Constraining Type Parameters of UML 2 Templates with Substitutable Classifiers

Arnaud Cuccuru, Ansgar Radermacher, Sébastien Gérard, and François Terrier

CEA LIST, Boîte 94, Gif-sur-Yvette, F-91191, France
`first_name.last_name@cea.fr`

Abstract. Generic programming is a field of computer science which consists in defining abstract and reusable representations of efficient data structures and algorithms. In popular imperative languages, it is usually supported by a template-like notation, where generic elements are represented by templates exposing formal parameters. Defining such generic artifacts may require defining constraints on the actual types that can be provided in a particular substitution. UML 2 templates support two mechanisms for expressing such constraints. Unfortunately, the UML specification provides very few details on their usage. The purpose of our article is to provide such details with regard to one of these constraining mechanisms (namely, "substitutable constraining classifiers") as well as modeling patterns inspired by practices from generic programming.

1 Introduction

UML 2 templates [1] (chapter 17.5) are inspired by template-like mechanisms of popular programming languages such as C++ or Java. They provide support for the three fundamental notions of template-based design: templates (i.e., metaclass *TemplateableElement*), formal parameters (i.e., *TemplateSignature*, *TemplateParameter* and *ParameterableElement*) and bindings (i.e., *TemplateBinding* and *TemplateParameterSubstitution*). A template is a kind of abstract element whose definition is parameterized by other elements. Elements that are exposed as parameters of a template definition are called its formal parameters. A concrete element (usually called bound element) can then be instantiated by binding a template, i.e., specifying a substitution for each of its formal parameters.

Defining generic structures or algorithms typically requires making assumptions on the types exposed as formal parameters of a template, by defining constraints on the actual types that can be provided in a particular substitution. For this purpose, UML 2 templates provide a refinement of the metaclass TemplateParameter called ClassifierTemplateParameter (used for the exposure of classifiers) that can be associated with a set of constraining classifiers. An additional boolean property of ClassifierTemplateParameter, called *allowSubstitutable*, enables two potential interpretations of these constraining classifiers. In the case where it is false, the interpretation is object-oriented. It indeed implies that a valid actual type must have a direct or indirect generalization relationship with each of the constraining classifiers. This approach will not be discussed in this article (see [2] and [3] for details). In the case where it is true, the set of constraining classifiers simply specifies a kind of contract. Any actual classifier satisfying this contract is a valid substitution for the formal parameter, i.e., it is substitutable. Unfortunately, the UML specification provides very few details on the usage of this constraining mechanism. It is the purpose of this article to provide such details and guidelines.

We propose a simple pattern where contracts (i.e., substitutable constraining classifiers) are defined as template classifiers and used as namespaces containing any element useful for the definition of a generic behavior or structure. We show that it is then possible to explicitly specify how an actual classifier realizes the contract by combining Realization and TemplateBinding relationships. The details of our proposal are described in section 2. In section 3, we discuss related works from the generic programming community. Section 4 then concludes this article and sets guidelines for future research.

2 Substitutable Constraining Classifiers in Action

Using substitutable constraining classifiers is a very loose form of specifying type compatibility rules. The semantic relationship between an actual classifier and the constraining classifiers is in fact equivalent to a Realization relationship. In UML, a Realization relationship is a kind of assertion that a given classifier realizes another one. It needs to be augmented with information explicitly specifying how the realization of the specification is actually done. In order to illustrate how this information can be made more explicit, let us consider a generic activity called *accum*, that computes the sum of the elements contained in a collection.

This activity is generic from two standpoints: 1.) the type of the elements to be accumulated (which must support the operator "+") and 2.) the type of the collection containing the elements (which is required to be "iterable forward"). In the following subsections, we illustrate our proposal by focusing on the second standpoint. We first describe a template class *Iterator* with a formal type parameter C (i.e., the type of the collection) and its associated substitutable constraining classifier *IterableForward*. Then, we show how the substitutability of a given classifier (Vector in our example) with respect to the contract represented by IterableForward can be explicitly specified.

2.1 Specifying the Contract

Descriptions of the class Iterator and the constraining classifier IterableForward (which constrains parameter C) are shown in Fig. 1. On the left-hand side of the figure (i.e., template signature of Iterator), the standard keyword "contract" associated with the formal parameter C renders the fact that *allowSubstitutable* is true.

Our proposal consists in using the members of IterableForward (shown in the *nestedClassifier* and *ownedBehavior* compartment of the class) to capture the requirements that must be fulfilled by a given classifier to be considered as a valid substitution for parameter C. Literally, a classifier satisfies the contract

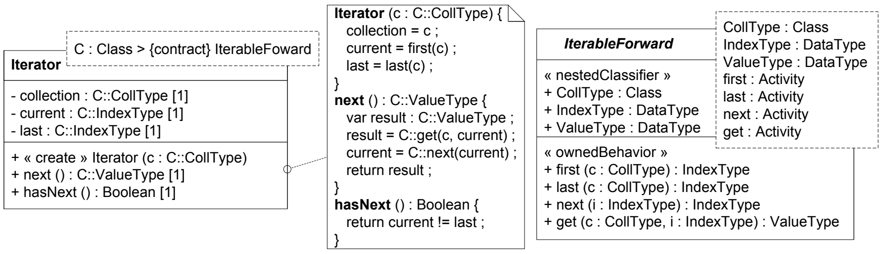

Iterator
C : Class > {contract} IterableFoward
- collection : C::CollType [1]
- current : C::IndexType [1]
- last : C::IndexType [1]
+ « create » Iterator (c : C::CollType)
+ next () : C::ValueType [1]
+ hasNext () : Boolean [1]
Iterator (c : C::CollType) {
collection = c ;
current = first(c) ;
last = last(c) ;
}
next () : C::ValueType {
var result : C::ValueType ;
result = C::get(c, current) ;
current = C::next(current) ;
return result ;
}
hasNext () : Boolean {
return current != last ;
}
IterableForward
« nestedClassifier »
+ CollType : Class
+ IndexType : DataType
+ ValueType : DataType
« ownedBehavior »
+ first (c : CollType) : IndexType
+ last (c : CollType) : IndexType
+ next (i : IndexType) : IndexType
+ get (c : CollType, i : IndexType) : ValueType
CollType : Class
IndexType : DataType
ValueType : DataType
first : Activity
last : Activity
next : Activity
get : Activity

Fig. 1. Specification of Iterator and its constraining classifier IterableForward

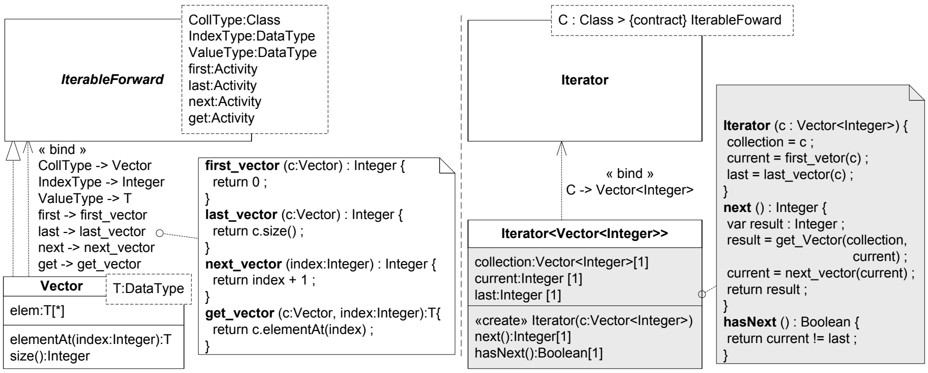
represented by IterableForward if, from this classifier: 1.) It is possible to derive the type of the collection (*CollType*), the type of the element used as an index for accessing elements contained in the collection (*IndexType*) and the type of the elements it contains (*ValueType*) (In the C++ terminology, these types would be called "traits" [4] of IterableForward.) and 2.) It is possible to derive the activities that compute the index of the first element (*first*), the index of the last element (*last*), the index following a given index (*next*) and the value of the element at a given index (*get*).

Provided the definition of IterableForward, it is then possible to generically specify the class Iterator with respect to its formal parameter *C*. In the left-hand side of Fig. 1, we can see that the model of the class Iterator relies on explicit references to traits and activities of *C* (as illustrated by the usage of fully qualified names for each element) for typing its properties and implementing its operations. For a given actual classifier *S* provided as a substitution for parameter *C* (in a binding of the template Iterator), any reference to a member *M* of *C* will be substituted with an element derived from *S* (provided that *S* realizes IterableForward). Defining how a classifier *S* actually realizes IterableForward is the purpose of the following section.

2.2 Specifying the Substitutability

Let us show how a class *Vector* can be made a valid substitution for the parameter *C* of template Iterator. Determining how Vector realizes IterableForward (and each of its members) can be achieved by specifying a template binding relationship between Vector and IterableForward (along with a substitution for each of these members). This solution is illustrated in the left-hand side of Fig. 2.

We can see that the substitutions associated with the binding relationships state that: CollType will be played by Vector itself, IndexType by Integer and ValueType by T (which is itself exposed as a parameter of Vector). Similarly, a substitution is provided for each of the activities of Iterator. The actual activities (*first_ vector*, *last_ vector*, *next_ vector* and *get_ vector*) are signature compatible with those of IterableForward, except that the type of their parameters has been substituted with the actual types provided as substitutions for traits of IterableForward. Additionally, they encapsulate an implementation suited to



```
first_vector (c:Vector) : Integer {
  return 0 ;
}
last_vector (c:Vector) : Integer {
  return c.size() ;
}
next_vector (index:Integer) : Integer {
  return index + 1 ;
}
get_vector (c:Vector, index:Integer):T{
  return c.elementAt(index) ;
}
```

```
Iterator (c : Vector<Integer>) {
 collection = c ;
 current = first_vetor(c) ;
 last = last_vector(c) ;
}
next () : Integer {
 var result : Integer ;
 result = get_Vector(collection,
                         current) ;
 current = next_vector(current) ;
 return result ;
}
hasNext () : Boolean {
 return current != last ;
}
```

Fig. 2. Realization of the contract IterableForward and instanciation of Iterator

the specificities of Vector (e.g., activity *get_ vector* is described using operation *elementAt* of Vector).

In UML 2, defining a binding relationship between a bound element and a template normally implies a two steps generative process. It consists in replicating the template and then substituting any reference to formal parameters with actual elements provided in the binding. In the case of Vector, we clearly do not want this generative process to apply. The presence of the Realization relationship (appearing on the left side of the binding relationship in Fig. 2) captures this intention. The Realization relationship is just used as an assertion that Vector realizes IterableForward and the TemplateBinding relationship explicitly specifies how this realization is actually done without requiring any replication. This reasonable interpretation is the only extension we introduce in standard UML 2 templates. Regarding the template Iterator and its formal parameter C, the binding relationship between Vector and IterableForward provides sufficient information for determining the substitution to be operated if an instance of the template Vector (e.g., Vector<Integer>) is provided as a substitution for the formal parameter C. The result of the substitution is illustrated in the right-hand side of Fig. 2.

Applying the same principles, the activity *accum* can be specified with two formal parameters: *C:Class > {contract}IterableForward* and *T:DataType = C::ValueType > Addable<T>*. Just like for Iterator, parameter *C* represents a collection type. It is associated with IterableForward as a substitutable constraining classifier. The additional formal parameter *T* represents the elements contained in the collection. Its constraining classifier Addable<T> is not substitutable (it represents an abstract class owning an operation *add(T,T):T*). Note that *T* has a default substitution (in the UML2 metamodel, it corresponds to the property *default* of metaclass TemplateParameter). It means that for a given binding of activity *accum*, the only required substitution concerns parameter *C*. The substitution for parameter *T* will be automatically inferred from *C* (i.e., C::ValueType). The following section sets links between these proposals and the fundamental principles put into action in the field of generic programming.

3 Related Works

Concepts and *models* are the most fundamental notions of generic programming [5] [6]. We have not used these words until now in order to avoid confusion with the homonym notions from MDE. A *concept* defines a set of requirements on a type (like associated types and functions), and a type *models* a concept if it satisfies its requirements. A *concept* can be associated with a formal type parameter to constrain the possible substitutions. A given type is a valid substitution if it *models* the concept. Having provided these basic definitions, the mapping with our proposal is straightforward. A *concept* is captured with a template classifier representing a contract. IterableForward (illustrated in Fig. 1) is thereby a concept, and CollType, ValueType and IndexType are its associated types. Activities *first*, *last*, *next* and *get* are used to describe the signatures of functions that should be available for a given type to *model* the concept. The fact that a type (for example Vector of Fig. 2) *models* a concept is captured by the combined usage of Realization and TemplateBinding relationships.

Concerning works more directly related to UML 2 templates and the expression of constraints on type parameters, we have shown in previous publications [2] [3] how classifiers could be parameterized with policy classes by using non-substitutable constraining classifiers. Except these works, there are (to our knowledge) no other publications directly addressing the subject.

4 Conclusion

UML 2 provides support for template-based modeling, as well as dedicated mechanisms for expressing constraints on type parameters. Expressing such constraints is crucial when considering behavioral aspects. For that purpose, we have proposed modeling patterns related to the usage of substitutable constraining classifiers. Our middle-term goal is now to put these proposals in practice for the definition of a generic library similar to the STL of C++ (i.e., generic collection types and iterators). While we have explained that the mechanisms we have proposed directly map to fundamental notions of generic programming, it would not make sense to directly map the hierarchy of *concepts* (with their associated requirements) of the STL, as it is strongly influenced by the facilities of the language (i.e., pointers, increment and dereferencing operators, etc.). Further studies are therefore required to adapt this hierarchy, taking into account the specific properties of UML.

References

1. OMG: Unified Modeling Language: Superstructure version 2.2 (2008)
2. Cuccuru, A., Mraidha, C., Terrier, F., Gérard, S.: Templateable Metomodels for Semantic Variation Points. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 68–82. Springer, Heidelberg (2007)

3. Cuccuru, A., Mraidha, C., Terrier, F., Gérard, S.: Enhancing UML Extensions with Operational Semantics. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 271–285. Springer, Heidelberg (2007)
4. Myers, N.: A New and Useful Template Technique: "Traits". In: Lippman, S.B. (ed.) C++ Gems, pp. 451–457. SIGS Publications, Inc., New York (1996)
5. Austern, M.H.: Generic Programming and the STL. Addison-Wesley Pro., Reading (1999)
6. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., Lumsdaine, A.: Concepts: Linguistic Support for Generic Programming in C++. In: OOPSLA 2006, pp. 291–310. ACM, New York (2006)