Received December 13, 2021, accepted January 4, 2022, date of publication January 18, 2022, date of current version January 25, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3143898

UML Templates Distilled

JOSÉ FARINHA[1] AND ALBERTO RODRIGUES DA SILVA[2]
[1]Harlow Inc., 1649-003 Lisbon, Portugal
[2]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, 1000-029 Lisboa, Portugal

Corresponding author: Alberto Rodrigues da Silva (alberto.silva@tecnico.ulisboa.pt)

This work was supported in part by the Portuguese Fundação para a Ciência e Tecnologia (FCT) under Grant UIDB/50021/2020.

ABSTRACT UML templates are possibly the most neglected and misused piece of knowledge in UML modelling. This subject has been disregarded in the research and practice literature and even by modelling tools providers. This paper suggests that such oblivion results from a general misunderstanding that UML templates are just graphical representations of genericity like it is found in programming languages, and from the insufficient support from the modelling tools, with a consequence of poor usage of UML templates in practice. Indeed, the capabilities and potential of UML templates are far-reaching. Increasing awareness around them could bring significant benefits for UML users, namely, higher-level abstraction and reuse. Therefore, this paper provides a distilling tutorial on UML templates to highlight their flexibility and advantages. That presentation follows a tutorial style and is supported by several illustrative examples, varying from simpler to more complex ones. This tutorial reviews the Template construct's core concepts and terminology, presents constraining classifiers and shows how to define properties and operations as template parameters. Then, it presents and discusses advanced aspects such as operation templates, parameter defaults, the relationship between binding and generalization, and the specific semantics of package templates. Furthermore, the paper discusses the related work and uncovers some of the UML templates' limitations and opportunities for improvement.

INDEX TERMS Object-oriented modelling, genericity, UML, templates, UML templates.

I. INTRODUCTION

UML templates have been cast to oblivion ever since they were introduced in the preliminary versions of the language specification [1]. Indeed, UML templates have been poorly described and used in practice. Despite there are plenty of academic and technical books on UML (e.g., [2]–[5]), those have only addressed templates in a brief and shallow way, as if to provide just a general hint of some already known subject. Quite differently, OMG's standard UML documentation [1] provides detailed information on templates, but it is far from a clear and cohesive source. It lacks good usage-oriented examples, scatters the information on templates along with multiple chapters (reflecting the various kinds of templates in UML), and is almost silent about crucial aspects that deserve far more elaboration and emphasis. Consequently, there is a real need for a better bibliography since, between too shallow and too formal sources, nothing can be found that provides a clear, comprehensive, and well-focused explanation of UML templates.

The associate editor coordinating the review of this manuscript and approving it for publication was Derek Abbott.

Similarly, modelling tools hardly support UML templates. Notably, most of the current UML editors do not consider the full spectrum of UML template features, like all possible kinds of templates and their parameters. Also, to our best knowledge, none of the existing tools supports the expansion semantics defined by the language [1].

Some reasons help to explain this situation. One of them is that UML templates are mainly used as just representations for generics in object-oriented programming languages (OOPLs). This implies that their processing in UML is unnecessary and that it suffices to have them translated to the target language syntax. Indeed, UML tools only support templates in what they have in common with OOPLs' generics and, thus, no processing is offered by such tools for the sake of the semantics defined in UML for templates. Nothing specific to UML templates is supported, and both the templates' definitions and their instances are simply translated onto the syntax of target OOPLs.

However, this is arguably a general misinterpretation: UML templates are more than representations for OOPLs' generics. Firstly, UML templates are more abstract and expressive than most mainstream OOPLs' generics. This derives as much from UML's higher level of abstraction and wide-ranging set of constructs as from the language's bold approach to genericity. Regarding this latter aspect, UML provides more kinds of template and template parameters than OOPLs use to support. For instance, while in most OOPLs only types and methods can be templatized, in UML packages can also be templates.

Furthermore, since in UML the concept of Type unfolds to many constructs (such as Class, Association, Use Case, and State Machine), there is a large spectrum of development artefacts that can be templatized. Regarding template parameters, in OOPLs only types and values are allowed. In UML, properties, operations, and packages are also accepted as template parameters. UML also provides two semantics for integrating template instances with the model elements that embody them: (1) semantics based on inheritance for classifier templates, and (2) semantics based on package merge for package templates, while OOPLs only provide the former. Thus, UML templates are more than representations of programming languages' generics. Unfortunately, current literature has failed at stressing this out.

Secondly, UML template instances cannot be black boxes that are merely transformed and translated to target languages' generics. If templates are to be used in modelling, then the contents of every template instance must be visible and usable by other parts of the model. For example, if a class *SixPack* is defined as *Set<Beer, 6>* in a model, then *SixPack*'s features – such as *add (Beer)* and *get* () : *Beer* – have to be generated and made available for use in the context of that model. This is inconsistent with the claim that *Set<Beer, 6>* must be transposed to the target programming language and only processed at that level. Hence, although UML templates can be transcribed onto OOPLs' generics (to take advantage of genericity at the target language level, too), they must be also semantically processed in the context of UML. Once again, UML templates hardly can be taken as mere shortcuts to OOPLs' generics.

This paper intends to contribute to a better understanding and use of UML templates. This contribution is pertinent because the potential of UML templates has mainly been underrated. The paper provides a distilling presentation of the capabilities of templates, mostly to stress out that the features of UML templates are not found in the common OOPLs. Specifically, the focus is on the merge semantics of package templates, the broad spectrum of template and template parameter kinds, and the possibility of building model element names from template arguments.

The paper has five sections. Section 2 briefly introduces the background of generics and templates as defined originally in popular OOPLs and UML. Section 3 provides an extensive tutorial on UML templates based on several illustrative examples and discusses their main aspects. Section 4 discusses the related work. Finally, Section 5 concludes with a summary of the main contributions and the discussion of open issues.

II. BACKGROUND

Generic programming (GP) is frequently described informally as a programming paradigm by which code is written in terms of types "to be specified later". Concrete pieces of code are obtained instantiating generic code when needed for specific types by providing these as parameters. GP is supposed to scale development since the effort of defining a generic piece of code and instantiating it multiply is less than the required to program multiple, similar pieces of code in a repeatable way.

GP was pioneered by the ML language in 1973 [1], but it only got widespread visibility with Musser and Stepanov's contributions that motivated the design of the C++'s Standard Template Library [2]. According to Musser and Stepanov, GP "centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software" [2]. That means that programming languages shall allow the writing of algorithms independently of the concrete types they will operate. Consequently, those algorithms are programmed only once and instantiated multiple times whenever required.

GP has been implemented by most programming languages, however with nuances: GP means different things for different languages, depending on what is considered an "algorithm" and a "type" for each language. I.e., the level of genericity in a language depends on what kind of elements in source code can be defined as a template and what can be considered a template parameter. According to Dos Santos and Jarvi [1], there are two main approaches: one follows the idea of gradual lifting of concrete algorithms, as prescribed by Musser and Stepanov; the second has its roots in an algebraic view of datatypes [2] and a calculational approach to program construction [3], it is also known as *datatype-generic programming* [4], and is commonly found in functional languages like Haskell, Standard ML, or Clojure.

In object-oriented languages, genericity mechanisms are commonly associated with the *Class* construct. As such, merging generic code instances with other code generally relies on the subclassing mechanism. For example, if class *Team* is defined as a set of *Person* objects, then *Team* can be defined as a subclass of *Set <Person>*. If teams have a name, a budget, and a manager, such properties are then defined on the class *Team*. Also, if adding a person to a team is to be conditioned to specific criteria, then the method *Set<Person>.add (Person)* must be redefined in *Team* to check the necessary criteria before forwarding the call to *Set <Person>* (*Team*'s superclass).

The principle seems reasonable, but it works well only if the generic piece of code does not span more than a single class. It has been shown that subclassing is not an effective merging technique when dealing with *family polymorphism* [5], i.e., in case the generic piece of code encodes a complex concept that encompasses multiple interrelated classes. The typical example is that of a generic *Graph <N, E>*, which can be instantiated as *Road Map <City, Road>*, *Social Network <Person, Friendship>*, etc. But also, most of the GoF design patterns [6] are notable cases of family polymorphism [7], [8]. And even when a concept or design pattern is represented with nested classes, subclassing reveals limitations [9]. To cope with this problem, the concept of *package template* was proposed with semantics that does not rely on subtyping [10]. The newness of package templates adds compositional semantics to *package import*. On package template instantiation, generic code is composed with the target package's code, element by element and, recursively, with elements paired according to their name's equality. Adding to this, that element renaming is allowed on instantiation, package templates bring flexibility and control over the merging of generic code instances with other code, removing the limitations of subtyping-based genericity. Therefore, package templates widened the possibilities of genericity. Although they have not been fully implemented in any programming language, their emulations on top of Java and Groovy have already been approached in [11] and [12], respectively.

Generic programming exists in many flavours. Hence, because UML aims at being a *lingua franca* among programming languages, it should cover most of those flavours. This must be why it encompasses a broad set of genericity capabilities. In the beginning (at least as of version 1.3, the oldest that can be backtracked today), the approach to genericity was bold, meaning that, although the emphasis was on class templates, all kinds of a model element could be declared as a template [13] (p. 2-66): "a template represents the parameterization of a model element, such as a class or an operation, although conceptually any model element may be used (but not all may be useful)". The downside was that template parameters were not model elements in their full right [13] (p. 2-40): "Each parameter is a dummy *ModelElement* designated as a placeholder (...) the template parameter element lacks structure. For example, a parameter that is a Class lacks Features; they are found in the actual argument". The direct consequence of this was that, since any template will reference its parameters and these are not fully specified elements, the well-formedness of the template itself could not be validated. Only its instances could. Therefore, UML 1.3 templates were like expansion macros in programming languages.

As of UML 2.0, templates became more pragmatic as well as conservative. For the sake of pragmatism, not all element kinds could be defined as a template or as a template parameter, only those classified as "templateable" and "parameterable" could, respectively. Even so, UML templates are still far-ranging in terms of genericity. Packages are among the templateable elements, with the merge-based semantics aforementioned. Regarding conservatism, template parameters are no longer dummy elements, but fully operational ones, enabling well-formedness checking for every component.

From UML 2.0 to the current version of the language, version 2.5.1, templates have matured along with the same approach, and it can be said that their recent formalization is up to their potential. Despite that evolution at the UML specification level, current modelling tools and bibliography have not kept the same pace. They have contributed to the poor adoption of UML templates in the modelling practice. For example, tools like Enterprise Architect [14], MagicDraw [15], PowerDesigner [16], StarUML [17], and Visual Paradigm [18] fall short at supporting the current capabilities of UML templates. Regarding bibliography, the current UML specification [19] does not provide a pedagogic explanation on how to use them in practice; neither books nor scientific papers exist that offer an enlightening description of the matter. (A general analysis of the related work is further presented in Section 4.)

However, we believe that increasing awareness around UML templates can significantly benefit UML users. So, this paper provides a distilling tutorial on UML templates that highlights their flexibility and advantages. That presentation follows a tutorial style and is supported by several illustrative examples, as follows in the next section.

III. TUTORIAL ON UML TEMPLATES

This section presents the main aspects and details of UML templates as aligned with the current version of the UML (i.e., version 2.5.1) [19]. The presentation follows a tutorial style and discusses several illustrative examples. Most of these examples are connected to the goal of sorting objects, varying from simpler (in Section 3.1) to more complex (in the remaining subsections), to show how new requirements pose new challenges and how UML templates respond to these. This section presents the following: the core and basic concepts of templates, how to constrain classifiers, how to expose operations as template parameters, how to expose properties as template parameters, how to use operation templates, how to use parameter defaults, discuss the relationship between binding and generalization, and finally how to use package templates.

A. CORE CONCEPTS

The concept of *template* in UML allows creating generic model elements intended to be reproduced multiple times wherever the problem they address happens. A *UML template* is a reproducible model element written in terms of other elements (types, properties, operations, etc.) that are meant to be abstract. These elements are said *exposed as parameters* of the template and are meant to be replaced by concrete elements wherever the template is instantiated. The substitution by elements of the target model contextualizes the template to that model. For instance, in Fig. 1, Array is a class template defined in terms of T and k, which are exposed as parameters.

A template is recognized graphically by a dashed rectangle on its top-right corner, as shown in Fig. 1. The dashed

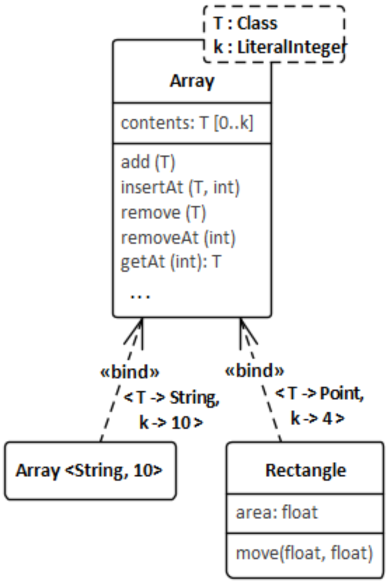


FIGURE 1. **Example of a simple class template.**

rectangle represents the *template's signature*, which lists the template's parameters.

A reproduction of a template in a target model is called an *instance of the template*. The construct in UML for the instantiation of templates is the *binding* relationship, a directed relationship from the instance to the template. It allows to get an element entirely defined as an instance of a template (e.g., *Array <String, 10>* is an instance of the *Array <T, k>* template) as well as have an element with specifications of its own having those merged with an instance of a template (such as *Rectangle* in Fig. 1, which hosts an instance of the Array<T, k> template). In either case, the element hosting the template instance is said to be *bound* to the template. Also, in the context of that binding, it is referred to as the *bound element*.

1) TEMPLATE ELEMENTS

The UML model elements that can be defined as templates (these said *templateable* elements) are all those subsumed by the metaclasses *Classifier*, *Package*, *Operation,* and *String Expression*. *Classifier* templates encompass all elements that may have instances, including classes, interfaces, associations, use cases, activities, and state machines. *Package templates* can be defined to pack recurring model fragments that surpass the context of a single classifier. *Operation templates* are intended to abstract the commonalities among similar operations concerning their signatures and bodies. Finally, *String Expression* templates define textual patterns that may be used to generate strings in a disciplined way. These generated names can then be used as literals, comments text, or to name model elements. The complete taxonomy of templateable elements in UML [19] is shown in Fig. 2.

A template may not be used as a common, non-template element. For instance, a class template is not allowed to

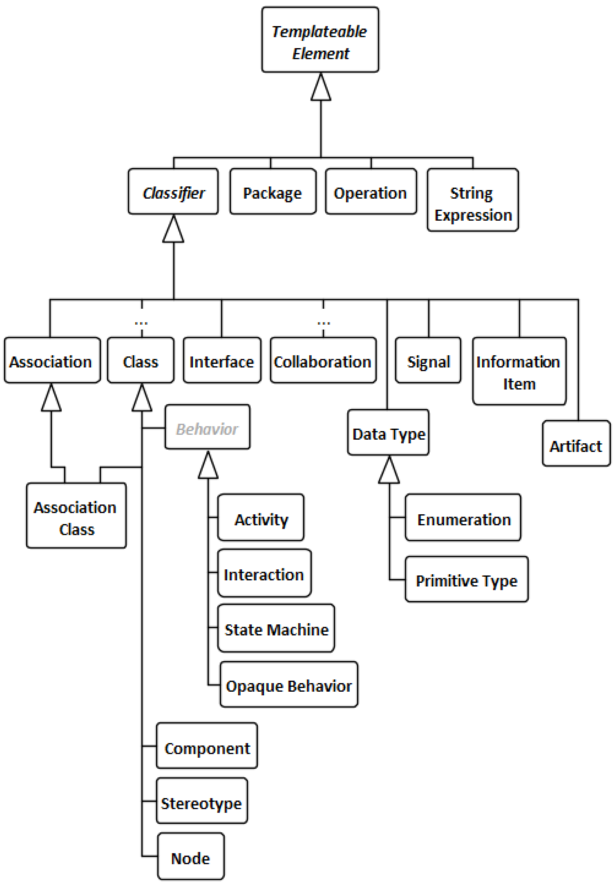


FIGURE 2. **Taxonomy of templateable elements (adapted from [19]).**

classify objects, except in the body of the template itself or of a related template. That means that statements like "new MyClassTemplate" or "myAttribute: MyClass Template" are not allowed in common model elements, only inside templates and under certain conditions (not relevant at this point).

2) TEMPLATE PARAMETERS AND ARGUMENTS

Template parameters declare what elements, amongst those participating in the definition of the template, represent general or non-concrete concepts, those that must be replaced by concrete elements for complete instantiation of the template. Such elements are said *exposed as parameters* or *parametered elements* of the template under consideration. If any parametered elements are not replaced, then the template instance is still a template, not a plain model element.

Strictly, UML differentiates a parameter from the element it exposes. The parameter is a declaration superimposed to the parametered element, making it replaceable when bounding to the template. Additionally, the parameter construct allows for the specification of a default substituting element, which will be applied if the parameter is not explicitly substituted in a binding.

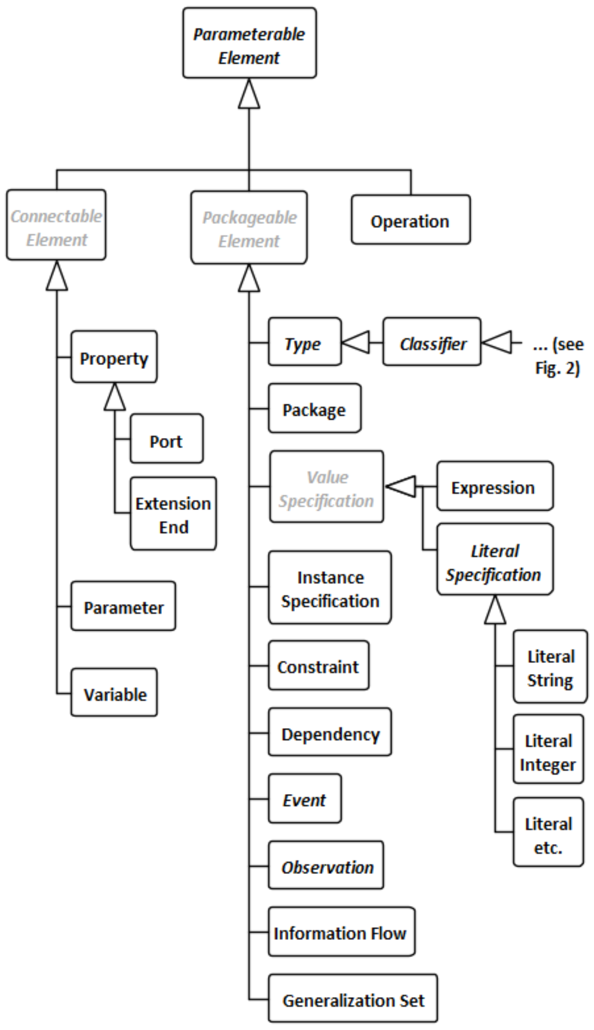


FIGURE 3. Taxonomy of parameterable elements (adapted from [19]).

For brevity reasons, a parametered element is sometimes referred to simply as a parameter in this text. Fig. 3 shows the complete taxonomy of parameterable elements.

The UML construct for assigning an actual argument to a formal template parameter is called *Substitution*. It is said that the *argument* substitutes *the parameter*, meaning that inside the template instance every reference to the parametered element will be substituted by a reference to the actual argument. In this text, for brevity, an actual argument is referred to as the *substitute*. For instance, in Fig. 1, class *Array* is a template with two parameters: T and k. T's type is *Class*, meaning that it must be substituted by a class. k's type is *Integer Expression* and therefore must be substituted by expressions that evaluate to *Integer* values, including those expressions merely composed of a single literal constant.

The name and type of a template parameter are determined by the element it exposes. The name is directly adopted by the parameter, meaning that in Fig. 1, the first parameter is named "T" because it exposes a class named "T" (not shown in the figure), and parameter k exposes an expression named "k" (also not shown). Consequently, the usual naming conventions in UML apply, including the syntax associated with namespaces. Namely, if a template exposes a feature of an external class, the parameter's name is qualified (e.g., *T::name*). Also, a parameter exposing an operation is named upon the full signature of the operation: its name and its ordered list of parameter types, the latter including the operation's return type. E.g., a template parameter exposing the operation *getField (fieldName: String): String* will be named "getField (String): String". In this text, template parameters' names will be sometimes abbreviated for the sake of clarity when ambiguity does not occur.

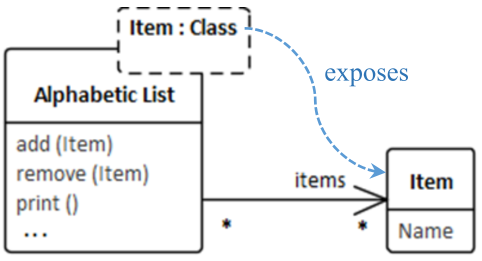


FIGURE 4. Definition of a class template with a class parameter.

Any model element accessible from a template may be exposed as a parameter of that template. In Fig. 4, *Alphabetic List* is a class template with one parameter: *Item*. (The dashed line labelled "exposes" is merely illustrative because UML has no graphical notation that links a parameter to the element it exposes.)

3) BINDING RELATIONSHIP

A binding to the above template is shown in Fig. 5. The semantics of the *binding relationship* makes everything specified for a template valid for the bound element as if the whole contents of the template were copied into the bound element. In the general case, such a copy must be adapted to the context of the bound element. This is done using substitutions that are specified in the context of the binding.

In this example, the bound class is *Alphabetic List <Person>*, which is said anonymous. This means that if a name is not given to a bound element, it will be named after the template and the actual arguments specified for the binding under consideration. The name will be formed using the following string pattern: "*TemplateName <argument1, argument2, ...>*". In the example, the single template parameter (*Item*) is substituted by class *Person*, a fact that is textually notated in the form '*parameter − > substitute*' next to the graphical representation of the binding. The semantics of the binding relationship is illustrated with this example on the right: *Alphabetic List<Person>* is a reproduction of *Alphabetic List* where all references to *Item* are replaced by references to *Person*. In the template, one of the references to *Item* is done through a property *item*, which is also an association-end connecting to class *Item*. Therefore,

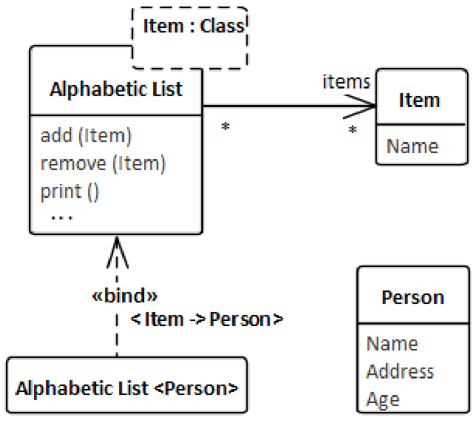


FIGURE 5. Binding example, featuring an anonymous class.

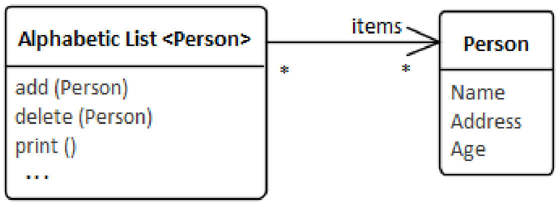


FIGURE 6. Binding to a class template, semantics expanded view.

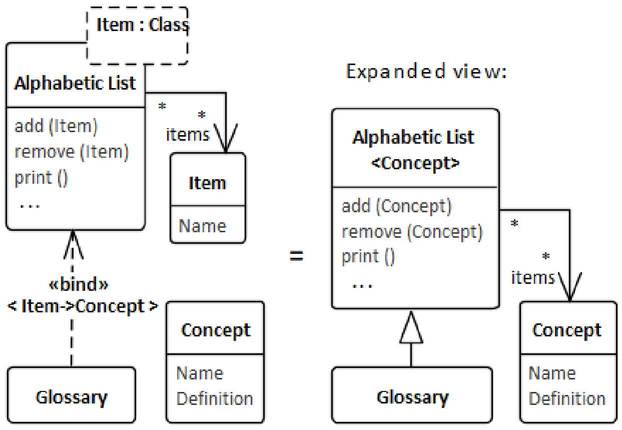


FIGURE 7. Binding to a class template featuring a named class.

Alphabetic List<Person> gets a property that references *Person*. The diagram of Fig. 6 is called the expanded view of the binding.

Fig. 7 shows another binding to the same template, from the class Glossary. The semantics of the Binding relationship is defined as shown by the expanded view on the right of that figure. Fig. 8 shows a bound class with specifications of its own. In such cases, the binding merges the instance of the template with the contents of the bound class. For class templates, such merging is done through inheritance.

The purpose of the *Alphabetic List* is to maintain a list of items sorted alphabetically. To perform the ordering, *Alphabetic List*'s methods assume an attribute called *Name*

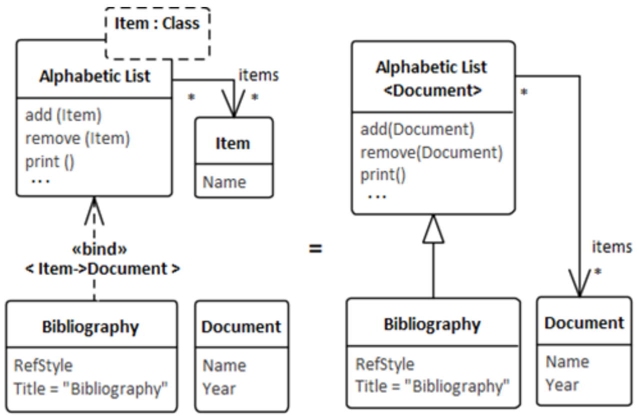


FIGURE 8. A bound class with contents of its own.

in *Item* and use it as the ordering criteria (behaviour not shown in the figure). Since the methods of *Alphabetic List* are copied, undergo substitutions, and are then inherited by the bound classes, these will also use a *Name* attribute. Therefore, the *Item* must be substituted by a class with a *Name* attribute. If it doesn't, the methods of the bound class will not compile. The situation is exemplified in Fig. 9: Document has a *Title* attribute, instead of *Name*; therefore, every expression in the template referring to *Item*'s *Name*, once transposed into class *Bibliography*, will refer to a non-existing *Document*'s *Name* and raise compilation errors. For instance, expressions '*it.name*' and '*iter.name*' will not be compilable in *Bibliography*.

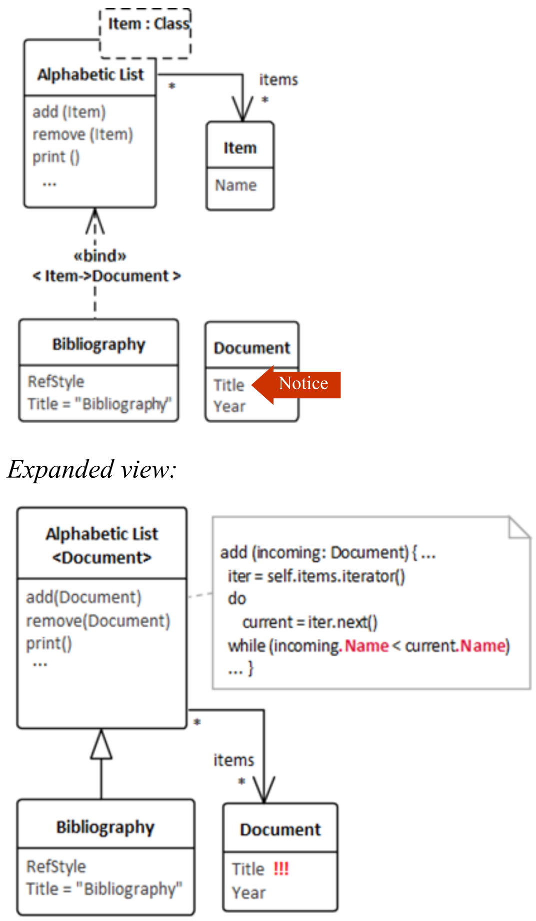
Cases like the example in Fig. 9 require flexibility regarding the attribute corresponding to *Name* in the substituting class. That is why UML allows properties (attributes) to be exposed as template parameters. In the example, exposing *Name* as a template parameter allows it to be substituted by the appropriate attribute in every binding under consideration. The definition of the template becomes as in Fig. 10.

For typed elements, which is the case of properties, UML establishes that the type of a substitute must conform to the type of the parametered element it substitutes. Therefore, the Name must be substituted by an attribute whose type is String or a subtype of it, as in the example in Fig. 10.

B. CONSTRAINING CLASSIFIERS

A more general template for keeping ordered lists would allow ordering criteria based on more than one attribute or a calculated expression. That could be implemented if the template considers a comparator method, say compareTo (T) defined in T, which encodes the ordering criteria internally, as shown in Fig. 11.

This template is only instantiable if the class substituting *T* responds to *compareTo (...)*. A binding functionally equivalent to the one previously shown in Figure 6 would be one as in Fig. 15. It may be noted that class Person does define compareTo (Person).



```
add (incoming: Document) { ...
  iter = self.items.iterator()
  do
    current = iter.next()
  while (incoming.Name < current.Name)
  ... }
```

FIGURE 9. A bind that produces non-compilable code.

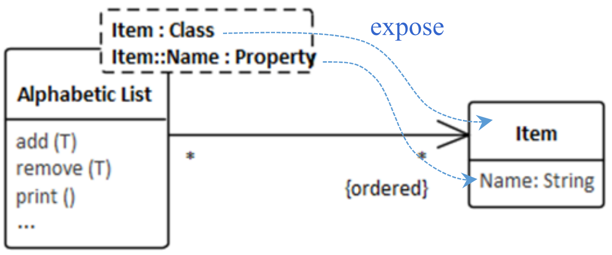
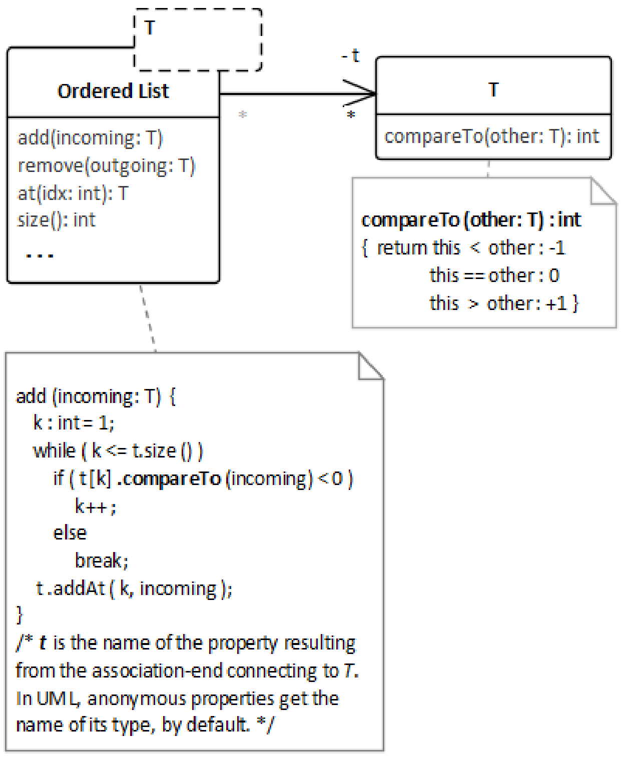


FIGURE 10. Definition of a template with a class parameter and a property parameter.



```
compareTo (other: T) : int
{ return this < other : -1
         this == other : 0
         this > other : +1 }
```

```
add (incoming: T) {
  k : int = 1;
  while ( k <= t.size () )
    if ( t[k] .compareTo (incoming) < 0 )
      k++ ;
    else
      break;
  t .addAt ( k, incoming );
}
/* t is the name of the property resulting
from the association-end connecting to T.
In UML, anonymous properties get the
name of its type, by default. */
```

FIGURE 11. Class template *Ordered List*.

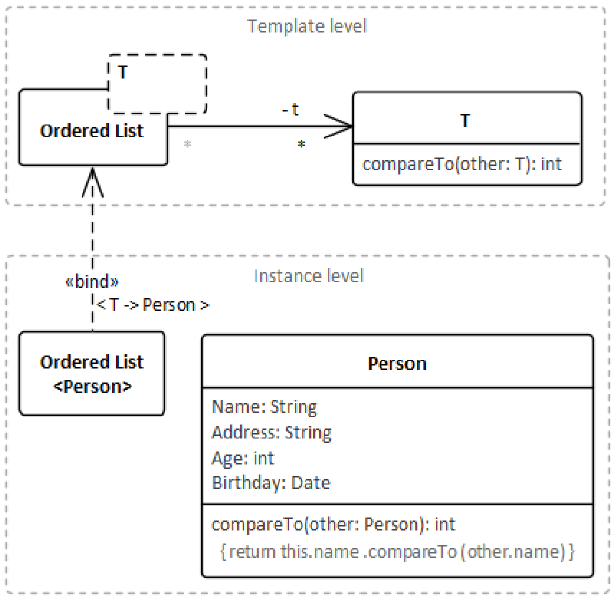


FIGURE 12. Class *Ordered List <Person>*.

The way to ensure safe substitutions in scenarios like this one – i.e., to ensure that every substituting classifier provides the features that the template assumed (in this case, *compareTo* (T)) – is by declaring a *constraining classifier* for the template parameter. A constraining classifier is specified by appending "> *ConstraingClassifierName*" to the parameter's declaration.

In the example, it would be T : *Class* > *Comparable*<T>, where *Comparable*<T> is a classifier that publicly provides a *compareTo* (T) operation, or, more generally, *compareTo* (*Comparable*<T>), as in the Java platform. Template parameters with a constraining classifier are must be substituted only by subtypes of that classifier. In the example, T may only be substituted by interfaces that specialize *Comparable*<T> or by classes that implement it. To have a full guarantee that the provided constraining classifier declaration corresponds to checking all the assumptions made by the template about parameter T, UML enforces that T shall

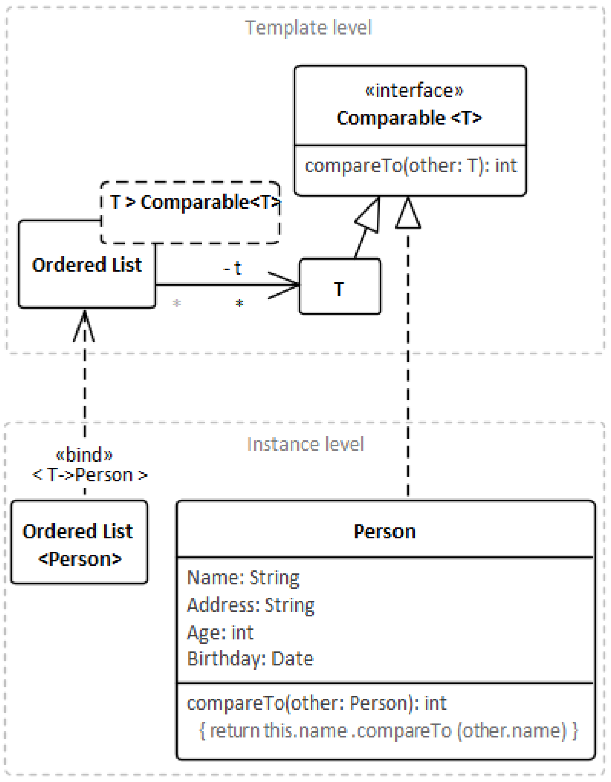


FIGURE 13. Constraining classifier.

be a direct subtype of the constraining classifier and that it provides no additional features. That ensures every feature of *T* used by the template exists in the constraining classifier and, by definition, in every subtype of it. The new definition of the example template and a binding to it are shown in Fig. 13. There can be more than one constraining classifier for a template parameter when needed.

C. EXPOSING OPERATIONS AS TEMPLATE PARAMETERS

Because the ordering is controlled uniquely by *compareTo(...)*, the template in Fig. 13 cannot provide multiple ordering criteria for the same class of objects. In the example above, *Ordered List <Person>* can only order *Person* objects by name. Having also a list ordered by age is not viable using that template. Prospective solutions can be equated, such as adding a parameter to *compareTo (...)* to specify the ordering criteria, but none of these are likely elegant. Alternatively, the ability to expose operations as template parameters in UML allows for a simple solution: if *compareTo (...)* is exposed as a template parameter, it is possible to substitute that operation with one that implements the desired ordering criteria. Fig. 14 shows the new parameter and two bindings that substitute it. The actual name of the new template parameter in the example is *T::compareTo (Comparable<T>): int*. In the figure, this name is sometimes shown stripped of the namespace, parameter, and return type, for the sake of legibility.

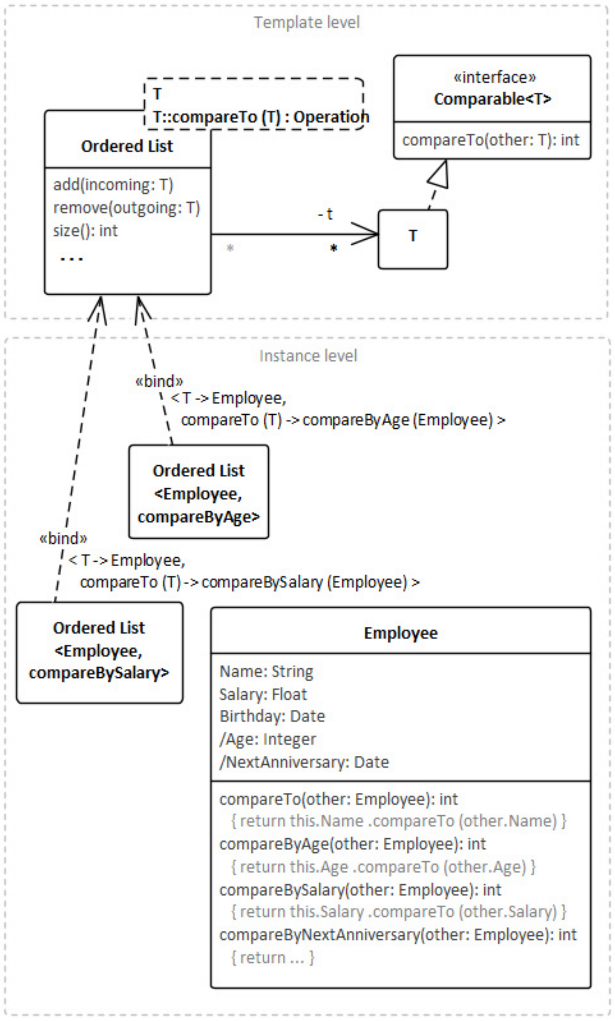

Template level
T
T::compareTo (T) : Operation
«interface»
Comparable<T>
compareTo(other: T): int
Ordered List
add(incoming: T)
remove(outgoing: T)
size(): int
...
- t
*
*
T
Instance level
«bind»
< T -> Employee,
compareTo (T) -> compareByAge (Employee) >
Ordered List <Employee, compareByAge>
«bind»
< T -> Employee,
compareTo (T) -> compareBySalary (Employee) >
Ordered List <Employee, compareBySalary>
Employee
Name: String
Salary: Float
Birthday: Date
/Age: Integer
/NextAnniversary: Date
compareTo(other: Employee): int
{ return this.Name .compareTo (other.Name) }
compareByAge(other: Employee): int
{ return this.Age .compareTo (other.Age) }
compareBySalary(other: Employee): int
{ return this.Salary .compareTo (other.Salary) }
compareByNextAnniversary(other: Employee): int
{ return ... }

FIGURE 14. Operation template parameters.

D. EXPOSING PROPERTIES AS TEMPLATE PARAMETERS

All the *compareBy...(...)* operations have a similar definition. That suggests such definition could be abstracted out into a template. These operations differ only in the *Employee*'s property being compared (e.g., name, age, salary). Since UML allows properties to be exposed as template parameters, abstracting these operations into a template is straightforward, as shown in Fig. 15.

Each binding generates an operation *compareBy <property> (...)* in *Employee*.

UML 2.5.1 enforces that a property is only substituted by a property of the same type or of a subtype ([19, Sec. 7.8.18.5 and 9.9.17.7]). In this case, since the parametered property's type is *Comparable<Object>*, the question that may arise is whether *String*, *Float*, *Date*, and *Integer* are subtypes of *Comparable<Object>*. Indeed, they do. That's because the *Comparable<T>* template is covariant relatively to its parameter *T*, which means that if *S* is a subtype of *Object*, then *Comparable<S>* is a subtype of *Comparable<Object>*.

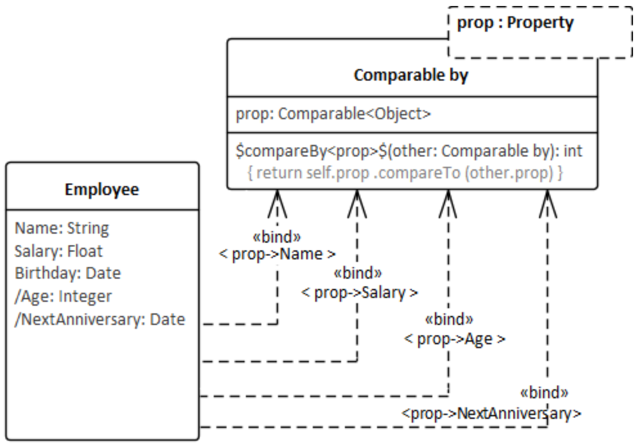


FIGURE 15. Generalizing an operation through a class template.

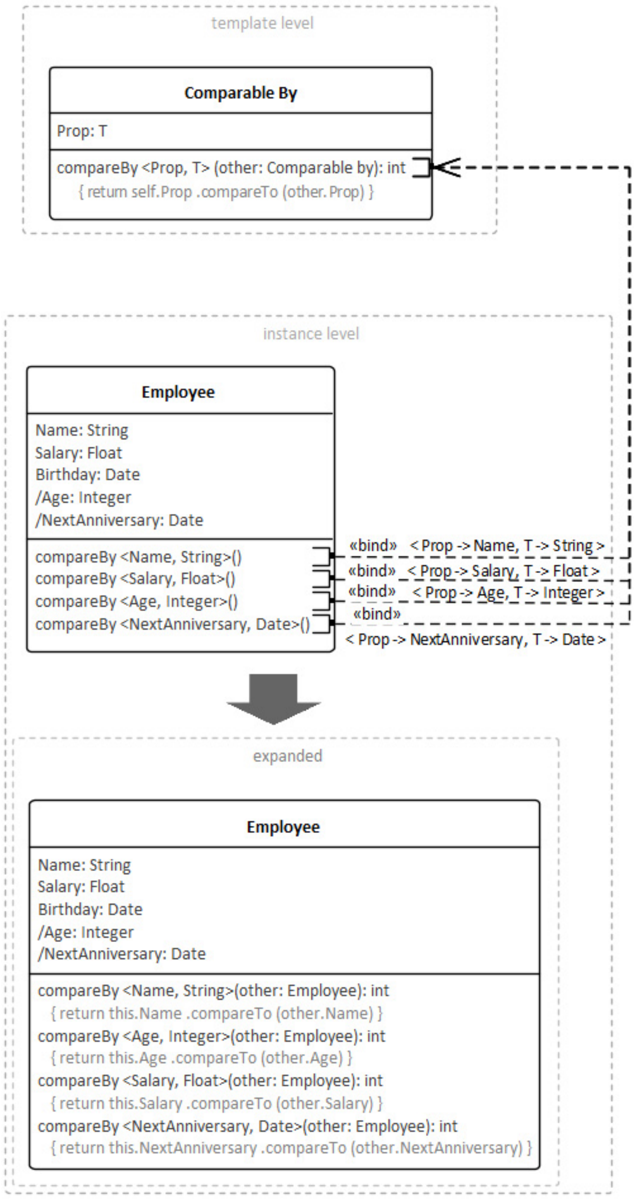

template level
Comparable By
Prop: T
compareBy <Prop, T> (other: Comparable by): int
{ return self.Prop .compareTo (other. Prop) }
instance level
Employee
Name: String
Salary: Float
Birthday: Date
/Age: Integer
/NextAnniversary: Date
compareBy <Name, String>()
compareBy <Salary, Float>()
compareBy <Age, Integer>()
compareBy <NextAnniversary, Date>()
«bind» < Prop -> Name, T -> String >
«bind» < Prop -> Salary, T -> Float >
«bind» < Prop -> Age, T -> Integer >
«bind» < Prop -> NextAnniversary, T -> Date >
expanded
Employee
Name: String
Salary: Float
Birthday: Date
/Age: Integer
/NextAnniversary: Date
compareBy <Name, String>(other: Employee): int
{ return this.Name .compareTo (other.Name) }
compareBy <Age, Integer>(other: Employee): int
{ return this.Age .compareTo (other.Age) }
compareBy <Salary, Float>(other: Employee): int
{ return this.Salary .compareTo (other.Salary) }
compareBy <NextAnniversary, Date>(other: Employee): int
{ return this.NextAnniversary .compareTo (other.NextAnniversary) }

FIGURE 16. Generalizing an operation through an operation template.

E. BUILDING ELEMENT NAMES WITH PARAMETERS

Fig. 15 shows one additional capability of UML templates: the possibility of using parameters to construct element names. In the example, the name of the generated operation is built concatenating "compareBy" with the name of the property that happens to substitute *prop*. In a definition of a template, a pair of "$" characters indicates that the enclosed string includes parameter names embraced by '<' and '>'. In every bound element, such parameter names are automatically replaced by the names of the corresponding substitutes. In the example, the following operations will be created in *Employee*: *compareByName (...)*, *compareBySalary (...)*, *compareByAge (...)*, and *compareByNextAnniversary (...)*.

F. OPERATION TEMPLATES

Operations may also be defined as templates in UML. As an example, the previously shown *compareBy<prop>(...)* can be made generic as a template by itself, instead of defined inside a class template, as above. The notation for operation templates is only textual: the name of the operation template is suffixed with the signature of the template embraced by '<' and '>'. Hence, *compareBy<prop>(...)* defined as an operation template is shown in Fig. 16 (the template signature is simplified, for clarity).

Fig. 16 shows class *Employee* in two ways: its regular specification on top and its expanded view at the bottom. The expanded view shows the full, flattened definition of the operations. These get the same signature and body of the operation template plus superimposed substitutions. These are those specified for each binding plus the tacit substitution of the owner of the operation template (class *Comparable By*) by the owner of each bound operation (class *Employee*) –note that the type of the *other* parameter is *Comparable By* in the template and *Employee* in the bound operations. There is no reference to this tacit substitution in the UML 2.5 standard document, but it is assumed in this paper.

Hence, there are two alternatives for making *compareBy <Property> (...)* generic: a class template member and an operation template. These seem roughly equivalent. However, the operation template alternative has a slight advantage: it is simpler to specify a non-default name for the bound operation. As shown in the example in Fig. 17 and Fig. 18, where the bound operation is meant to be called *northOf (...)* and to return the difference of the *y* component between two coordinates, the bindings with either alternative are: (1) With a class template, an extra parameter is needed to supply the name for the operation, as in Fig. 17; or (2) With an operation template the name of the bound operation may be set directly, as shown in Fig. 18.

Therefore, since an extra parameter is avoided, both the template definition and the binding are simpler in the latter case.

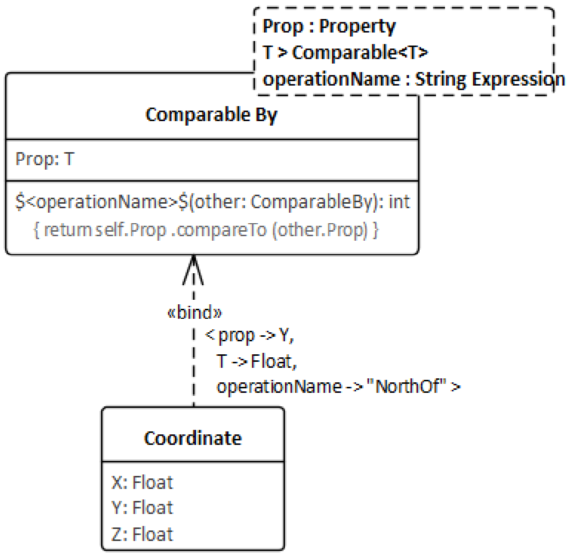

Prop : Property
T > Comparable<T>
operationName : String Expression
Comparable By
Prop: T
$<operationName>$(other: ComparableBy): int
{ return self.Prop .compareTo (other.Prop) }
«bind»
< prop -> Y,
T -> Float,
operationName -> "NorthOf" >
Coordinate
X: Float
Y: Float
Z: Float

FIGURE 17. Supplying a name for an operation in a bound class.

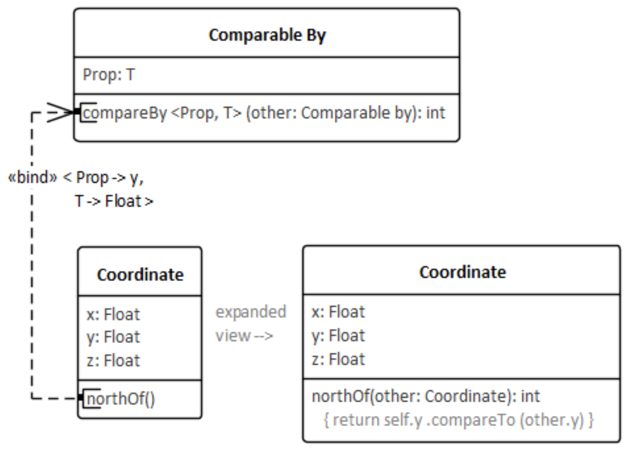


FIGURE 18. Supplying a name for an operation bound to an operation template.

G. PARAMETER DEFAULTS

Fig. 17 also illustrates the possibility of declaring default substitutes for template parameters. Those are specified in UML after a "=" appended to the name and type of the parameter. The default is used in every binding that does not explicitly substitute for the parameter. In the example, the default substitute for the parameter *operationName* is the "$compareBy<prop>$" string expression. Hence, if a direct substitute were not supplied in the binding in Figure 17, the operation's name would be "compareByY".

H. BINDING AND GENERALIZATION

This section shows how the Binding and Generalization relationships relate to each other. The following aspects are highlighted: (1) The effect of a generalization among class templates; and (2) generalization and binding as two different constructs for reuse and refinement.

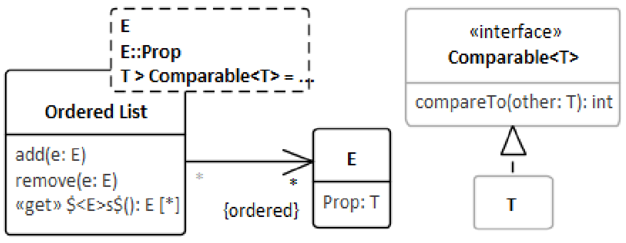


FIGURE 19. A template for ordering by a property.

As a motivating example, one more variant of the templates presented so far is shown in Fig. 19, and its evolution is subsequently undertaken. This new version of the *Ordered List* template does the ordering of *E* objects based on a property *Prop* of those. To ensure that the values of *Prop* are orderable, its type *T* implements interface *Comparable<T>*. The same is guaranteed for every substitute of *T*, since *Comparable<T>* is also a constraining classifier for *T*.

It must be noted that *prop*, by being declared public, represents a property in the C# sense: it models a pair of operations – *getProp ( )* and *setProp (value)* – that mediate the access to a private implementation of the property; these operations are transparently called whenever *prop* is read or written, respectively, from outside the class. In this paper, these *get* and *set* operations are declared as *≪get≫ prop ( )* and *≪set≫ prop (value)* for legibility reasons, as the correspondence between these operations and the related property gets easier to spot.

The ordering of the elements is introduced by operation *add (E)*, which is the only way to add elements to the list (note that *Ordered List::e* is private). Hence, the accuracy of the ordering of objects already added to the list is only guaranteed while their *prop*'s values do not change. If *prop* is mutable, an update to *prop* may require the object in question to undergo a repositioning. Therefore, to preserve the ordering, the list must be notified of every update to its elements' ordering attribute and react accordingly. This is a clear instance of the *Observer* pattern [6]. *Ordered List* is an observer of *E::prop* and the reaction to updates on this property may be provided by a new operation *reorder (E)*.

Applying the general principle of modularity, the above *Ordered List* may be kept as an adequate solution for immutable objects, while the features to deal with mutability are introduced in a specialization of it – *Ordered List of Mutable* – as shown in Fig. 20.

The *Generalization* construct encompasses the semantics of inheritance, with applicability to every non-private model element defined for the superclass. This is valid also for template parameters. This means that every specialization of a template is also a template, and its signature encompasses the signature of its parent. Therefore, although the class *Ordered List of Mutable* in Fig. 20 does not show a dashed rectangle on its top-right corner, it is a template.

To clarify that objects added to *Ordered List* must be immutable, *prop* is defined in *E* as read-only. In the

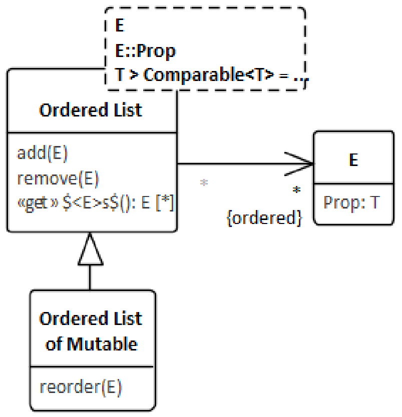


FIGURE 20. Subclassing a template, inheritance of template parameters.

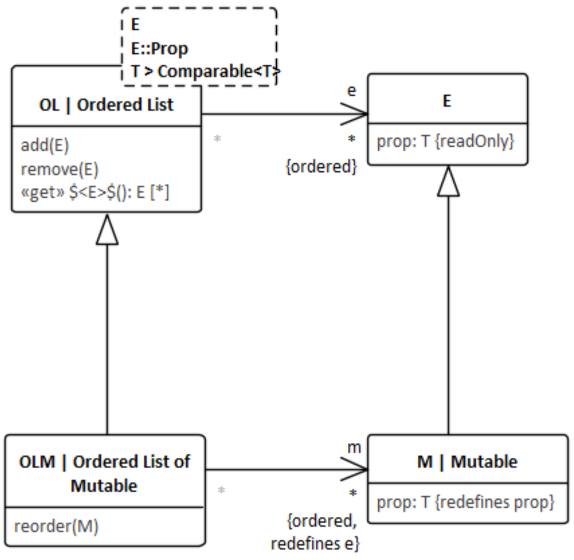


FIGURE 21. Ordered List of mutable, 2nd step towards definition.

specialization of *E*, called *Mutable*, *prop* is redefined as read-write. The Java equivalent of *E* and *Mutable* are shown in Fig. 21: *E* only includes a constructor accepting a value for *prop* and a getter of this property, while a setter is only defined in the *Mutable* subclass. It should then be stressed that *Ordered List of Mutable* is tailored to maintain a list of mutable objects; therefore it connects specifically to *Mutable* objects, not to *E* objects. This is the type of situation that the *redefinition* concept in UML is made for. Thus, the modelling solution is as in Fig. 21. The equivalent of *E* and *Mutable* in a Java setting is as in Fig. 22.

To keep the diagrams more compact and legible, hereafter classes are shown both with names and abbreviated aliases, in the format "*alias* | *name*", and aliases will be used in most textual specifications.

The notification behavior may be obtained by binding *Mutable* to a template encoding the role of *Subject* in the *Observer* design pattern, as shown in Fig. 23.

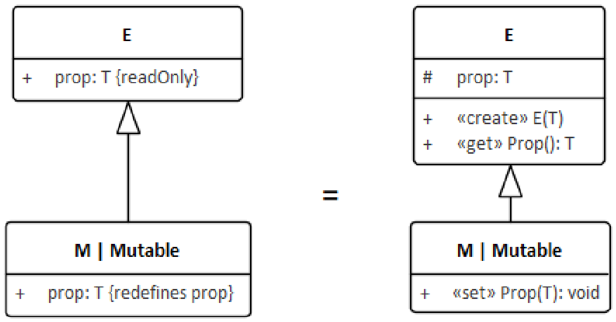


FIGURE 22. Semantics of a read-write property redefining a read-only one.

There are two problems with the modelling of the new template. Specifically in the generalization connecting *OLM* to *OL*. The first is a subclassification problem: An *Ordered List of Mutable (OLM)* object may not be an *Ordered List (OL)* object. If it were, it would be possible to collect *E* objects into *OLM* objects – notice that although it is not possible to execute *myOLM.add (myE)*, it is possible to assign *myOLM* to *myOL* and then execute *myOL.add (myE))*. Since *E* objects are not furnished with the required notifying behaviour, that would break the semantics of *OLM*. Specifically, the problem is that *OLM* inherits *add (E)*, while it shouldn't. The second problem has to do with the definition of *OLM* as a template. Being a subclass of *OL*, *OLM* inherits parameter *E*, while the class that should be exposed as a parameter of this template is *Mutable (M)* instead.

Therefore, the generalization between *OLM* and *OL* does not apply. Nevertheless, OL's features and semantics unequivocally apply to *OLM* if class *E* is replaced by *M*. This suggests that the relation between *OLM* and *OL* should be a *binding*, not a generalization. The binding shown in Fig. 24 makes it possible to derive the desired features from *Ordered List*'s into OLM and, at the same time, solve the problems above. An *OLM* object is no longer an *OL* object, and *OLM* 's parameters are *M* and *prop*. It should be noted that parameter *E* will not be part of *OLM*'s signature because it was substituted by the binding to *OL*. On the contrary, t-parameter *E::prop* was not substituted; therefore, OLM inherited it.

Therefore, this example shows that the Binding relationship is a construct for reuse and extensibility, similarly to the Generalization. But one should be used when the semantics of inheritance and inclusion are not desirable.

Although closer to a final solution, the new template is not modelled correctly. The problem that persists might be evident in Fig. 25, where the expanded view of the two bindings (to Ordered List, in Fig. 23, and to Subject, in Fig. 24) are shown jointly.

The problem is that there is no guarantee that properties *OLM::m* and *M::observers* are always consistent with each other, as they should be. Since those properties are independent of each other (in Fig. 25), elements may be added

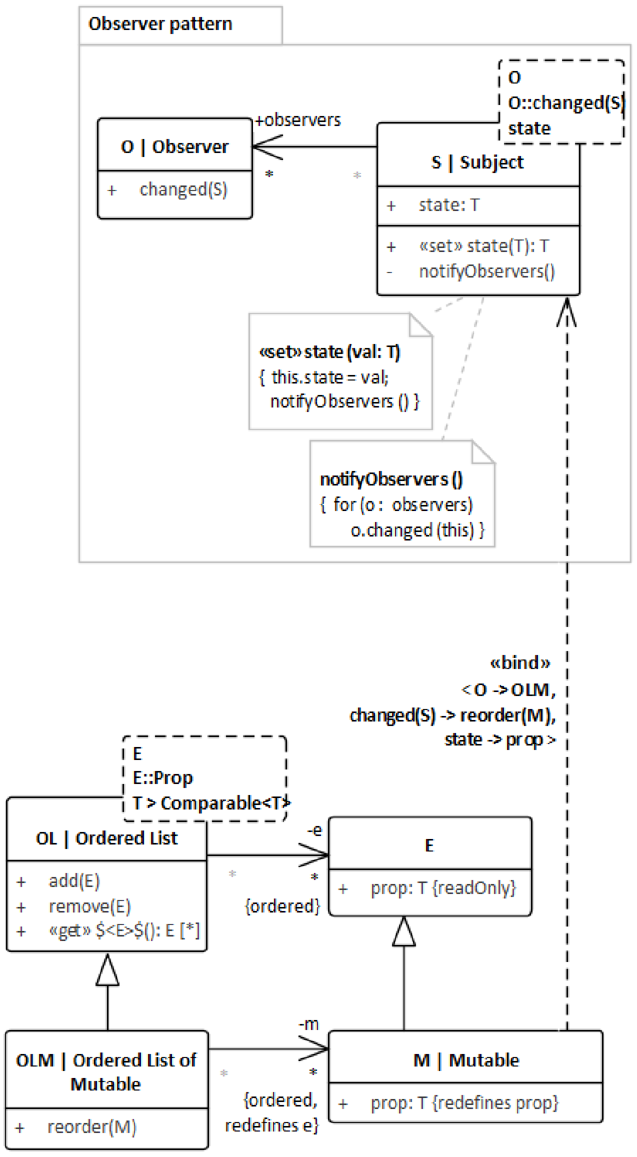
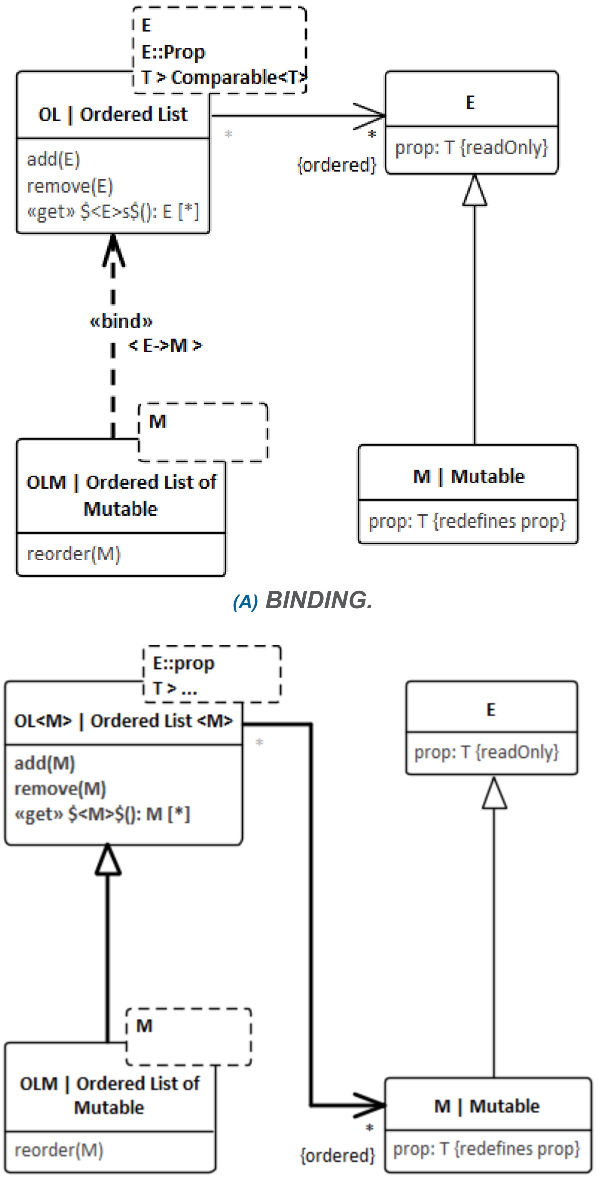


FIGURE 23. Binding to *Subject* of the *Observer* pattern.



(B) EXPANDED VIEW OF THE BINDING.

FIGURE 24. Binding instead of generalization.

to OLM::m without having these elements' *M::observers* property updated with a new observer and vice versa. In UML, the construct to impose consistency between properties is the Association. Defining two properties as ends of the same association makes them inverse of each other. Therefore, consistency between *OLM::m* and *M::observers* can be achieved by introducing an association between OLM and M and declaring this association's ends as redefinitions of *OL::e* and *S::observers*, as in Fig. 26.

As it might be clear by now, class templates are not always glued as easily as expected. Consequently, class templates are generally advisable if the solution being modeled adequately fits within the boundaries of a single class. A package template is usually better suited if it spans more than one. Since a package is a container for several classes, when a package template is instantiated, several classes are reproduced and their interrelationships preserved, exempting the modeler from the burden of glueing the parts together. The following section illustrates how a package template is preferable to its emulation with a set of class templates.

I. PACKAGE TEMPLATES

If two or more interrelated classifiers model a solution intended to be generic, then it is better defined as a package template.

There are two important aspects to highlight concerning package templates: (1) the benefits of package templates compared to agglomerates of class templates; and

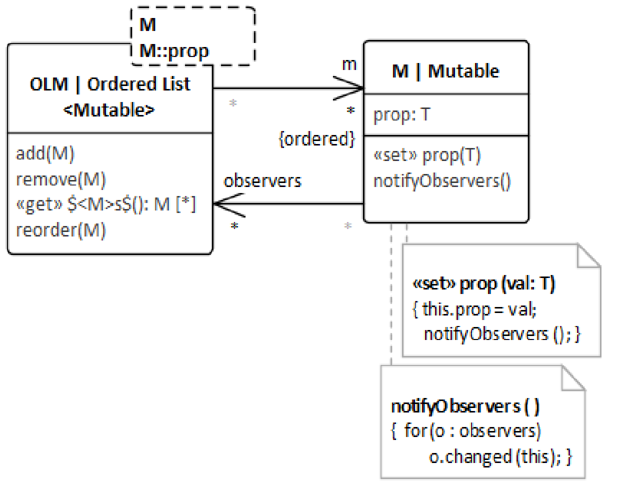


FIGURE 25. Join view of the expansions of the bindings to *OL* and *Subject*.

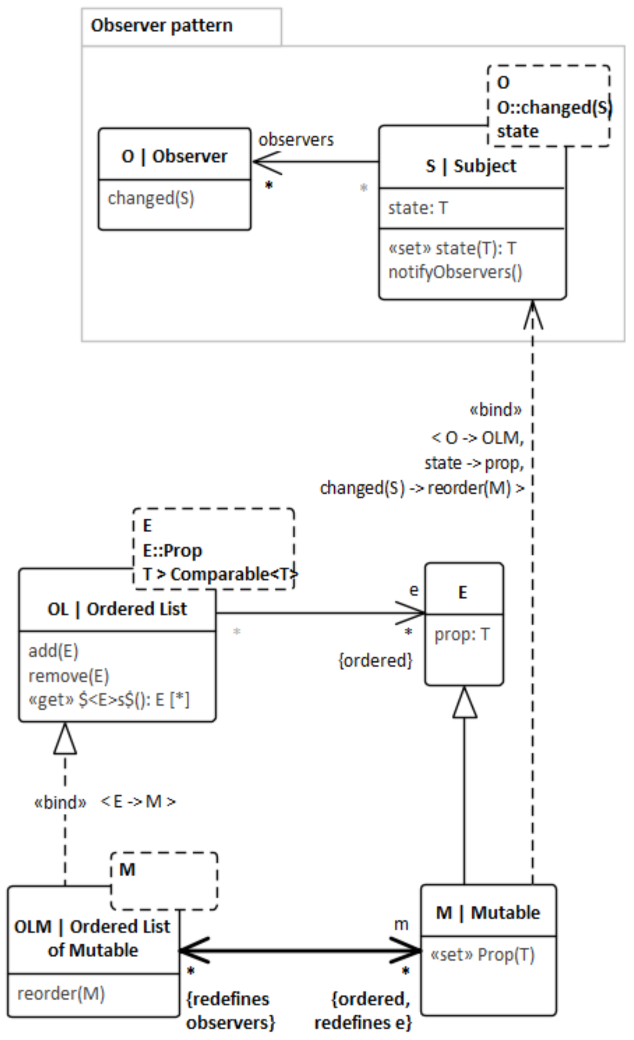


FIGURE 26. Complete modelling of an ordered list of mutable objects.

(2) two different strategies for parameterizing a package template, corresponding to two different results when instantiating it.

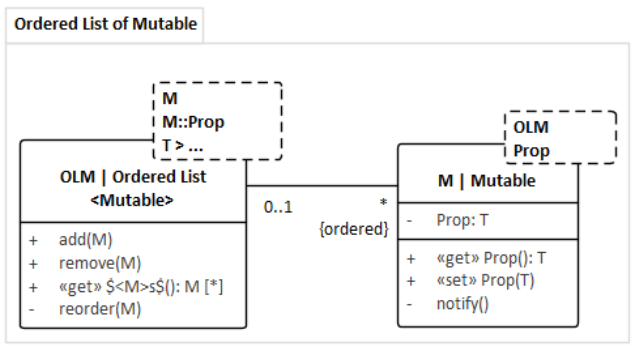


FIGURE 27. Expanded view of the package *Ordered List of Mutable* (first attempt).

1) PACKAGE TEMPLATES VS. CLASSIFIER TEMPLATES

A question that may be put when considering package templates is whether these are useful or if the job could be done simply by resorting to classifier templates. After all, if most common programming languages do not provide package templates, why does UML provide them? The answer is that UML provides modelling constructs with more semantics than those commonly offered by programming languages, and those constructs do not scale well with genericity if only classifier templates are available. For instance, one such construct is the *Association*, intended to give consistency between two or more opposite properties. It also elevates the concept of object connectivity to a first-class construct, by allowing it to be instantiated and specialized. By contrast, rich semantics must be assembled from smaller, less abstract constructs with the common programming languages. These, although less expressive, scale well with genericity merely supported by classifier templates. In this section, the Association will exemplify that the same does not happen in UML.

When binding to a package template, the contents of that package are cohesively reproduced inside the bound package. This reproduction as a whole brings two benefits: (a) increased expressivity because several elements are reproduced at once (with a single binding); and (b) consistency assurance between the template's internal structure and the bound package's.

The previous section shows that combining two unrelated class templates into a body with integrated semantics is not straightforward. In the current section, the challenge is taken one step forward to show that even if the classifier templates are already interconnected, the transposition of that connection onto the bound classifiers still requires some modelling effort and redundancy.

The package *Ordered List of Mutable* shown previously in Fig. 26 attempted to emulate a package template with a set of classifier templates. To make it clear, the expanded, flattened view of that package is shown in Fig. 27.

However, this emulation is not complete because one more classifier in the *Ordered List of Mutable* package needs to

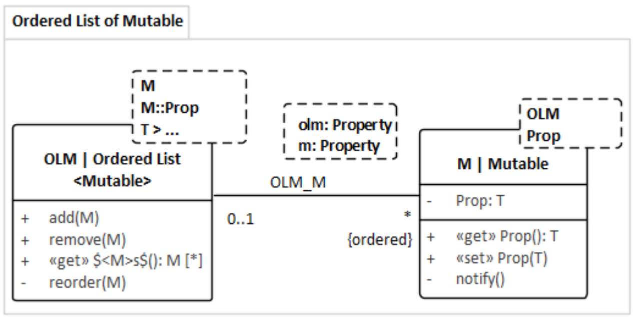


FIGURE 28. Package fully emulated by classifier templates (second attempt).

be declared as a template: the association connecting *OLM* to *M*. If not, the bindings to *OLM* and *M* will not generate an association between the corresponding bound classes. The problem shown in Fig. 25 (independent properties) occurs again. To compensate for that, the complete emulation must be as in Fig. 28.

The two template parameters of the association – *olm* and *m* – are meant to be substituted by the properties generated in the classes binding to *M* and *OLM*, respectively, to have the bound classes and association properly integrated. Otherwise, the association template will introduce two more properties between the bound classes and those already existing. Instantiating the classifier templates above would be as shown in Fig. 29, where the ranking of players in games is modeled (*points* are plausibly a mutable property while a game is going on). The expanded view of the bound classifiers is also shown in that figure.

The redundancy of specifications is obvious. To tie things together, most of the related elements are pretty much related twice: *Player* binds to *M* and *Player* also substitutes *M* (in the binding on the left); the same between *Game* and *OLM*; and *points* substitutes *Prop* in two bindings.

Additionally, such redundancy allows for inconsistencies. For instance, the bindings in Fig. 30 are accepted by UML but do not reproduce on *Game* and *Player;* the cohesion exists among the class templates, because *Prop* is not being substituted by the same property on both bindings, as it should. These are problems that do not occur when using package templates. Modelling the above solution as a package template and binding to it is as shown in Fig. 31. It can be noticed in the expanded view that the cohesion between the classes is intrinsically reproduced in the bound template. It is also noticeable that it is achieved through a more concise and clear specification. Package templates are more expressive, safe, and clear than combinations of classifier templates when the modelling solution spans more than one classifier.

2) PARAMETERED ELEMENTS VS. PARAMETERED NAMES

Regarding the second aspect to be highlighted about package templates, it should be noted that the first two template

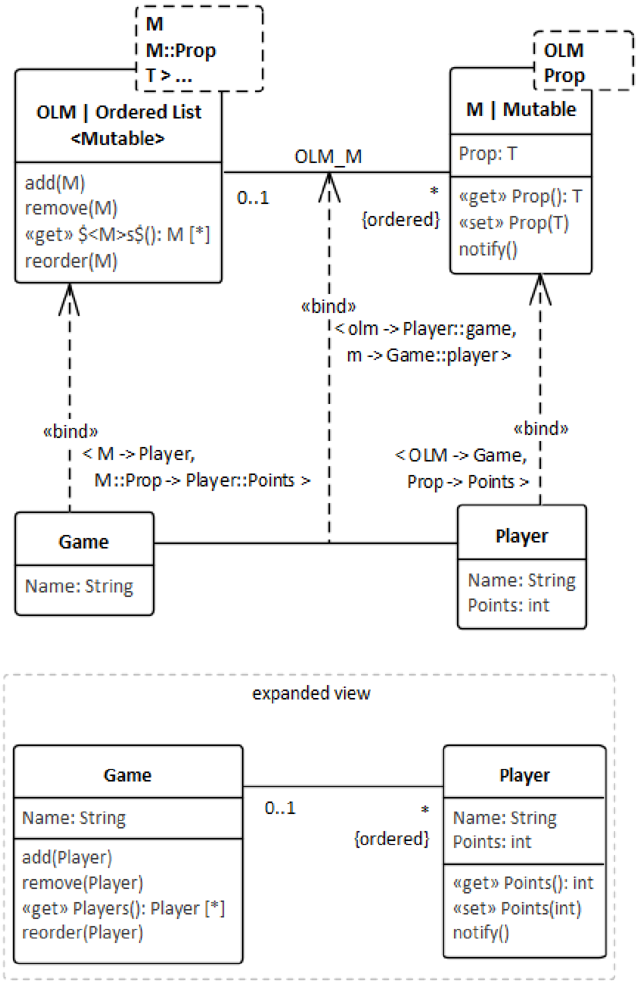


FIGURE 29. Binding to classifier templates to emulate a package template.

parameters in Fig. 31 are string expressions, being referenced through "$< . . . >$", which are used to name the classes inside the package. In other words, the names of the *OLM* and *M* classes are being exposed as parameters, not the classes themselves. This is necessary to have the contents of those classes transposed into the instances of the package template [19, Sec. 7.4.5.1 and 12.2.3.2], as will be explained next.

While defining the signature of a package template, each classifier inside the package may be considered with three possibilities: expose it as a parameter; expose its name as a parameter; and expose neither of them. The choice among these possibilities determines how the classifier will be reproduced inside the template instance.

In the first case, when binding to the template, substituting the parameter causes the classifier to replace its substitute. Thus, the classifier will not be transposed into the bound package. Considering the example above, if classes *OLM* and *M* were exposed as parameters, only the association would be transposed into the bound package (see Fig. 32). This type of parameterization should be used only if the purpose is merely

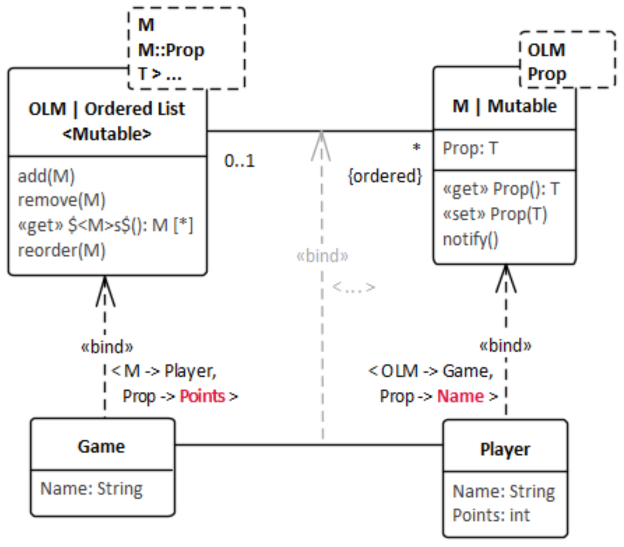


FIGURE 30. A viable set of bindings that do not preserve the joint semantics of the class templates.

to introduce new classifiers in the bound package and leave the existing ones unchanged.

If the contents of a classifier are intended to be transposed, then that classifier's name should be exposed as a parameter. As mentioned previously, the semantics of the *Binding* relationship states that the instantiation of a template is a two-step process: *expansion* and *merging* [19, Sec. 7.3.3.3]. The first step, *expansion,* produces an anonymous template copy with every parametered element replaced by its substitute. For the examples above, expansion produces the class *Ordered List <Person>* and the package *Ordered List of Mutable <"Game", "Player", Player::points>*.

The second step, *merging*, consists of combining the anonymous element's body with the bound element's body to get this one's actual specification.

The merging is specific to the kind of template. For classifier templates, merging is achieved through inheritance, as seen in Fig. 7 and Fig. 8. For package templates, the merging is performed using the UML's *Package Merge* construct, which is illustrated in Fig. 33. This construct's semantics makes the contents of the *Merged Package* to be copied into the *Receiving Package* and combined (merged) with this one's, based on the equality of the elements' names and signatures. In other words, every element is combined with its homonymous or simply copied if no homonymous exists. That's to say that the contents of a class named *C* in the merged package (its features, inner classifiers, etc.) are copied into a class also named *C* in the receiving package, and, recursively, every nested element is merged as well. The rules regarding homonyms and how duplicates are resolved are presented in [19, Sec. 12.2.3.2].

The operationalization of a package binding using a template merging explains why *OLM* and *M* are typed as *String Expression*. If these parameters are substituted by

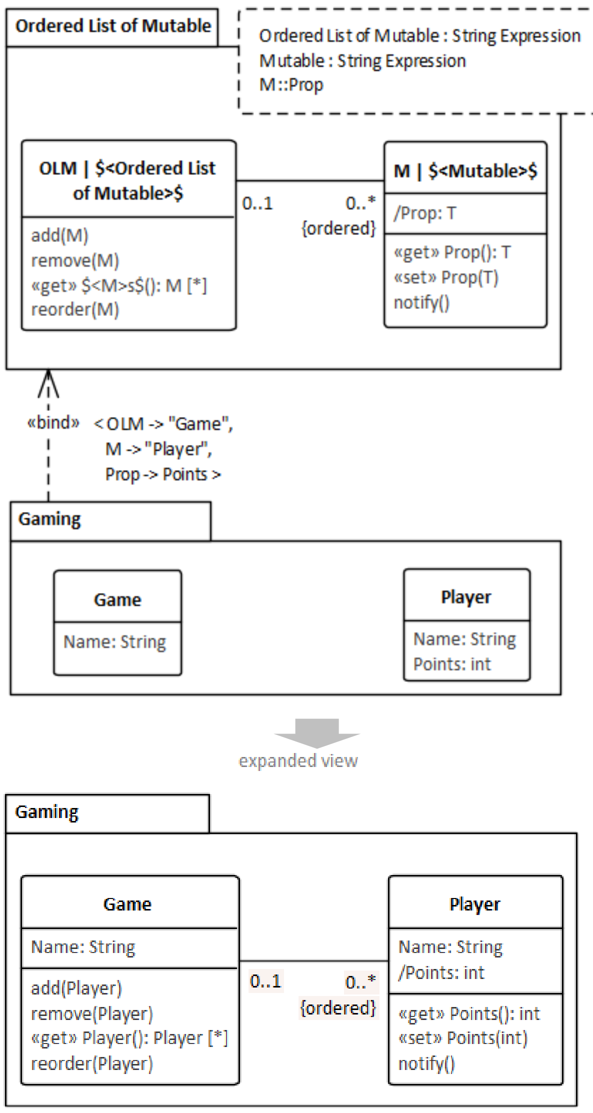


FIGURE 31. Binding to a package template.

names of classes already existing in the bound package, these classes are merged with the classes in the template. As seen in Fig. 31, the contents of *OLM* and *M* are effectively reproduced inside *Game* and *Player*.

IV. RELATED WORK

This section lays out the state-of-the-art that motivated this paper and presents the current situation of the UML templates adoption based on the following references: standard documentation, popular books, research papers, and modelling tools. It also discusses related concepts and techniques, such as UML profiles and patterns.

A. UML SPECIFICATION

The formal writing style of the UML 2.5.1 specification [19] is tough to read when it comes to the text of templates. This is due to two reasons: first, the need to be technically correct

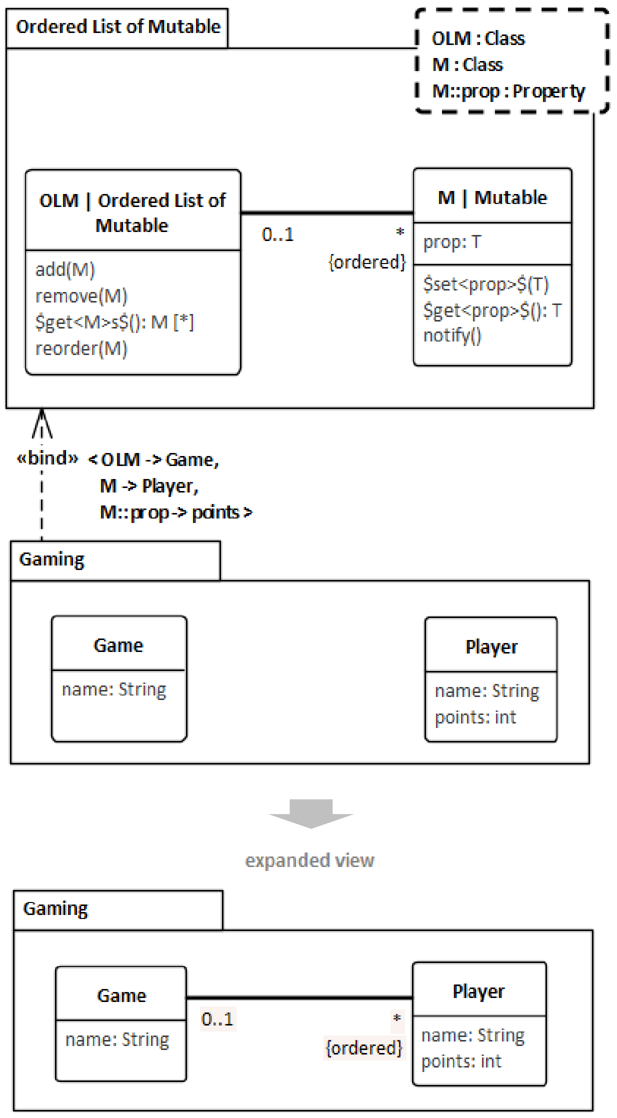


FIGURE 32. The effect of substituting classifiers in a binding to a package template.

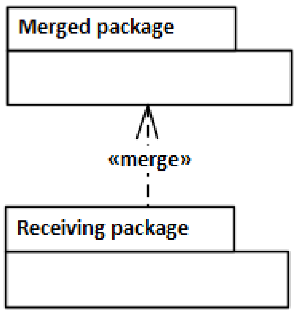


FIGURE 33. The *Package Merge* construct, graphical notation.

leads to a very verbose discourse, overloaded with long terms, such as ''classifier template parameter''; second, the information on templates is scattered over several sections, as many as the number of element types (class, property, package, etc.) that can be templated plus one section dedicated to what is common to all types of templates. This makes it hard to get the whole picture regarding templates. For instance, the merge semantics for package binding is only perceived if the reader is doing a thorough reading and notices that in the example provided, it is the name of classes exposed as parameters, not the classes themselves.

B. POPULAR TECHNICAL BOOKS

Even though it is a seminal book on UML, *The UML User Guide* [21] only dedicates a 1-page to briefly introduce templates: using a single example of a class template that shows the graphical notation for the signature, binding relationship, and actual arguments. The *UML Reference Manual* [22] is the book with the lengthiest sections on templates. Even so, it only introduces class and interface templates. Larman's *Applying UML and Patterns (3rd ed.)* [23] dedicates less than a 1-page to templates.

Other popular UML books – such as *UML Distilled* (3rd ed.) [24], *The Object Primer* (3rd ed.) [25], *UML in Practice* [26], *UML 2, and the Unified Process* (2nd ed.) [27] – do not mention templates at all.

C. RESEARCH PAPERS

Research aiming at strengthening or clarifying UML templates has been scarce. Also, the pieces of work that can be found do not form a cohesive body; most of them address a specific and distinct issue, but not the whole picture as we propose in this paper. To our knowledge, the publications that have addressed this topic are the following:

Caron and Carré [28] aim at strengthening the verification rules for template instantiation, meaning that it provides an inceptive step towards a type system for UML templates.

Cuccuru *et al.* [29] put forth semantics for a *substitutable classifier,* a feature of template parameters that happens to be presented in a scattered way by the UML standard documentation, leading to the perception that it lacks a definition. Although the solution provided unquestionably embodies an interpretation of a substitutable classifier, the concept (once deciphered) has a broader meaning.

Vanwormhoudt *et al.* [30] propose that a template signature could be a model, not a listing of parameters. The idea is that such a model should be mimicked by the template's arguments on instantiation. This is another initial approach towards a type system for UML templates. In another paper [43], the idea is brought into an MDE (model-driven engineering) context, where a set of operators are elicited and considered pertinent to manipulating templates.

Abbas *et al.* [31] propose a bridge from UML class templates annotated with OCL constraints to a formal language to grant proper support to a part of the UML language.

Farinha and Ramos [32], [33] propose another approach towards a type system, however considering a simplified scope for UML: that the language is being used just to model the constructs of the common OOPLs, leaving out high-level constructs and UML-specific features such as multiplicities.

Farinha [34] proposes an extension to support cross-metaclass substitutions, i.e., that in a parameter substitution the substituted and the substituting elements could have different but convertible kinds. For instance, that behaviour could substitute an operation.

Most of these papers focus exclusively on class templates, and the majority only present simple examples. Furthermore, none of these papers provides a broader overview of UML templates as discussed in this paper.

D. MODELLING TOOLS

To assess the support of UML templates by modelling tools some of the most popular ones were analyzed, namely: Eclipse Papyrus [35], Enterprise Architect [14], MagicDraw [15], PowerDesigner [16], StarUML [17], and Visual Paradigm [18].

This analysis verifies that most of these tools only support the basic UML template's features, namely those in popular OOPLs. This means that current modelling tools do not offer a significant share of the spectrum of templateable and parameterable element kinds. And those that support UML-specific genericity (notably, Eclipse Papyrus) do not provide a semantic-oriented editing style for it, for example: if a template parameter is declared with the Package type, the tool does not check whether there is a package with the supplied name in the model, nor it does present the user with a list of packages to choose from when the parameter is to be substituted.

Especially notable is the generalized absence of the possibility to expand template instances. This means that none of the referred tools allows users to get a beer from a *SixPack* object if *SixPack* is defined as *Set <Beer, 6>*.

E. RELATED CONCEPTS

Other concepts can be related to UML templates that deserve to be mentioned: UML profiles and patterns.

UML profiles provide a lightweight extension mechanism for creating or customizing UML-based languages for multiple domains (e.g., business process modelling, service-oriented architecture, interactive applications) or specific platforms (e.g., Java Platform,.NET Framework) [19]. Profiles are defined with custom stereotypes, tagged values, and constraints that are applied to specific model elements, like Class, Attribute, Operation, and Use Case. However, the profile mechanism is not a first-class extension mechanism because it does not allow modification of existing metamodels or to create a new metamodel (as, for example, MOF language does). Profile only allows adaptation or customization of an existing metamodel with constructs specific to that domain or platform. For instance, it is impossible to eliminate constraints that apply to a metamodel; it is only possible to add new constraints to a profile. UML profile is an extension mechanism at the metamodel level, while UML templates provide extension and flexibility mechanisms at the model level. Consequently, because the UML template is a UML first-class construct, it shall be possible to create profiles with template stereotypes.

Patterns are a systematic way to capture the experience of experts about good or best practices and document these pieces of wisdom in an accessible way for reuse. Patterns are appreciated by academics and practitioners alike because they describe and reason about good designs in a way that makes it possible for others to understand and reuse them. Patterns are also rules that express a relation between a specific environment (context), a problem (conflict of forces), and a solution (resolution of the conflict) [36].

Many *design and architectural patterns* have been proposed for recurrent software and system engineering problems [6], [37]. Their solutions are usually represented with UML class, object, and interaction diagrams, sometimes exemplified by concrete arrangements of objects. Some researchers have discussed techniques to represent these patterns directly using UML or using UML extensions for patterns (like UML profiles) [38], [39], and few directly with UML templates [40], [41]. For instance, Sunyé discussed the representation of patterns with a former version of UML in which templates were named "parameterized collaborations" [40] and Vanwormhoudt *et al.* with the current version of UML [30], [42]. Vanwormhoudt *et al.* show how to use UML templates to represent pattern- and aspect-oriented modelling focused on aspectual templates. This concept requires template parameters to form a model of systems into which new functionalities are to be injected [30].

V. CONCLUSION

The complete understanding of UML templates has been neglected and poorly applied or supported by modelling practitioners, academics, and even by tool developers. This paper indicates that this situation results from the fact UML templates are very flexible but also hard to learn and to use in practice; from the general misunderstanding that UML templates are just graphical representations of genericity like it is found in programming languages, and from the insufficient support of modelling tools.

Indeed, the capabilities and potential of UML templates are far-reaching. As such, increasing awareness around them could bring considerable benefits for UML users, namely in what concerns model reuse, to increase the productivity of system design and improve model quality via early checking, by the reuse of proved models [42]. Despite these benefits, we are aware that the use of UML templates introduces a higher level of abstraction and complexity and, therefore, challenges related to learnability and understandability, which may inhibit or prevent beginner modelers from using them in practice, without proper training.

This paper provides a comprehensive overview of UML templates following a pedagogic style and is supported by several illustrative examples. It introduces the core and basic concepts of templates, but many other relevant aspects that are not commonly discussed in the literature, such as: how to constrain classifiers, how to expose operations as template parameters, how to expose properties as template parameters, how to use operation templates, how to use parameter defaults, the relationship between binding and generalization and, finally, how to use package templates.

With this contribution, we hope to motivate tool builders towards providing better and practical support to UML templates, researchers to enhance and evaluate their usage, and practitioners to produce and use them in real scenarios.

Tool builders could put their efforts into providing full support to the semantics of the UML templates, something that is essential to their practical use. Without that, a template instance does not embody the corresponding template definition. Therefore, it is a black box that only becomes fully defined at the target language level. None of the existing UML modelling tools provides a complete implementation of template semantics.

Researchers have several issues and enhancements to improve this current situation. The main shortcoming of UML templates is the thin verification mechanism that the language establishes for template instantiation. Regarding this, UML is at the same stage as C++ was before the introduction of Concepts [43]. Although with UML's verification rules an incorrect instantiation will not make it, the diagnosis of the problem will not be adequate, as it requires that the template user know its internals. This was the same problem as C++. UML lacks a concept system, but, unfortunately, it cannot be promptly borrowed from C++ or other languages because UML has constructs that do not exist in those languages and require specific verification rules. In addition to providing improved error reporting, a concept system may also be leveraged to automate or semi-automate the elicitation of substitutes for template parameters. That would make the instantiation of templates a quick, smooth, and easy experience.

Additionally, a sound concept system would increase the capabilities of UML templates. For instance, the current version of UML (2.5.1) imposes an excessively stringent restriction to have the constraining classifier mechanism guarantee semantic correctness for template instances: a class being exposed as a parameter must be an empty class [19, Sec. 9.9.5.6, constraint *parametered_element_no_features*]. With this restriction, the class *Item* used in Figs. 4 to 10 could not have the 'Name' attribute, i.e., 'Name' would have to be defined in a superclass of Item, which would unnecessarily increase the complexity of the template. As it shall be shown in a forthcoming paper, it is possible to design a concept system that considers the notion of structural subtyping, which would dispose of the constraining classifier mechanism. With the restriction mentioned above removed, templates would become more straightforward.

Furthermore, researchers can also conduct experiments and empirical studies to understand and evaluate the key benefits and challenges of the UML templates adoption, namely concerning its usability and reusability.

Modelling practitioners could benefit from a more substantial adoption of UML templates based on a solid template construct backed up by proper tools. By using templates, practitioners can move to a higher level of abstraction and model reuse. These pieces of models can become more generic and understandable, a factor known to ease and accelerate model-based engineering.

REFERENCES

[1] G. Reis and J. Järvi. (Oct. 2005). *What is Generic Programming*. [Online]. Available: http://lcsd05.cs.tamu.edu/papers/dos_reis_et_al.pdf

[2] J. Jeuring and P. Jansson, "Polytypic programming," in *Advanced Functional Programming* (Lecture Notes in Computer Science), vol. 1129. Berlin, Germany: Springer, 1996, pp. 68–114.

[3] R. S. Bird, "An introduction to the theory of lists," in *Logic of Programming and Calculi of Discrete Design, NATO ASI*, vol. 36. Berlin, Germany: Springer, 1987, pp. 5–42.

[4] J. Gibbons, "Datatype-generic programming," in *International Spring School on Datatype-Generic Programming* (Lecture Notes in Computer Science), vol. 4719. Berlin, Germany: Springer, 2017, pp. 1–71, doi: 10.1007/978-3-540-76786-2_1.

[5] E. Ernst, "Family polymorphism," in *Proc. Eur. Conf. Object-Oriented Program* (Lecture Notes in Computer Science), vol. 2072. Berlin, Germany: Springer, 2001, pp. 303–326.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley, 1995.

[7] E. W. Axelsen, F. Sárensen, and S. Krogdahl, "A reusable observer pattern implementation using package templates," in *Proc. 8th Workshop Aspects, Compon., Patterns Infrastruct. Softw.*, 2009, pp. 37–42, doi: 10.1145/1509276.1509286.

[8] B. C. D. S. Oliveira, M. Wang, and J. Gibbons, "The visitor pattern as a reusable, generic, type-safe component," *ACM SIGPLAN Not.*, vol. 43, p. 439, 2008, doi: 10.1145/1449955.1449799.

[9] F. Sárensen and S. Krogdahl, "Generic packages with expandable classes compared with similar approaches," in *Proc. Norwegian Inform. Conf.*, Oslo, 2007.

[10] S. Krogdahl, B. Müller-Pedersen, and F. Sárensen, "Exploring the use of package templates for flexible reuse of collections of related classes," *J. Object Technol.*, vol. 8, pp. 59–85, Oct. 2009.

[11] E. W. Axelsen and S. Krogdahl, "Package templates: A definition by semantics-preserving source-to-source transformations to efficient java code," in *Procs. 11th Int. Conf. Generative Program. Compon. Eng.*, 2012, pp. 50–59, doi: 10.1145/2371401.2371409.

[12] E. W. Axelsen and S. Krogdahl, "Groovy package templates: Supporting reuse and runtime adaption of class hierarchies," in *Proc. 5th Symp. Dyn. Lang.*, 2009, pp. 15–26, doi: 10.1145/1640134.1640139.

[13] OMG. (2000). *OMG Unified Modelling Language Specification, Version 1.3*. [Online]. Available: https://www.omg.org/spec/UML/1.3/PDF.

[14] Sparx Systems. *Enterprise Architect's Home Page*. Accessed: Oct. 2021. [Online]. Available: http://www.sparxsystems.com/products/ea/

[15] No Magic. *MagicDraw's Home Page*. Accessed: Oct. 2021. [Online]. Available: http://www.nomagic.com/products/magicdraw

[16] SAP SE. *SAP PowerDesigner's Product Page*. Accessed: Oct. 2021. [Online]. Available: http://www.sap.com/products/powerdesigner-data-modelling-tools.html

[17] MKLabs Co. *StarUML's Product Page*. Accessed: Oct. 2021. [Online]. Available: http://staruml.io/

[18] V. P. International. *Visual Paradigm's Product Page*. Accessed: Oct. 2021. [Online]. Available: http://www.visual-paradigm.com/

[19] OMG. (2017). *OMG Unified Modelling Language, Version 2.5.1*. [Online]. Available: http://www.omg.org/spec/UML/2.5.1

[20] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. (2019). *The Java Language Specification, SE 12*. Accessed: Jul. 2019. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se12/jls12.pdf

[21] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modelling Language User Guide*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2005.

[22] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modelling Language Reference Manual*. Boston, MA, USA: Addison-Wesley, 2005.

[23] C. Larman, *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Iterative Development*, 3rd ed. Pearson, 2004.

[24] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modelling Language*, 3rd ed. Boston, MA, USA: Addison-Wesley, 2003.

[25] S. W. Ambler, *The Object Primer: Agile Modelling-Driven Development with UML 2.0*. Cambridge, U.K.: Cambridge Univ. Press, 2004.
[26] P. Roques, *UML in Practice*, 1st ed. Hoboken, NJ, USA: Wiley, 2006.
[27] J. Arlow and I. Neustadt, *UML 2 and the Unified Process*. Reading, MA, USA: Addison-Wesley, 2005.
[28] O. Caron and B. Carrá, "An OCL formulation of UML2 template binding," *Unified Model. Lang. Model. Lang. Appl.*, vol. 3273, pp. 27–40, Jan. 2004.
[29] A. Cuccuru, A. Radermacher, S. Gárard, and F. Terrier, "Constraining type parameters of UML2 templates with substitutable classifiers," in *Proc. 12th Int. Conf., Modeling*, Denver, CO, USA, vol. 5795, 2009, pp. 644–649, doi: 10.1007/978-3-642-04425-0_51.
[30] G. Vanwormhoudt, O. Caron, and B. Carr, "Aspectual templates in UML," in *Proc. SoSym*, Apr. 2015, pp. 1–15, doi: 10.1007/s10270-015-0463-3.
[31] M. Abbas, C.-B. Ben-Yelles, and R. Rioboo, "Modelling UML state machines with FoCaLiZe," in *Proc. Integr. Formal Methods*, 2014, pp. 87–102, doi: 10.1007/978-3-319-10181-1_6.
[32] J. Farinha and P. Ramos, "Computability assurance for UML template binding," in *Model-Driven Engineering and Software Development*. Berlin, Germany: Springer, 2015, pp. 190–212.
[33] J. Farinha, "A Demonstration of compilability for UML template instances," in *Proc. 4th Int. Conf. Model-Driven Eng. Softw. Develop.*, 2016, pp. 397–404.
[34] J. Farinha, "Extending UML templates towards flexibility," in *Proc. 2nd Workshop Flexible Model-Driven Eng.*, 2016, pp. 32–41. [Online]. Available: http://ceur-ws.org/Vol-1694/FlexMDE2016_paper_1.pdf
[35] Eclipse Foundation. *Papyrus's Home Page*. Accessed: Oct. 2021. [Online]. Available: https://www.eclipse.org/papyrus/
[36] F. Buschmann, H. Kevlin, and C. S. Douglas, *Pattern-Oriented Software Architecture, On Patterns and Pattern Languages*, vol. 5. Hoboken, NJ, USA: Wiley, 2007.
[37] F. Khomh and Y.-G. Gueheneuc, "Design patterns impact on software quality: Where are the theories?" in *Proc. IEEE 25th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Mar. 2018, pp. 1–5.
[38] H. Marouane, C. Duvallet, A. Makni, R. Bouaziz, and B. Sadeg, "An UML profile for representing real-time design patterns," *J. King Saud Univ.-Comput. Inf. Sci.*, vol. 30, no. 4, pp. 478–497, Oct. 2018.
[39] F. Rademacher, S. Sachweh, and A. Zándorf, "Towards a UML profile for domain-driven design of microservice architectures," in *Proc. Int. Conf. Softw. Eng. Formal Methods*. Berlin, Germany: Springer, 2017, pp. 230–245.
[40] G. Suny and A. L Guennec, "Design patterns application in UML," in *Proc. ECOOP Conf.* Berlin, Germany: Springer, 2000, pp. 44–62.
[41] J. Dong, S. Yang, and K. Zhang, "Visualizing design patterns in their applications and compositions," *IEEE Trans. Softw. Eng.*, vol. 33, no. 7, pp. 433–453, Jul. 2007.
[42] G. Vanwormhoudt, M. Allon, O. Caron, and B. Carré, "Template based model engineering in UML," in *Proc. 23rd ACM/IEEE Int. Conf. Modeling Driven Eng. Lang. Syst.*, Oct. 2020, pp. 47–56.
[43] D. Gregor, J. Járvi, J. G. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine, "Concepts: Linguistic support for generic programming in C++," in *Proc. 21st ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang., Appl.*, 2006, pp. 291–310, doi: 10.1145/1167473.1167499.

JOSÉ FARINHA received the degree and M.S. degrees in electrical and computer engineering, with a minor in computer science from the Technical University of Lisbon, in 1990 and 1995, respectively. He is currently pursuing the Ph.D. degree. His professional activity has always been in information technology, with a particular focus on information systems and databases (IS/DB). He worked in industry and consulting for eight years in e-commerce, data quality, and geographic information systems. He took on technical, consulting, and management positions during this period. He was a Lecturer in higher education for 15 years at ISCTE-IUL, Lisbon University Institute, where he taught and supervised academic and business projects in his main specialty field (IS/DB), business intelligence, and management informatics. In this period, he published a few papers in scientific journals and conferences, all in data modeling, most of them on the relation of UML with genericity. He is also an Entrepreneur in information systems applied to the fashion industry. His research interests include UML modeling, genericity, and pattern-based development, particularly interested in their application to IS/DB.

ALBERTO RODRIGUES DA SILVA received the degree in informatics engineering from the New University of Lisbon, in 1989, the M.Sc. degree in electrical and computer engineering from the Technical University of Lisbon, in 1992, the Ph.D. degree in computer science and engineering, in 1999, and the Habilitation degree in computer science and engineering, in 2016.

He is currently an Associate Professor with Habilitation at the Instituto Superior Técnico, Universidade de Lisboa (IST-UL), and a Senior Researcher at INESC-ID Lisboa. He has taught more than 4000 students and supervised more than ten Ph.D. students and more than 70 M.Sc. students. He has authored six technical books and more than 200 publications in journals, conferences, and workshops with peer review and has also been editor of five scientific books. His research interests include information systems, software engineering, model-driven engineering, requirements engineering, document automation, project management, and application in multiple domains.

Dr. Silva is a member of the Portuguese Chartered Engineers Association (OE), Project Management Institute (PMI), and a Senior Member of the Association of Computer Machinery (ACM). He received the following awards and professional certifications JEEP, Scrum Master, PMP, and IST Excellent Teaching.