# Aspectual Template in UML

Gilles Vanwormhoudt, Olivier Caron, Bernard Carré

# Aspectual Template in UML

Gilles Vanwormhoudt[1,2], Olivier Caron[1], and Bernard Carré[1]

[1]LIFL, UMR CNRS 8022, University of Lille, France - 59655
Villeneuve d'Ascq cedex
[2]Institut TELECOM
{*Gilles.Vanwormhoudt, Olivier.Caron, Bernard.Carre,* }*@lifl.fr*

16 april 2013

**Abstract**

UML Templates allow to capture models whose some of their constituents are parameters. This construct is general enough to be used in many ways, such as generic class representation, Design Pattern modeling, view or aspect-oriented modeling (AOM). In this paper, we concentrate on this last usage and the specific characteristics of so called "Aspectual Templates". Such templates can be applied to enrich existing models as far as they conform to a required model. Template parameters are exploited here to specify some required model, so that they must be constrained to form a full model structure. After recall of UML templates and their metamodel, we present the specificities of their aspectual interpretation, existing works and identify the issues. Then we show how standard UML templates can be enhanced to capture aspectual ones. For this, a specialization of the UML template metamodel is detailed and OCL constraints are formulated due to this specific interpretation and its proper mechanisms. As a result, this metamodel specialization is fully compatible with the existing one so that aspectual templates are full UML standard ones. Finally, we present an algorithm which constructs the result of the application of an Aspectual Template to a model. This algorithm also works for aspectual template to template application. Presented results were implemented and made available in the EMF (Eclipse Modeling Framework) technology.

## 1 Introduction

After being considered only as documentation for a long time, software models are nowadays used as full artifacts. The MDA methodology (Model Driven Architecture [1]) identified the need to separate platform-independent modeling choices from platform-dependent ones in order to facilitate subsequent software generation, with respect to some "vertical" transformation chains. After this proof of concept, the MDE (Model Driven Engineering [16]) generalized the approach. It upgraded the status of models, from ingredients dedicated to MDA steps, to full first-class software objects, so reusable and composable. The challenge is to facilitate the capitalization of technology independent design efforts

and logic, then conversely, systematic "design by model reuse" methodology, in a productive and safe manner.

Once it was clear that software models could be isolated and composed, powerful technics arising from the programming world were considered to increase their capitalization and composition capabilities. We concentrate here on two main trends which contribute to this challenge. First, model parameterization [12, 28, 2] where a model exposes some of its ingredients as parameters. Fixing the parameters allow to obtain an instance model (component), which can then be composed. Second, Aspect-Oriented Modeling (AOM) [9, 24, 15, 29, 30], which issues from the application at the model level of the separation of concerns approach and aspect-oriented concepts [17].

The UML standard [27] contributes a lot to these trends through the "template" construct which allows to represent model schemes where some of their ingredients are parameters. A "binding" relationship allows to specify how a model is derived from a template through the substitution of these parameters. This binding can be partial. In that case, not all parameters are valued and remain parameters in the resulting model, so that it is itself a template. Indeed it is possible to bind models to constitutive templates but also templates to templates, in order to obtain richer ones. The power of the construct is general enough to render many needs which range from the modeling of generic classes (such as C++ templates) [10, 21], the capture of Design Patterns [25], View [12, 4] or Aspect Oriented Modeling [6, 22, 8, 23, 14].

We concentrate here on this last usage. AOM makes possible to define aspect models whose ingredients are intended to be injected into other models, as far as they conform to a required model. Template parameters are exploited here to specify this required model with the specific constraint that they must form a full model structure. This leads to so-called "Aspectual Templates", which can be applied to enrich existing models.

After recall (Section 2) of UML templates and their metamodel, we present their specific usage for aspect-oriented modeling (Section 3), existing works and identify the issues. In Section 4 we show how UML templates can be enhanced to represent aspectual ones. The obtained AOM engineering approach and its advantages are sketched. After that (Section 5), the specialization of the UML template metamodel dedicated to aspectual templates is detailed. The required constraints, due to their specific structure and mechanisms are formulated in OCL. In (Section 6) an algorithm is formulated for the construction of a consistent model resulting from the application of an aspectual template to a model. This algorithm also treats application of aspectual templates to (template) models. Finally, we present in Section 7 resulting technology that we offer in the EMF (Eclipse Modeling Framework) environment, before concluding.

# 2 Background on UML Templates

## 2.1 The UML Template Construct

In the UML Standard, a template is a model element which is parameterized by other model elements. Examples are classes or packages, so called respectively class templates or package templates. To specify its parameters, a template owns a signature. A template signature is a list of formal parameters where

each parameter refers to an element of the template model. Template elements have also a specific graphical notation which consists in superimposing a small dashed rectangle containing the signature on the top right-hand corner of the corresponding symbol.

Templates allow to obtain other model elements thanks to parameter substitution, declared in a dedicated "binding relationship". A binding relationship links a "bound model" to a template (from which it was obtained) through the specification of a set of template parameter substitutions that associate actual elements (of the bound model) to formal parameters (of the template). Constraints of the standard only impose that the type of each actual model element must be a subtype of the corresponding formal parameter one.

The binding relationship allows to set the existing link between a model and the template used for its construction. It is specified in UML as follows: "The presence of a TemplateBinding relationship implies the same semantics as if the contents of the template owning the target template signature were copied into the bound element, substituting any elements exposed as formal template parameters by the corresponding elements specified as actual parameters in this binding."([27], page 626). This implies that the content of the bound model is based upon the content of the template with any element exposed as a formal parameter substituted by the actual element specified in the binding. The correctness of this binding logic was formulated by OCL constraints in [3].

Figure 1 shows a class template on the left. This class, *Stack*, is graphically represented as a standard UML class with a dashed rectangle containing its signature. Here, the signature is composed of two formal parameters: *Element* of type *Class* and *Max* of type *int*. The right side of this figure shows the class *PlatesStack* which is bound to the *Stack* template through a "bind" relationship. This class is the result of the substitution of the template formal parameters *Element* and *Max* respectively to actual values *Plate* and *15*.

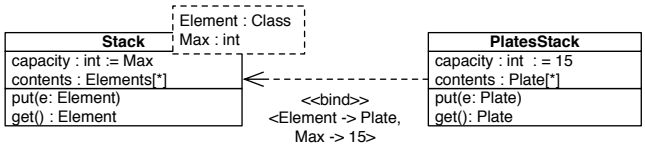


Figure 1: Class Template

Figure 2 shows an example of a package template, used here to model the well-known Observer Pattern (top of the figure). As indicated in the signature, the model is paramerized by the *Subject* and *Observer* classes, the observed *value* attribute (of *Subject*) and the $T$ type.

The rest of the figure illustrates the use of this pattern for the design of a Home Heating System, represented by the *HomeHeatingSystem* package. This package has it own content which is composed of *RoomSensor*, *HeatFlowRegulator*, *Furnace* classes and their relationships. The Observer Pattern Template was used here to install the dependency between *RoomSensor* and *HeatFlowRegulator*. This design choice is rendered by the binding relationship between *HomeHeatingSystem* and the *ObserverPattern* template which depicts the required parameter substitutions: *RoomSensor* to *Subject*, *HeatFlowRegulator* to *Observer*, the value *value* to *currentTemperature* and its type $T$ to *float*. As

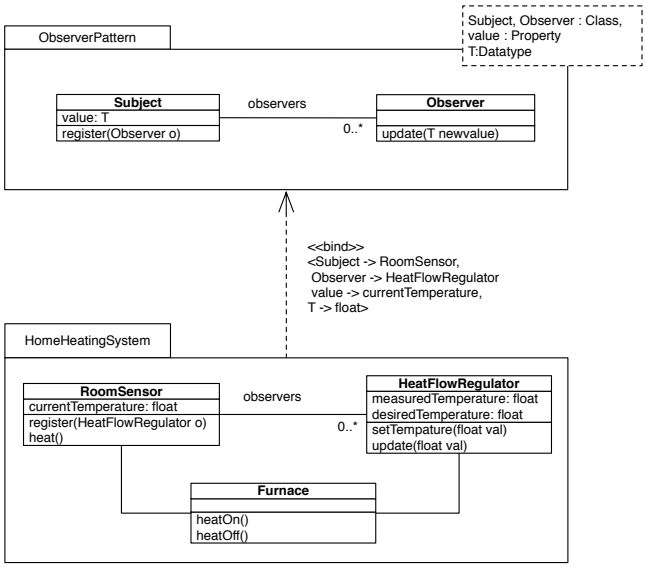


Figure 2: Package Template

a result of this binding, *HomeHeatingSystem* includes the model structure of the Observer Pattern, after substitutions were made. New ingredients were injected such as the *observers* association and the corresponding (correctly typed) *register/update* protocol. Note that actual classes of the bound model (*HomeHeatingSystem*) may have contents in addition to those specified by the formal parameters, due to their proper modeling context.

Finally, a bound element may have multiple bindings, possibly to the same template. In that case it is stipulated that the bound model gets the content of each binding considered in isolation. UML also introduces the notion of partial binding when not all formal template parameters are bound. In that case, the UML specification only states that the unbound formal template parameters are formal template parameters of the bound element ([27], page 634), which is itself a template.

## 2.2 The UML Template Metamodel

The Templates package in the UML metamodel [28] introduces four main classes for their structural representation: *TemplateSignature*, *TemplateableElement*, *TemplateParameter* and *ParameterableElement* (see Figure 3). *TemplateBinding* and *TemplateParameterSubstitution* metaclasses are both used to bind templates (see Figure 4).

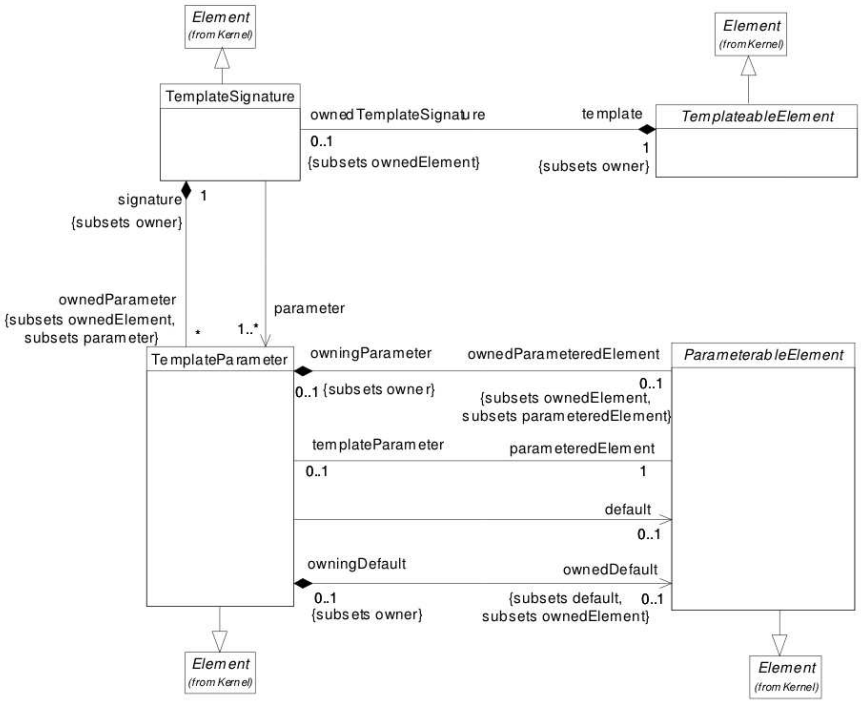


Figure 3: Template Metamodel

UML 2 elements that are subclasses of the abstract class *TemplateableElement* can be parameterized. *Classifiers*, in particular classes and associations, *Packages* and *Operations* are templateable elements.

The set of template parameters (*TemplateParameter*) of a template (*TemplateableElement*) are included in a signature *TemplateSignature*. A *TemplateParameter* stands a formal template parameter and exposes an element owned by the template thanks to the *parameteredElement* role.

Only parameterable elements (*ParameterableElement*) can be exposed as formal template parameters of a template or specified as actual arguments in a template binding. Such parameterable elements are: *Classifier*, *PackageableElement*, *Operation* or *Property*.

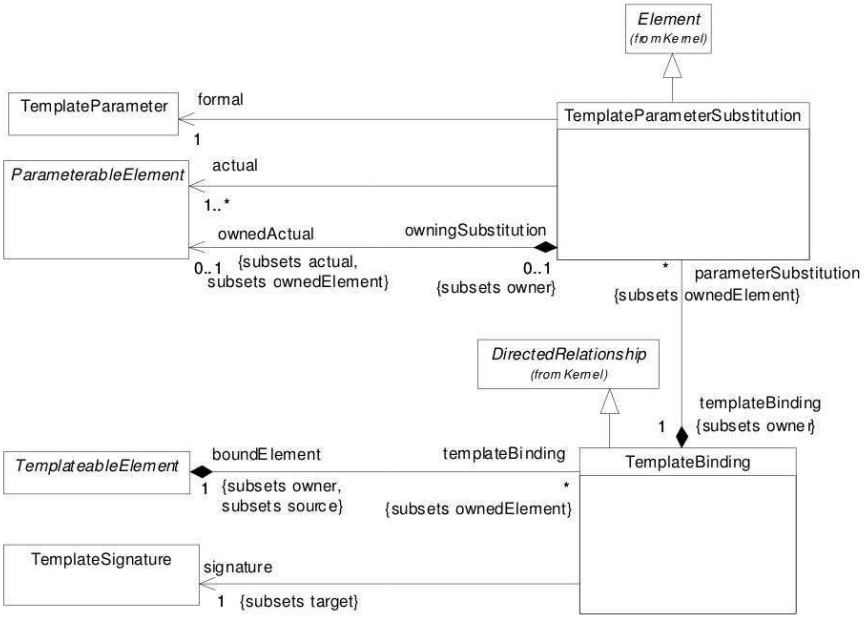


Figure 4: Template Binding Metamodel

The notion of template binding (*TemplateBinding*) describes the use of a template for a given system (cf. Figure 4). A template binding is a directed relationship labeled by the << *bind* >> stereotype from the bound element (*boundElement*) to the template (*signature*). The template binding owns a set of template parameter substitutions (*TemplateParameterSubstitution*). Substitutions associate actual parameterable elements of the bound to formal param-

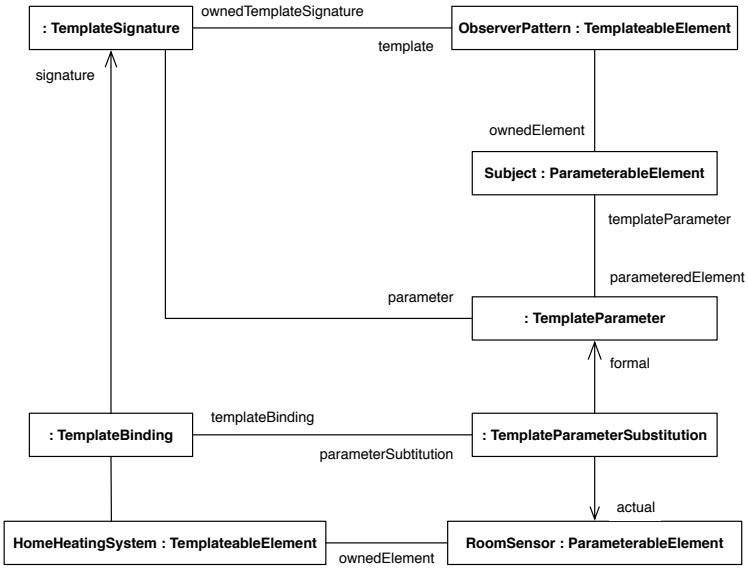


Figure 5: Excerpt of the Object Diagram for the *HomeHeatingSystem* Package

eters of the template signature.

Figure 5 shows an excerpt of the instantiation of this metamodel for the example described in Figure 2. It depicts the substitution between the *Subject* formal template parameter and the actual *RoomSensor* parameter of the bound *HomeHeatingSystem*.

Finally, the UML specification also introduces basic constraints for checking the correct definition of templates and their bindings. These constraints check that :

- All parameters of a signature are elements of the template.
- Formal parameter and corresponding actual arguments of a substitution have compatible metatypes.
- Each parameter substitution refers to a formal template parameter of the target template signature.
- A binding contains at most one parameter substitution for each formal template parameter of the target template signature.

As it will be explained later, these constraints or their intent which are general will be also valid for aspectual templates.

# 3 Template-based Aspect-Oriented Modeling

## 3.1 Existing works

The template construct allows to capitalize models which capture recurrent structures. Applications of this construct are numerous and range from the

modeling of generic classes to pattern formulation as exemplified previously but also Aspect-Oriented Modeling (AOM). Basically, AOM raises the idea of separation of concerns to the level of software models and applies aspect-orientation concepts to compose "aspect models" into various base models. Over the last years, many AOM technics have been proposed for both structural and behavioral models [18, 20, 29, 30]. All these technics provide a notion of aspect model and a process for weaving aspects with application models.

Few works have exploited the template construct for aspect-oriented modeling. In these works, so called "aspectual templates" aim to inject new functionalities into various base models. The capacity of such templates to expose some elements as parameters is exploited to specify the model structure required for making the weaving possible.

The Theme approach [8] proposes means for aspect-orientation in the analysis phase with Theme/Doc and in the design phase with Theme/UML. In Theme/UML, aspect models (called Themes) are specified using UML template packages containing class and sequence diagrams. The template parameters can be classes, operations or attributes. A relation (named "bind") is used to express the composition of a Theme and a base model. This relation binds the template parameters to concrete modeling elements of a base model, possibly using wildcard and multiple times.

France et al. [23] describe an aspect-oriented modeling technic in which aspect models are expressed using UML template packages containing class diagram, communication diagram and sequence diagrams. The approach is similar to Theme/UML but does not directly compose an aspect model (template) with a base model (called here "primary model"). Instead, a context-specific aspect model is first created by "binding" the parameters to application-specific values. It is this context-specific aspect model which is finally composed with the base model. During the composition, elements of same type and same name are merged to form a single one into the composed model. The approach also proposes "composition directives" which are intended to refine the default composition rules. They can be used to solve conflicts across aspect and base models and remove undesirable emergent properties during composition or during analysis of the composed model. This approach also provides directives to state the order of composition between aspects and the primary model.

Similar to the Theme/UML approach or the composition technic proposed by France et al., Klein et al. [14] use UML package templates to express reusable aspect models using class and sequence diagrams. In this approach, composing an aspect model with a base model involves binding the template parameters to base model elements, possibly with the help of pattern-matching technics. The resulting context-specific aspect model is then composed with the base model. In this approach, some aspects may depend on the structure or behavior provided by other aspects. Such a dependency is expressed at the model level by declaring an instantiation directive with the required aspect within the dependent aspect. This directive is exploited to correctly instantiate and compose the required aspect before it can be successfully composed with a base model.

In our previous work, we also contributed to this research by studying the construction of complex systems from aspectual templates. It appears that the construction process requires managing complex assemblies of aspectual templates with various forms of application. For instance, there are cases where multiple aspectual templates must be applied to a same target model while other

cases require to apply an aspectual template to a model resulting from another application. Even more, aspectual templates can be compose to each other in order to produce richer ones. This raises issues about ordering properties of applications, their independence or the equivalence of application chains. In [22], we addressed these issues in a consistent and systematic manner. This leads us to formalize properties which guarantee the correctness of composition chains and their alternative ordering capacities.

# 4 Enhancing UML Templates for AOM

## 4.1 Parameter as a Model

Aspectual templates have parameters that capture required elements from candidate models. Considering these parameters in isolation is compatible with the standard but is underspecified in order to capture the structure of the required model.

Figure 6 illustrates the problem. This figure shows a package template containing a set of elements for resource management functionalities related to a stock, some of them being exposed as parameters. As expected, all the parameters are elements included in the template content but one can also observe that they do not form a consistent model. Indeed, the *ref* property is exposed without its owning class whereas the latter is required to enable its mapping with a property contained in a base class. Similarly, the *in* association exposed as parameter is underspecified because one of its ends (the *Resource* class) is not declared as a parameter.

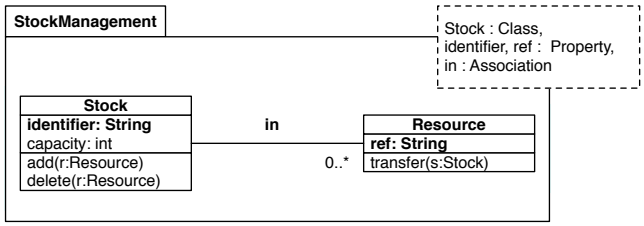


Figure 6: Inconsistent Set of Parameters

Figure 7 shows the preceding template where parameters were completed to form a full model required by the aspect. This required model specifies the structure expected from candidate models (two connected classes with string-based attributes in the example) to correctly inject the template functionality. Graphically, this specificity is rendered by replacing the parameters list by the corresponding (parameter) model[1].

[1]Parameter constituents are doted and bolded in the template core in order to highlight them. Others constitutents correspond to injected elements.

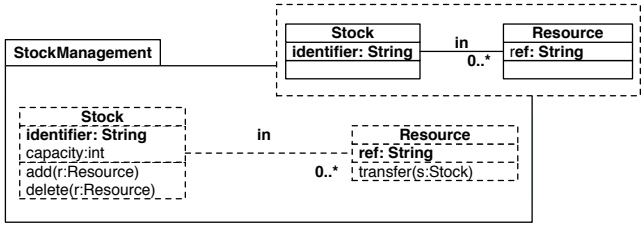


Figure 7: Resource Management Template

Figures 8 and 9 respectively illustrate aspectual templates for the injection of a functionality for searching resource in a stock and a counting one between two connected classes, one having a valuation method. They show that elements of the parameter model can be either properties, operations, associations and classes. The "resource allocation" aspectual template in Figure 10 is particularly interesting. It gives an example where classes of the parameter model are unconnected, the purpose being to install allocation management between classes representing Client-Product problems.

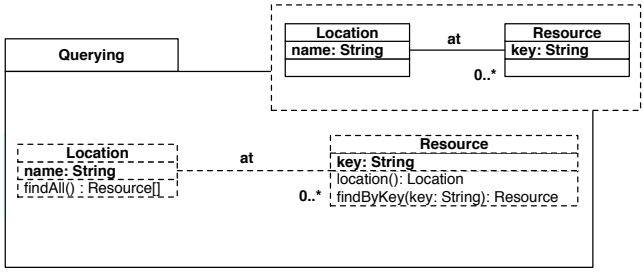


Figure 8: Querying Template

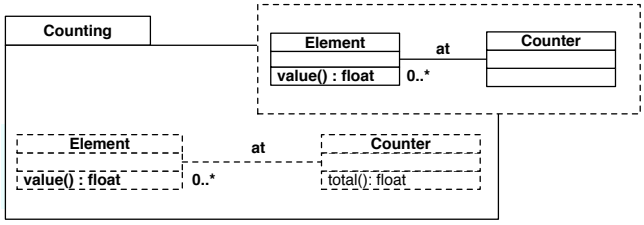


Figure 9: Counting Template

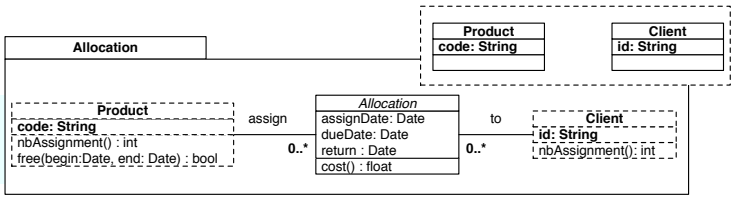


Figure 10: Resource Allocation Template

At this point, we get aspectual templates that are potentially reusable in many contexts thanks to parameterization. The next step is the application of such templates to extend a particular existing model. It is the goal of the next section to fulfill this need.

## 4.2 Applying Aspectual Templates

The application of aspectual templates is supported by an operator *apply*. It is based on the substitution of parameter model by conforming structure from the base model.

Let us take the example of a car hiring system. Figure 11 shows the base model of this system. This base describes the structure of the different domain classes used by the system (here *Car*, *Agency*, and *Client*). The resulting system must be able to search a specific car or a specific client and also to manage the different car allocations. To achieve this, we will use the aspectual templates presented before.

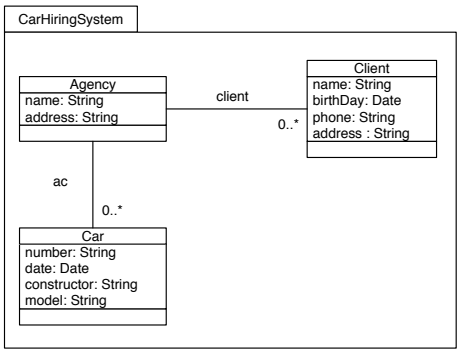


Figure 11: The Base System

Figure 12 shows how the *StockManagement* template is applied to the base system. Graphically, the *apply* operator is specified via a UML stereotyped dependency <<*apply*>> between the template and the target model. This dependency includes the substitution of formal parameters by elements of target model (actual arguments). To be valid, actual arguments of an "apply" relationship must form a model that structurally matches the parameter model of the aspectual template. This means the following: if a parameter of the template depends on another *e1* parameter (according to their modeling constraints), the

same applies to their corresponding bound elements; if two elements are connected by a *l1* link in the parameter model, their bound elements in the base model must be connected by a link bound to *l1*. This requirement is ensured by a set of constraints detailed in Section 5.2.2. In the example, parameters of the aspectual template (resp. *Stock*, *Resource*, *identifier*, *ref*, *in* ) are substituted by concrete elements of the base model (resp. *Agency*, *Car*, *name*, *number*, *ac*). We can verify that the structure formed by the parameter model is well-preserved by these actual arguments (emphasized by italic in the target on the figure).

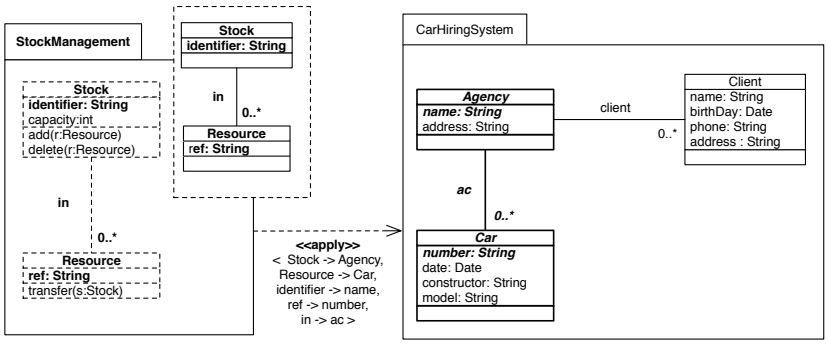


Figure 12: Applying Stock Management to Car Hiring System

A formulation of the resulting system is shown in Figure 13. We can see that aspectual template elements are injected into the base after substitutions were made. For instance, *add* and *delete* operations added to *Agency* have the type of their parameters changed from *Resource* to *Car*.

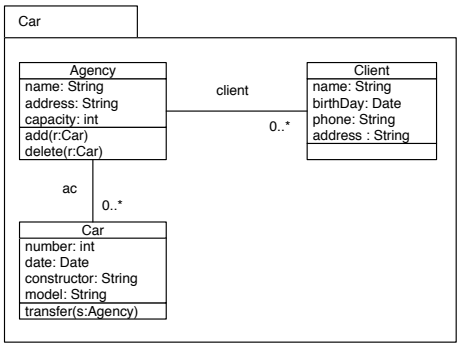


Figure 13: Base System with Car Management Functionality

These sketches show how to obtain an extended model from an existing one by the application of some aspectual template. To design richer aspectual templates from simpler ones and gain more reuse, composition of aspectual templates should be supported too. This facility is illustrated in Figure 14(a) where the *Querying* template applied to the *StockManagement* template allows to build a new aspectual template (see Figure 14(b)) providing stock management with a querying facility.

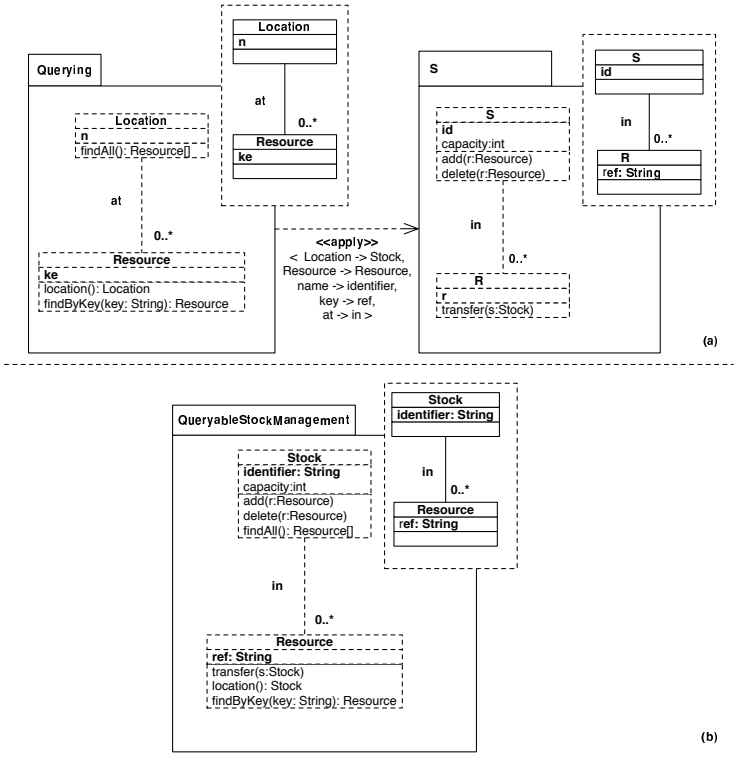


Figure 14: Template to Template Application

In the context of template to template application, it is possible to bind parameters of the source model to parameters of the target one. For example (Figure 14), the parameter *Location* is substituted by the parameter *Stock* in the resulting template. Note that, as a consequence, elements exposed as parameters in the target can be enriched. In the example, the *Stock* class has been enriched with the *findAll()* method.

The *apply* operator also supports different construction processes. As mentioned above, for the construction of complex systems from a set of aspectual templates, it should be possible to elaborate alternative application sequences and guaranty consistency properties of the resulting system. Related ordering and consistency properties are detailed and formalized in [22]. For example, Figure 15 shows the design of a possible template assembly for the design of the car hiring system on top of the model shown in Figure 11. The resulting system is presented in Figure 16. This example illustrates some needs. A first need is the ability to apply multiple aspectual templates to the same base. When such applications are independent, their evaluation order must not influence the result. It is the case of *Search* and *Allocation* applied to the base.

Another requirement is to express application chains. Such chains can be used to apply aspectual templates resulting from the composition of simpler ones. This is illustrated by the application of the *Querying* template to the *StockManagement* template, explained previously (Figure 14) which produces the *QueryableStockManagement* template. This new model is then used to add

stock management and querying functionalities on cars to our system. Note that an alternative construction chain would be to apply the *StockManagement* template to the base first, and then the *Querying* template to the resulting model. Both chains would produce the same result.

Another example is the application chain *Counting* to *Allocation* to *Base*. Alternative evaluation ordering of this chain, first *Counting* to *Allocation* then the resulting template to base or first *Allocation* to *Base* then *Counting* to the enriched *Base*, produces exactly the same result.

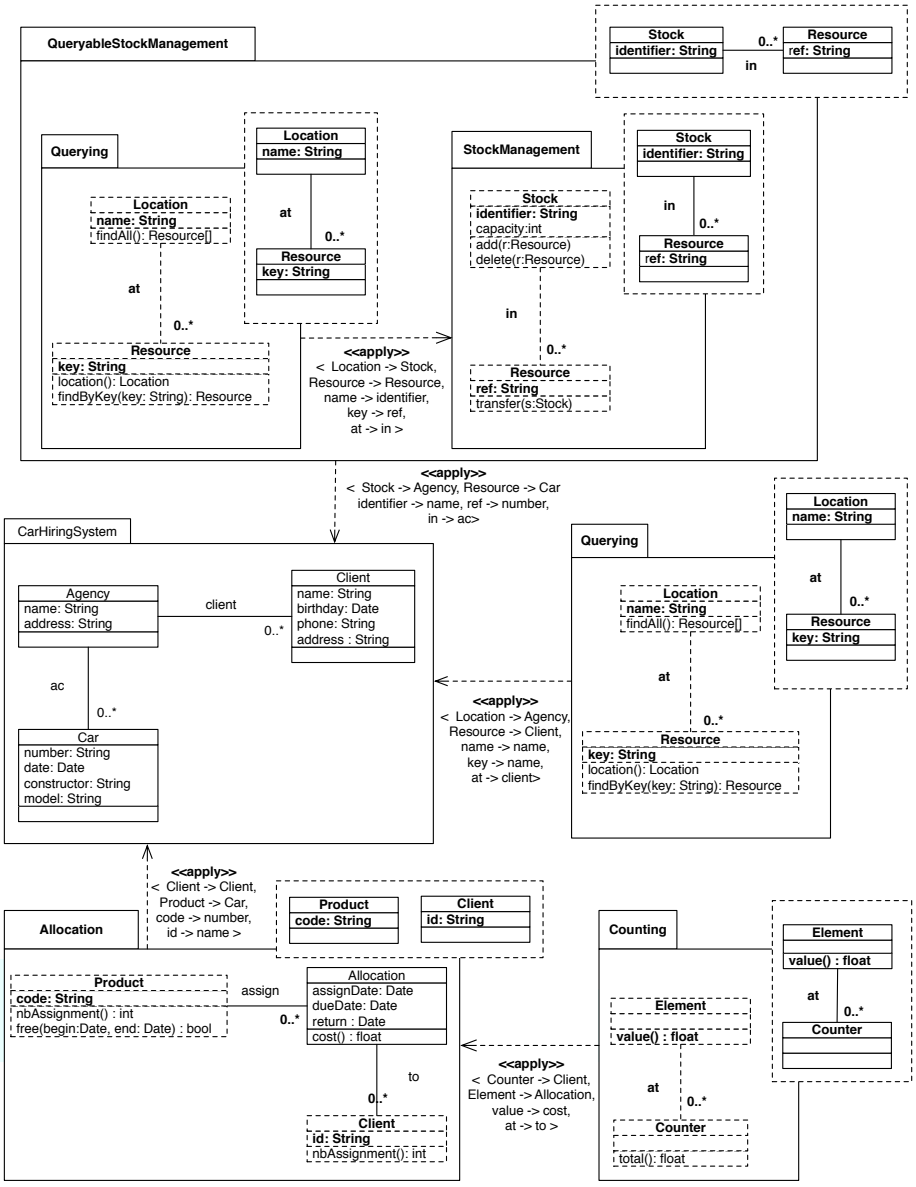


Figure 15: Example of Model Assembly

All the practices and properties presented previously allow to build new systems from prefabricated and validated aspectual templates with flexibility. This constitutes a good background for the furniture of aspectual template in UML. In the next section, we present how to obtain such templates in UML by specializing its metamodel to guarantee the correctness of previous practices.

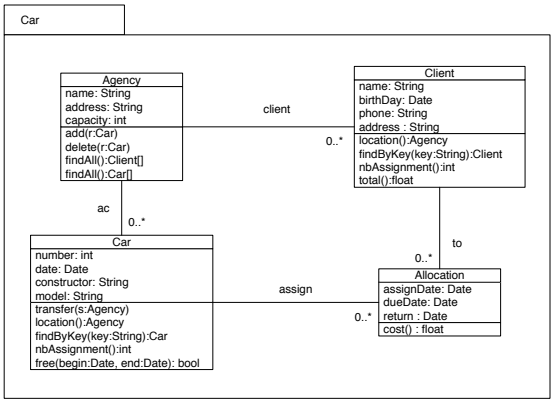


Figure 16: Resulting System from Model Assembly of Figure 15

# 5 From UML Templates to Aspectual Templates

In the following, we first present a subset of UML suitable for structural modeling (see Section 5.1). Then, we describe the extended metamodel with its constraints which allows to specify aspectual templates and their application (see Section 5.2).

## 5.1 Base Language Metamodel

For sake of simplicity, we will only consider the part of UML for structural modeling. Figure 17 shows the corresponding simplified metamodel which only includes classes, associations, properties and operations. A package contains classes and associations. A class contains a set of features (either operations or properties). An operation contains parameters and may return a value. The result and parameters of operations as well as properties are typed. Associations are restricted to binary ones. All these concepts exist in UML with their own OCL constraints. For reasons of space, we omit to recall them in the paper.

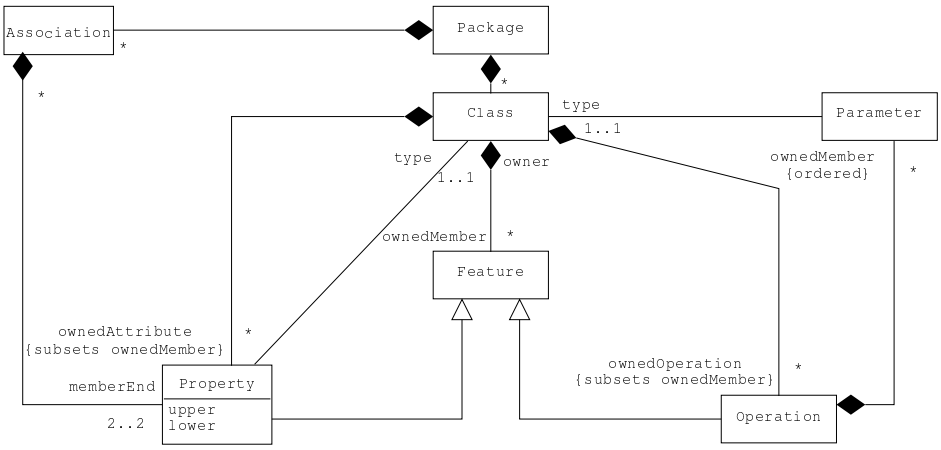


Figure 17: Base Language Metamodel for Structural Modeling

## 5.2 Metamodel Extension

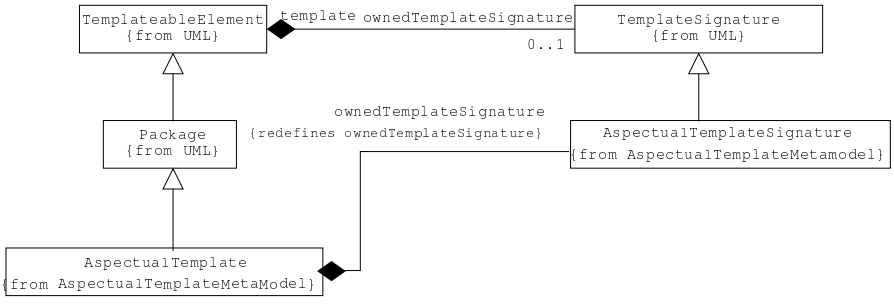


Figure 18: Aspectual Templates

Aspectual templates are defined by the subclass *AspectualTemplate* of *Package* so that they are *TemplateableElement*. As any *TemplateableElement* it owns a *TemplateSignature* more precisely typed by the *AspectualTemplateSignature* subclass. This subclass allows to constraint the parameters to form a valid

model due to aspectual template specificities.

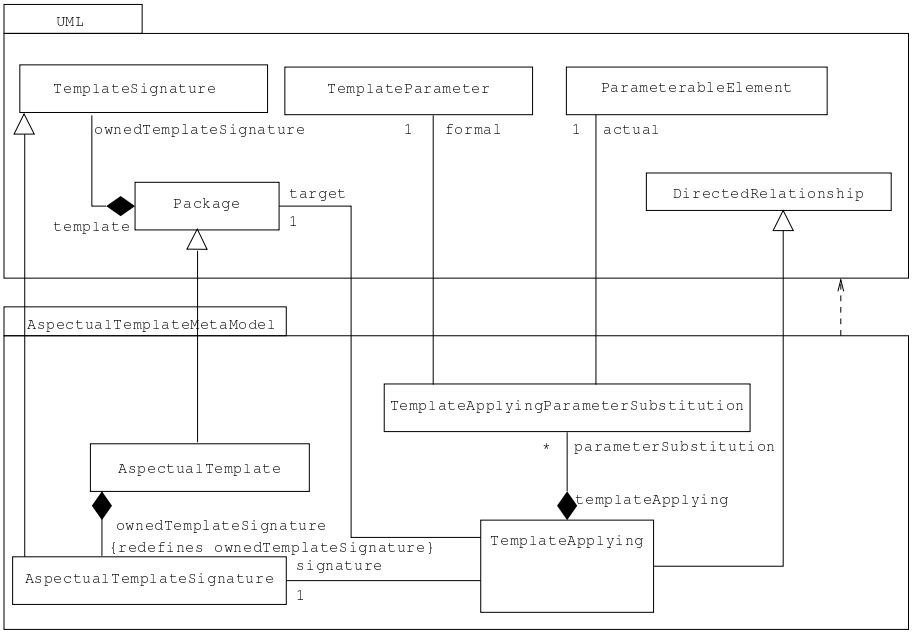


Figure 19: Aspectual Template Metamodel

Figure 19 shows the overall structure of the UML specialization. The apply operator is specified by the *TemplateApplying* metaclass. A *TemplateApplying* is a directed relationship between the signature of an aspectual template and some target package (*target* role). Thanks to subtyping, this target package can be an aspectual template, which enables application between aspectual templates. A

*TemplateApplying* owns a set of parameter substitutions (*TemplateApplyingParameterSubstitution*). A parameter substitution associates a formal parameter of the template signature to an actual parameterable element of the target package. Standard constraints existing in the metamodel for checking template signature are also guaranteed for aspectual templates by inheritance. Concerning other constraints related to parameter substitution and their conformance, we assume that they also exist for *TemplateAspectualSubstitution* and *TemplateApplying* to achieve similar standard checking.

Next, we state constraints which complete this structural metamodel. Subsection 5.2.1 presents constraints which check that template parameters of an aspectual template form a valid model while Subsection 5.2.2 is dedicated to the correctness of an aspectual template application. It focuses on the actual parameters side and their conformance with the model specified by formal parameters.

### 5.2.1 Checking template parameters

The specificity of an aspectual template compared to a general one comes from subtyping the *TemplateSignature* metaclass. The signature of an UML template considers the set of parameters as individual parameters while the aspectual template signature imposes that this set forms a full well-formed model.

That is the aim of the constraints formulated in this section. Figure 20 summarizes typical errors which will be explained along the presentation of the constraints.

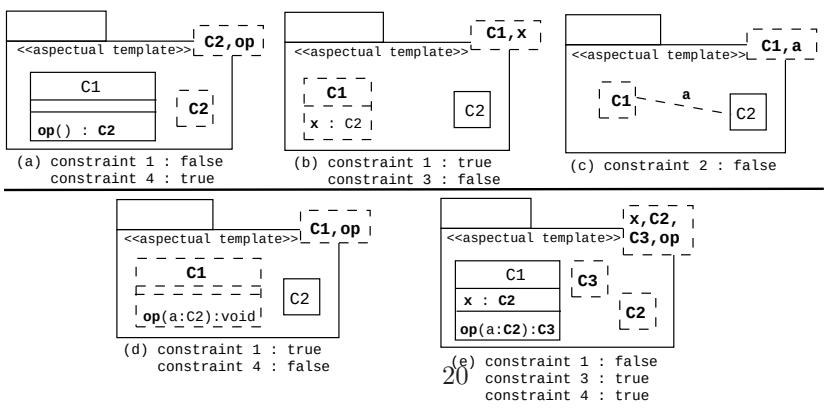




Figure 20: Examples of Invalid Aspectual Templates

Template parameters must respect the structure imposed by the base language metamodel (see Figure 17). Let us start with the composite association between *Class* and *Feature*. If a feature is exposed as a parameter, its class must also be a parameter. So the following constraint (number 1) prevents errors such as an operation without its owning class (Figures 20(a) and 20(e): *op* without *C1*) or an attribute without its owning class (Figure 20(e): *x* without *C1*)[2].

```
[1] Owning classes of parameter features must also be parameters

context AspectualTemplateSignature :
  self.ownedParameter->forAll(
    param : uml::TemplateParameter |
    let pe : uml::ParameterableElement = param.parameteredElement in
    pe.oclIsKindOf(uml::Feature) implies
    let ownerClass : uml::Class
       = pe.oclAsType(uml::Feature).owner.oclAsType(uml::Class) in
    ownerClass.isTemplateParameter()
  )
```

Constraint 2 deals with the consistency of associations. If an association is exposed as a parameter, its ending classes must also be parameters. Figure 20(c) does not respect that constraint because *C2* is not a parameter.

```
[2] Ending classes of a parameter association must also be parameters

context AspectualTemplateSignature :
  self.ownedParameter->forAll
  ( param : uml::TemplateParameter |
    param.parameteredElement.oclIsKindOf (uml::Association) implies
    let asso : uml::Association =
      param.parameteredElement.oclAsType(uml::Association) in
    asso.memberEnd->forAll( member | member.type.isTemplateParameter())
  )
```

The two following constraints (3-4) check the typing of features (properties or operations) that are parameters. In case of a property, constraint 3 checks that its type is also a parameter. Constraint 4 is similar in case of an operation : it checks that arguments and return value types are also parameters. For example (Figure 20):

- case (a): the type *C2* of operation *op* is present in the signature, constraint 4 is respected.
- case (b): the type *C2* of property *x* is not present in the template signature, constraint 3 is not respected
- case (d): the argument type *C2* of operation *op* is not present in the signature, constraint 4 is not respected

[2] We will use the standard UML query "isTemplateParameter" [27] (Templates Section):
```
"The query isTemplateParameter() determines if this parameterable element is
exposed as a formal template parameter.
ParameterableElement::isTemplateParameter() :  Boolean;
isTemplateParameter = templateParameter->notEmpty()"
```

- case (e): the types involved in the operation *op* are present in the template signature. The type *C2* of property *x* is present in the template signature. Both constraints 3 and 4 are respected.

```
[3] The type of a parameter property must also be a parameter

context AspectualTemplateSignature :
  self.ownedParameter->forAll(
    param : uml::TemplateParameter |
    let pe : uml::ParameterableElement = param.parameteredElement in
    pe.oclIsKindOf (uml::Property) implies
    pe.oclAsType(uml::Property).type.isTemplateParameter()
  )

[4] Types involved in a parameter operation must also be parameters

context AspectualTemplateSignature :
  self.ownedParameter->forAll(
    param : uml::TemplateParameter |
    let pe : uml::ParameterableElement = param.parameteredElement in
    pe.oclIsKindOf (uml::Operation) implies
    pe.oclAsType(uml::Operation).ownedParameter->forAll
    (p : uml::Parameter | p.type.isTemplateParameter())
)
```

### 5.2.2 Checking Template Applying

This section presents the set of constraints related to template applying. These constraints (5-8) check the conformance between the formal parameter model and the actual one. Figure 21 illustrates error cases which will be explained below.

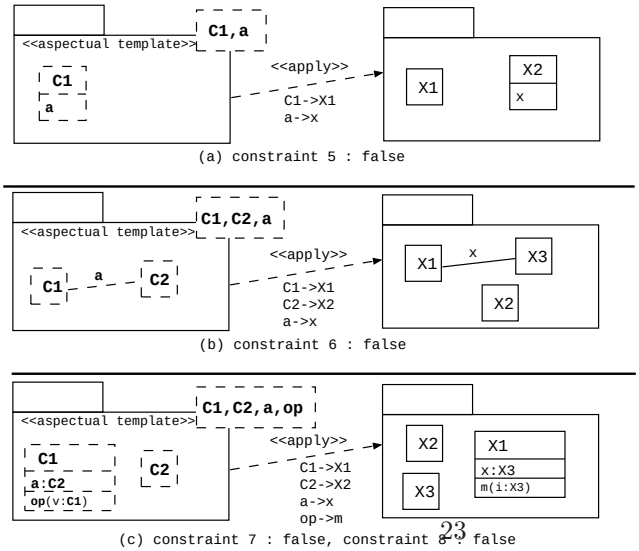




Figure 21: Examples of Conformance Checking for Actual Parameters

The two following constraints allow the checking of model structure. Constraint 5 focuses on the preservation of owned/owner relationships (for *Feature* parameters). Figure 21(a) illustrates that checking. The formal parameter $a$ is owned by the formal parameter *C1*. As a consequence, the actual parameter $x$ (associated to $a$) must be owned by the substituted class of *C1*: *X1*. As $x$ is owned by the *X2* class, the constraint 5 is violated.

```
[5] Owning relationships between formal parameters must be preserved in their
corresponding actual values

context TemplateApplying inv :
self.parameterSubstitution->forAll (s1, s2 |
  s1.formal.parameteredElement.owner = s2.formal.parameteredElement
  implies s1.actual.owner = s2.actual
)
```

Constraint 6 relates to the preservation of association structure. In Figure 21(b), member ends of the association $a$ are *C1* and *C2* classes. As a consequence, the member ends of the substituted association of $a$, that is $x$, should be the substituted classes of *C1* and *C2*, that are *X1* and *X2*. This preservation of the association structure is not respected in this example, the constraint 6 is violated.

```
[6] Member ends of a formal parameter association must be substituted by
 member ends of its actual value

context TemplateApplying inv :
  self.parameterSubstitution->forAll(s1, s2 |
  let asso:uml::ParameterableElement = s1.formal.parameteredElement in
  let cla:uml::ParameterableElement = s2.formal.parameteredElement in
  (asso.oclIsKindOf (uml::Association) and cla.oclIsKindOf (uml::Class)
     and asso.oclAsType(uml::Association).memberEnd->collect(type)
         ->includes(cla.oclAsType(uml::Class))
     implies
      s1.actual.oclAsType(uml::Association).memberEnd->collect(type)
       ->includes(s2.actual.oclAsType(uml::Class))
  )
)
```

The two following constraints focus on property substitution (constraint 7) and operation substitution (constraint 8). For a property parameter, its type must be substituted by the type of the substituted property. Figure 21(c) depicts an error. The type *C2* of the property $a$ is substituted by *X2*. Thus, the type of property $x$ which is substituted for $a$ should be *X2* and not *X3*.

```
[7] The substituted type of a formal parameter property must be
the type of its actual value

context TemplateApplying inv :
  self.parameterSubstitution->
  select(formal.parameteredElement.oclIsTypeOf(uml::Property))->forAll (tps |
  let prop:uml::Property =
     tps.formal.parameteredElement.oclAsType(uml::Property) in
  let substitutedProp: uml::Property = tps.actual.oclAsType(uml::Property) in
```

```
  if prop.type.isTemplateParameter() then
   self.parameterSubstitution->exists(
     formal.parameteredElement=prop.type and actual=substitutedProp.type)
  else
   prop.type=substitutedProp.type
  endif
)
```

Constraint 8 deals with operations: for each substitution, it is necessary to check if the signature of the formal parameter is compatible with the signature of the related actual argument. Violation of this constraint is shown in Figure 21(c): the first argument type of *op* is *C1*; as *C1* is substituted by *X1*, the first argument type of *m* should be *X1* and not *X3*.

```
[8] The substituted types of operation parameters must be
substituted by the corresponding types of the actual operation

context TemplateApplying inv :
  self.parameterSubstitution->select(formal.parameteredElement.
  oclIsTypeOf(uml::Operation))->forAll (tps |
  let formalOp : uml::Operation =
    tps.formal.parameteredElement.oclAsType(uml::Operation) in
  let actualOp : uml::Operation =
    tps.actual.oclAsType(uml::Operation)  in
  formalOp.ownedParameter->size()=actualOp.ownedParameter->size() and
  Sequence{1..formalOp.ownedParameter->size()}->forAll ( index  |
    let memberFormalOp : uml::Parameter =
    formalOp.ownedParameter->asOrderedSet()
    ->at(index).oclAsType(uml::Parameter) in
    let memberActualOp : uml::Parameter =
    actualOp.ownedParameter->asOrderedSet()
    ->at(index).oclAsType(uml::Parameter) in
    if memberFormalOp.type.isTemplateParameter() then
      self.parameterSubstitution->exists(
        formal.parameteredElement=memberFormalOp.type and
        actual=memberActualOp.type )
    else
      memberFormalOp.type=memberActualOp.type
    endif
  )
)
```

# 6 Applying Algorithm

This section presents an algorithm that applies an aspectual template to a target package or to another aspectual template.

The algorithm takes one input parameter which is a *TemplateApplying* specifying the source aspectual template, the target package and a set of substitutions. The effect of the algorithm is to modify the target package with additional elements of the aspectual template after parameter substitution.

We make the assumption that the *TemplateApplying* parameter is valid with respect to the set of rules presented in the previous sections. So, the checking of *TemplateApplying* is not included in the algorithm.

For describing the algorithm, we also assume the following:

- The *ParameterableElement* metaclass owns an *isTemplateParameter(): boolean* operation which returns true if the object is exposed as a template parameter, false otherwise.
- The *Package* metaclass owns an *addParameter(pe:ParameterableElement)* operation which adds *pe* contained in the package as template parameter. This operation creates and attaches a new *TemplateParameter* referencing the provided element to the *TemplateSignature* of the package.
- The *TemplateApplying* metaclass owns an *isSubstituted(pe:ParameterableElement) : boolean* operation which returns true if *pe* is a formal template parameter bound to an actual element in the target package, false otherwise.
- The *TemplateApplying* metaclass owns an *getActual(pe:ParameterableElement) : ParameterableElement* operation which returns the actual argument corresponding to the *pe* formal parameter.
- Map is a conventional class for mapping keys to values.
- The *clone()* operation is available on all model elements of the base language metamodel. It creates an element of the same metatype and copies its meta-attributes.

The algorithm is based on three steps which are the following:

**Insertion of unbound template classes into the target package:** In this step, iteration is made over the set of classes contained in the aspectual template. If a class is not a parameter or is an unsubstituted parameter, a clone without features is created and added to the target package. In case of an unsubstituted template class, a corresponding parameter is added to the target package signature. Finally, in preparation to the next step, mapping is made between each template class and its corresponding target class which is either one specified by the template applying or one created previously.

**Extension of target classes with features from corresponding template classes:** This step consists in extending all the target classes mapped to template classes with clones of their properties and operations which are not parameters or are unsubstituted ones. Starting from the mapping set up during the first step, the algorithm iterates over this mapping to determine each target class. For each added properties and operations, the template class(es) referenced by their type(s) is replaced by the corresponding target class.

**Insertion of unbound template associations into the target package:** In this last step, the algorithm inserts aspectual template associations which are not parameters or are unsubstituted parameters into the target package. New associations have their owned ends adapted to take into account substituted or cloned template classes.

**Algorithm 1:** execute($apply : TemplateApplying$)

```
Data: template : AspectualTemplate, base : Package, map : Map
begin
    map ⟵ {};
    template ⟵ apply.signature.template;
    base ⟵ apply.target;
    // Step 1:  Insertion of unbound template classes
    for tcl ∈ template.classes do
        if ¬tcl.isTemplateParameter() then
            bcl ⟵ tcl.clone();
            base.classes ⟵ base.classes + bcl;
        else
            bcl ⟵ apply.getActual(tcl);
        map ⟵ map+ < tcl, bcl >;
    // Step 2:  Extension of base classes from template classes
    for tcl ∈ map.keys() do
        bcl ⟵ map.get(tcl);
        // Properties
        for tprop ∈ tcl.ownedAttribute do
            if tprop.isTemplateParameter() then
                bprop ⟵ tprop.clone();
                bprop.type ⟵ map.get(tprop.type);
                bcl.ownedAttribute ⟵ bcl.ownedAttribute + bprop;
        // Operations
        for top ∈ tcl.ownedOperation do
            if ¬top.isTemplateParameter() then
                bop ⟵ top.clone();
                bop.type ⟵ map.get(bop.type);
                bcl.ownedOperation ⟵ bcl.ownedOperation + bop;
                for tparam ∈ top.ownedParameter do
                    bparam ⟵ tparam.clone();
                    bparam.type ⟵ map.get(tparam.type);
                    bop.ownedParameter ⟵ bop.ownedParameter + bparam;
        // Step 3:  Insertion of template associations
        for tassoc ∈ tcl.associations do
            if ¬tassoc.isTemplateParameter() then
                bassoc ⟵ tassoc.clone();
                base.associations ⟵ base.associations + bassoc;
                for tend ∈ tassoc.memberdEnd do
                    bend ⟵ tend.clone();
                    bend.type ⟵ map.get(tend.type);
                    bassoc.memberEnd ⟵ bassoc.memberEnd + bend;
```

The algorithm produces a resulting model which is consistent with the standard bind relationship. As explained in Section 2.1 the binding relationship allows to set the existing link between a model and some templates used for its construction. It implies that the contents of the bound model have all the contents of the template with any element exposed as a formal parameter substituted by the actual element specified in the binding.

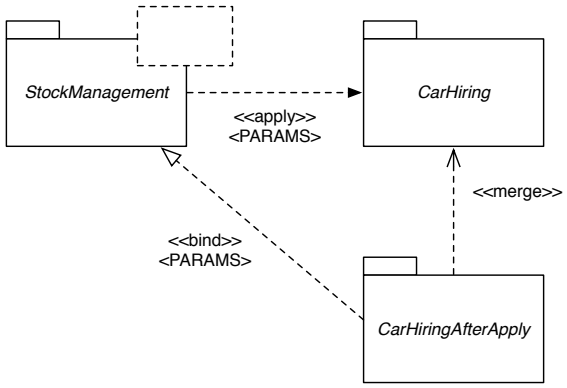


Figure 22: Relationships of Resulting Model

Figure 22 illustrates this property for the *StockManagement* template application considered previously. In this figure, *CarHiringAfterApply* is the model resulting from this template application to the *CarHiring* model[3]. As a result of the algorithm, the resulting model respects the constraints of a bind relationship to the source template with the same set of substitutions than the apply relationship. It is therefore possible to capture this compliance by setting up such a bind relationship between this (bound) resulting model and the initial template. In addition to the bind relationship, the figure also shows the existence of a standard merge relationship between the resulting model and the base model. This relationship figures that the apply operation preserves the content of the base model in the resulting one.

Regarding the bind relationship from resulting model to source template, it is possible to complete the previous algorithm so that this relationship is set up automatically. This can be useful to keep trace of template applications used to obtain intermediate and final models. Another interest of setting this relationship is to enable automatic removal or replacement of added elements using an "unweaving" process similar to [19].

# 7 Integration to Case Tools

Based on the present work, technologies have been implemented and made available in the Eclipse Modeling Framework environment. This implementation relies on the following plugins:

**EMF plugin:** [13] (based on Essential MOF) is a Java modeling framework that includes a metamodel for describing models. It provides tools and

[3]Here, this resulting model is made distinct from the base model for clarity reasons.

runtime support to produce a set of Java classes for these models and a basic editor.

**UML plugin:** EMF facilities have been used by its designers to represent the UML metamodel and generate UML plugins[4] that support representation and editing of UML models.

**OCL plugin:** this is an implementation of the Object Constraint Language (OCL) from OMG standard for EMF-based models. It provides Java APIs for parsing and evaluating OCL constraints and queries on Ecore or UML models.

On top of these plugins, new plugins dedicated to aspectual templates are offered[5]:

- an extended UML plugin that supports representation of extended UML models with the new concepts: aspectual template and template applying.
- an aspectual template engine that checks all OCL constraints specified in this paper and implements the apply algorithm.
- an editor plugin generated by EMF facilities (see an example on the left side of the capture screen in Figure 23). This editor allows to specify new UML aspectual templates, import existing aspectual templates and base models from a repository and specify applications. The edited models and applications can be validated at any time using the editor.
- a basic UML grapher (see an example on the right side of the capture screen in Figure 23) showing assembly of aspectual templates graphically.

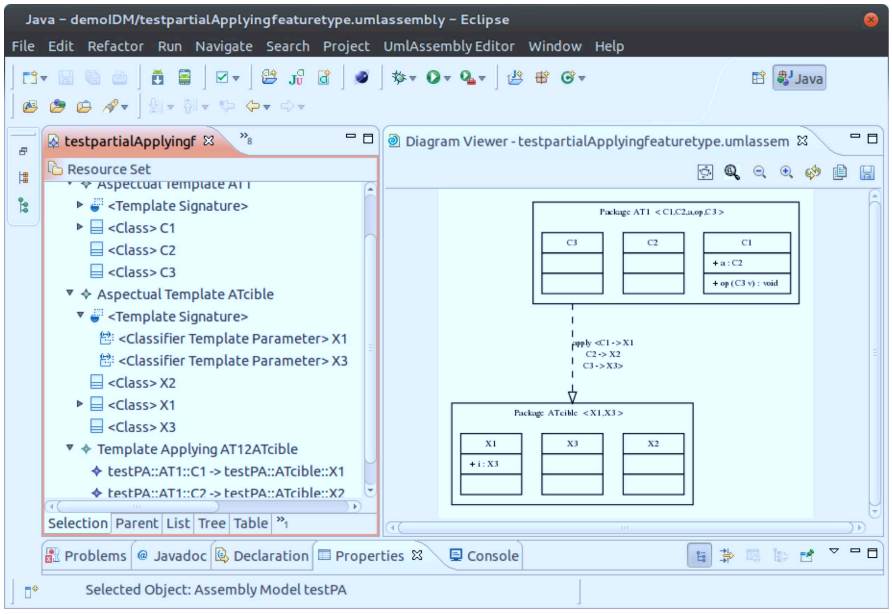


Figure 23: Using Aspectual Templates in Eclipse

[4] see UML project at http://www.eclipse.org
[5] Available at the URL: `http://www.lifl.fr/GOAL/cocoa/pmwiki.php?n=Main.CocoaModeler`

Thanks to plugin-based facilities promoted by Eclipse, these plugins can be integrated into compliant case tools.

# 8 Conclusion

UML Templates allow to define models whose some of their constituents are parameters. This construct is general and is used in many ways, such as generic classes representation, Design Pattern modeling, View or Aspect Oriented Modeling (AOM). In this paper, we concentrated on this last usage and the characteristics of so called Aspectual Templates. Rationale of these templates is their capacity to be applied to other models (being parameterized or not) in order to enrich them, as far as they conform to some required parameter model.

As a consequence aspectual templates need to specialize the general template notion in the following way:

- Parameters are used to specify a required model, so that their flat structure (as a set of typed model elements) in the standard must be constrained to conform to well-formedness rules.
- The process of their application to some model must deal with specific constraints, compared to the general template binding mechanism.
- In particular, when template to template application, it is necessary to ensure that the resulting aspectual template parameters remain a consistent model.

In order to capture these specificities, we specialized the UML Template metamodel and state the needed OCL constraints relative to this specific interpretation. As a major result, this specialization allows aspectual template composition in a homogeneous and consistent way: construction of richer aspectual templates from the (recursive) composition of others ones, their capitalization, and finally their usage within modeling assemblies in order to obtain a system. Finally, at a much more operational level, we present an algorithm which shows how to produce a model from the application of an aspectual template to a target model. This algorithm also works for aspectual template to template application. All these results were implemented and made available in the EMF (Eclipse Modeling Framework) technology.

The paper concentrated on the core elements of the general template notion specified in UML. However, the standard includes other ingredients for template definition and their binding. We may cite the notion of default value for unbound parameters or the capacity to define a new template by extending an existing one thanks to the concept of *RedefinableTemplateSignature*. How these additional ingredients relate with the aspectual interpretation of templates and may complement this usage is a valuable issue.

For sake of simplicity, this paper only considers basic structural object models. Though, the presented guidelines may extend to other modeling constructs, such as cardinalities of associations, meta-attributes of model elements or inheritance links. This will lead to investigate a richer conformance relationship with specific substitution capabilities in templates application. In our previous work on contextualization of object models [5], we already studied similar issues and provided solutions specifically for cardinalities and inheritance links. Other

works on the merging of class diagrams like [7, 11] would also be a helpful basis to study this issue. More generally, the presented guidelines may also extend to other kind of diagrams. For example, the usage of Aspectual Templates to sequence diagrams was studied in [14, 26]. In the future, work must be done to systematize the approach to other UML templateable elements.

Finally, we showed how it is possible to construct systems as well as "off-the-shelf" rich aspectual templates from the application of multiple ones. This leads to the notion of "model assembly" whose attended properties have been already stated in ([22]). Beside aspectual template application, such assemblies may also include other reusing relationships such as standard "merge" and "extends". Work still remains to be done in order to study how these relationships systematically compose to the help of the consistent specification of model assemblies.

# References


[1] MDA. Home Page. http://www.omg.org/mda.

[2] J. Bigot and Ch. Pérez. Increasing Reuse in Component Models through Genericity. In *Proceedings of the 11th International Conference on Software Reuse, ICSR '09*, volume 5791 of *LNCS*, pages 21–30. Springer, 2009.

[3] O. Caron, B. Carré, A. Muller, and G. Vanwormhoudt. An OCL Formulation of UML 2 Template Binding. In *Proceedings of 7th International Conference on The Unified Modeling Language. Model Languages and Applications (UML 2004)*, volume 3273 of *LNCS*, pages 27–40. Springer, October 2004.

[4] O. Caron, B. Carré, A. Muller, and G. Vanwormhoudt. A Coding Framework for Functional Adaptation of Coarse-Grained Components in Extensible EJB Servers. In *47th International Conference Objects, Models, Components, Patterns (Tools'09)*, number 33 in LNBIP, pages 215–230. Springer-Verlag, 2009.

[5] Olivier Caron, Bernard Carré, and Laurent Debrauwer. Contextualization of OODB Schemas in CROME. In *Proceeding of the 11th International Conference on Database and Expert Systems Applications DEXA 2000,*, volume 1873 of *LNCS*, pages 135–149. Springer-V, 2000.

[6] T. Clark, A. Evans, and K. Stuart. Aspect-oriented Metamodelling. *The Computer Journal*, 46(5):566–577, 2003.

[7] S. Clarke. Extending standard UML with Model Composition Semantics. In *Science of Computer Programming*, volume 44, pages 71–100. Elsevier Science, 2002.

[8] S. Clarke and R. J. Walker. Generic Aspect-Oriented Design with Theme/UML. In *Aspect-oriented software development*, pages 425–458. Addison-Wesley Professional, 2004.

[9] Siobhán Clarke and Robert J. Walker. Composition patterns: An approach to designing reusable aspects. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001,*, pages 5–14. IEEE Computer Society, 2001.

[10] A. Cuccuru, A. Radermacher, S. Gérard, and F. Terrier. Constraining Type Parameters of UML 2 Templates with Substitutable Classifiers. In *Proceedings of 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*, volume 5795 of *LNCS*. Springer, 2009.

[11] J. Dingel, Z. Diskin, and A. Zito. Understanding and improving UML package merge. *Software and System Modeling*, 7(4):443–467, 2008.

[12] D. D'Souza and A. Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1999.

[13] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework, 2nd Edition*. Addison Wesley, 2009.

[14] Jorg J. Kienzle, W. Al Abed, F. Fleurey, J-M. Jézéquel, and J. Klein. Aspect-Oriented Design with Reusable Aspect Models. In S. Katz, M. Mezini, and J. Kienzle, editors, *Transactions on Aspect-Oriented Software Development VII - A Common Case Study for Aspect-Oriented Modeling*, volume 6210 of *LNCS*, pages 272–320. Springer, 2010.

[15] J.M. Jézéquel. Model driven design and aspect weaving. *Software and System Modeling*, 7(2):209–218, 2008.

[16] S. Kent. Model Driven Engineering. In *Proceedings of the 3rd International Conference on Integrated Formal Methods*, volume 2335 of *LNCS*, pages 286–298. Springer, May 2002.

[17] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer, 1997.

[18] J. Klein, L. Hélouët, and J. M. Jézéquel. Semantic-based Weaving of Scenarios. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, pages 27–38. ACM Press New York, NY, USA, 2006.

[19] J. Klein, J. Kienzle, B. Morin, and J.M. Jézéquel. Aspect Model Unweaving. In *Proceedings of 12th International Conference on Model Driven Engineering Languages and Systems, MODELS'09*, volume 5795 of *LNCS*, pages 514–530. Springer, 2009.

[20] Ph. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J. M. Jézéquel. Introducing Variability into Aspect-Oriented Modeling Approaches. In *Proceedings of 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, LNCS, pages 498–513. Springer, October 2007.

[21] J. Lara and E. Guerra. From Types to Type Requirements: Genericity for Model-driven Engineering. *Software and System Modeling*, pages 1–22, 2011.

[22] A. Muller, O. Caron, B. Carré, and G. Vanwormhoudt. On Some Properties of Parameterized Model Application. In *Proceedings of 1st European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05)*, volume 3748 of *LNCS*, pages 130–144. Springer, November 2005.

[23] Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J. M. Bieman, N. McEachen, E. Song, and G. Georg. Directives for Composing Aspect-Oriented Design Class Models. In *Transaction on Aspect-Oriented Software Development I*, volume 3880, pages 75–105. Springer, 2006.

[24] Greg Straw, Geri Georg, Eunjee Song, Sudipto Ghosh, Robert France, and JamesM. Bieman. Model composition directives. In *UML 2004 - The Unified Modeling Language. Modelling Languages and Applications*, volume 3273 of *LNCS*, pages 84–97. Springer, 2004.

[25] G. Sunyé, A. Le Guennec, and J-M. Jézéquel. Design Patterns Application in UML. In E. Bertino, editor, *Proceedings of ECOOP 2000*, volume 1850 of *LNCS*, pages 44–62. Springer, 2000.

[26] S. Thiello. Model Templates for Roles Interaction, Master thesis. Technical report, University of Lille, 2010.

[27] UML 2.4.1 Superstructure Specification, 2011. http://www.omg.org/spec/UML/2.4.1/.

[28] Auxiliary Constructs Templates, chapter 17. UML 2.4.1 Superstructure Specification, 2011.

[29] J. Whittle, K. Praveen, A. Jayaraman, M. Elkhodary, A. Moreira, and J. Araújo. MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. In *Transactions on Aspect-Oriented Software Development VI, Special Issue on Aspects and Model-Driven Engineering*, volume 5560 of *LNCS*, pages 191–237. Springer, 2009.

[30] M. Wimmer, A. Schauerhuber, G. Kappel, W. Retschitzegger, W. Schwinger, and E. Kapsammer. A Survey on UML-based Aspect-oriented Design Modeling. In *ACM Computing Surveys*, volume 43, pages 28:1–28:33. ACM, October 2011.