht University, 1996.
18. Tommi Mikkonen. Formalizing design patterns. In *ICSE'98*, pages 115–124. IEEE CS Press, 1998.
19. William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.
20. Gunnar Övergaard. A formal approach to collaborations in the unified modeling language. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.
21. S Ramakrishnan and J 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 (WM3)*, Los Angeles, California, USA, May 1999. ACM press.
22. Don Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.
23. Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4), 1997.
24. Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk - why every Smalltalker should use the Refactoring Browser. *The Smalltalk Report*, 14(10):4–11, 1997.
25. UML RTF. *OMG Unified Modeling Language Specification, Version 1.3, UML RTF proposed final revision*. OMG, June 1999.
26. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
27. Jiri Soukup. Implementing patterns. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 395–412. Addison-Wesley, Reading, MA, 1995.
28. Gerson Sunyé. Génération de code à l'aide de patrons de conception. In Jacques Malenfant and Roger Rousseau, editors, *Langages et Modèles à Objets - LMO'99*, pages 163–178, Villeneuve s/ mer, 1999. Hermes. In French.
29. Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.