

On the Key Role of Composition in Object-Oriented Modelling^{*}

Friedrich Steimann, Jens Gößner, Thomas Mück

Universität Hannover, Institut für Informationssysteme
Appelstraße 4, D-30167 Hannover
{steimann, goessner, mueck}@kbs.uni-hannover.de

Abstract. The success of object-oriented software modelling depends to a large extent on the ability to create adequate abstractions. While abstraction itself must remain an intellectual process, a modelling language can support or hinder this process by offering different kinds or dimensions of abstraction. For instance, adhering to the object-oriented paradigm UML incorporates classification and generalization as its key abstraction mechanisms. When it comes to taking the complexity out of real systems, however, we argue that classification and generalization alone are ill-suited to produce abstractions that are both manageable and meaningful. As a remedy, we propose to regard composition as an alternative form of abstraction, and find that it naturally comes with properties that are practically needed. We contrast our view of composition with that of it being a special kind of association, with the composition of deployable elements, and with UML's model management constructs such as packaging.

»The introduction of suitable abstractions is our only mental aid to reduce the appeal to enumeration, to organize and master complexity.« Edsger W. Dijkstra, 1968
[4, p. 2]

1 Introduction

“Among current users of UML, one major complaint is that some of its features do not scale to industrial-size problems. This deficiency requires some adjustment to the existing modeling features; we single out the following two:

- [...] *the ability to recursively decompose objects into structures of interconnected finer-grain objects [...]*
- [...] *the ability to hierarchically compose and combine individual behavior specifications [...].”* [14]

Indeed, it appears that UML has no sufficient means of recursively partitioning large models into smaller ones and, conversely, of composing small units of specification into larger ones. This is surprising, the more so as it is in sharp contrast to certain older modelling languages as, for instance, the SA language of SADT [8], or SDL [7]: both languages allow a system to be recursively decomposed into units of which each one — at a given level of abstraction — completely replaces for the parts of the system it abstracts from.

^{*} in: *UML 2003: Modeling Languages and Applications* Proceedings of the 6th International Conference

It seems that some of the benefits of earlier modelling paradigms, based on the idea of stepwise refinement or functional decomposition, have been forgotten over the excitement about the possibilities of data abstraction and inheritance, perhaps because functional decomposition is nominally tied to procedural programming, the paradigm that was to be displaced by object-orientation. Much to the detriment of object-oriented modelling, however, data models are hard to divide and conquer hierarchically. In fact, even though generalization was initially regarded as the key to mastering the complexity of even huge problem domains, for reasons explained below it never really worked. As a consequence, object-oriented modellers today are left with some primitive model management offerings (such as packaging) whose effect on models does not go beyond that of a pair of scissors on a piece of paper. Relief is desperately needed.

In the following, we argue that composition is a natural form of abstraction possessing properties whose unavailability in UML causes the noted poor scalability of modelling. We then contrast this notion of composition with its variations currently manifested in the UML standard, finding that the latter are either subsumed by the former or out of place all together. To substantiate our argument, we demonstrate how composition as a form of abstraction can tackle some of the scaling problems of UML. A discussion with related work concludes our contribution.

2 Different Forms of Abstraction in Object-Oriented Modelling

Everyone in the modelling community agrees that abstraction is the key to mastering complexity. However, abstraction is a very general concept that has many incarnations. In fact, discovering and precisely capturing the differences between the various forms of abstraction is central to formally defining a modelling language.

Besides its defining characteristic, namely the reduction of the level of detail, abstraction should possess certain technical properties.¹ In particular, abstraction should be

- *systematic*, i.e., there should be rules or guidelines the application of which governs the abstraction;
- *invertible*, i.e., there should be some inverse process (“refinement” or “expansion”) that allows the filling in of information dropped by abstraction;
- *recursive*, i.e., it should not be limited to a single level of abstraction;
- *encapsulating*, i.e., the elements that have been abstracted away should be invisible to the abstraction’s context;
- *structure-preserving*, i.e., an abstraction should be able to take the place of the elements it abstracts from, dragging their structural embedding in the model to itself.

In spite of the technicality of these properties, it should be clear that abstraction is an intellectual process implementing non-trivial modelling decisions, in particular that

¹ Because of their impact on the modelling process, we refer to these as *model abstraction* (as opposed to *domain abstraction*) properties.

despite the property of being systematic, it cannot be automated. Note that being structure-preserving does not imply that abstraction is also type-preserving: the product of abstraction can be of a different type than the elements it abstracts from; in fact, it can abstract from elements of different types.

We can identify at least three different kinds of abstraction that are of use in object-oriented modelling: abstraction through *classification*, abstraction through *generalization*, and abstraction through *composition*. As we will argue, of these the third has the most favourable properties; characteristically enough, it is also treated rather negligently in the current UML standard.

2.1 Classification

A classifier abstracts from a set of instances sharing same structure and behaviour. Classification corresponds to the transition from an element to the set, or class, where the class is something abstract, in particular, something which does not itself exist in the modelled domain. For instance, in a domain with several documents and other things there is no one entity that could be called “Document” (if anything, there might be a prototypical document). In Russell’s type hierarchy, classifiers reside one level above their instances. The inverse of classification, instantiation, reduces the type level.

Classification is recursive, meaning that classifiers can be classified. Because classification invariably raises the type level, classifiers of classifiers are abstracter still; consequently, they too have no representatives in the modelled domain. In object-oriented and data modelling, the classifiers of classifiers are often called meta-classifiers; in fact, classification is the “Metaisierungsprinzip” [17] forming the levels of the ISO IRDF [5] and the OMG’s MOF [11]. However, the use of recursive classification for domain-specific abstraction is quite limited; rather, leading from modelling to meta-modelling it is sometimes used for language specification.

In object-oriented programming terms, a class is considered a unit of encapsulation, since it separates the interface from the implementation. In object-oriented modelling, however, the encapsulation property of classification means something different: for a classifier to encapsulate, it must hide the instances it abstracts from in the context in which it occurs. While this is characteristically the case for diagrams on the specification level, encapsulating the instances of a single classifier (as seen on a combined object/class diagram) cannot be structure-preserving, since dragging the links to other instances to the class would make the links span two type levels (from the instance to the type level), significantly changing their meaning. However, unlike generalization and composition, classification is usually not applied individually: in fact, in presence of strict levelling classification cannot be performed on a single model element — rather, it must be applied to all elements of a diagram. In this case, the encapsulation property of classifiers is granted as above; in addition, if the parallel classification of the structural embedding (e.g., links to associations) is accepted as leaving the meaning of the structure unaltered, then this form of abstraction is also structure-preserving, although not in the strictest sense. We will return to this observation below.

Figure 1 indicates the special role of classification as an abstraction mechanism: while generalization and composition are obviously infinitely recursive, recursiveness of classification would require that `Instance` and `Classifier` were comprised under a single concept. However, the distinction between a class and its elements is presumably the single most important achievement of modern logic, and giving it up in modelling would set back the discipline to the dark ages of semantic nets.

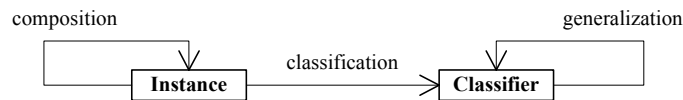


Figure 1. Different forms of abstraction

2.2 Generalization

Of the different forms of abstraction, generalization is conceivably the most prominent in the object-oriented community, the more so since its inverse, specialization, goes hand in hand with inheritance, one of the defining characteristics of object-oriented technology. At first glance, generalization seems to be a powerful concept: ever since Aristotle's introduction of *genus et differentiae* as a conceptual framework for structuring the universe, people have come up with myriads of taxonomical orderings that have served their purpose well.

Although generalization plays an important part in the structuring of modelling domains, it causes some nontrivial problems, too. In fact, since the specializations of a model only exist because the differences they introduce (the *differentiae*) do matter, the general is rarely a sufficient representative of its specializations. In a class diagram, this usually materializes in associations ending at specialized classes, expressing that none of their siblings can play the required role, even though they are comprised under the same abstraction (the generalization). When hiding the classes subsumed by the generalization from a diagram (exploiting the abstraction for model management; see below), these associations cannot be faithfully redirected to the generalization (since it does not possess the required properties); they must be dropped instead. Generalization, therefore, cannot be encapsulating and structure-preserving at the same time.²

It appears that the prominence of generalization is the true crux of object-oriented modelling: while being a defining characteristic, it promises what it cannot deliver, namely to serve as the primary mechanism for the mastering of complexity. Indeed, while generalization helps structure a model and its diagrams, its use for simplification through the omission of detail comes at a high cost. Hence, as a form of model abstraction (in the sense of Footnote 1), generalization is largely a letdown.

² The problem can be boiled down to the simple observation that class hierarchies can generally not be encapsulated by a single interface; instead, they typically define a hierarchy of interfaces (one for each class). In fact, unless the subclasses are completely hidden behind the root of the hierarchy (the Family pattern [16]), class hierarchies are poor abstractions.

2.3 Composition

Composition is something very natural: according to most contemporary explanations of the universe, all things are composed of smaller things, and this recursively down to the composition of atoms (or whatever the indivisible particles of the world are). Despite being natural, composition is a form of abstraction: the composite represents its components in sufficient detail in all contexts in which the fact of being composed does not matter. It is a question of philosophical debate whether a composite is an object in the same right as its atomic components³; however, since we treat composition as a form of abstraction [10], we prefer to regard composites as *abstract instances* (see below).

Composition is a relationship with two roles, the *component* and the *composite*. As suggested by Figure 2 a), it is both recursive and invertible. Note that the composition relationship is not defined using UML's composition symbol; this is intentional, firstly because the diagram belongs to the metalevel, and secondly because the composite role does not consist of component roles. In fact, Figure 2 b) would introduce composition to the UML specification analogously to how generalization is introduced [11].

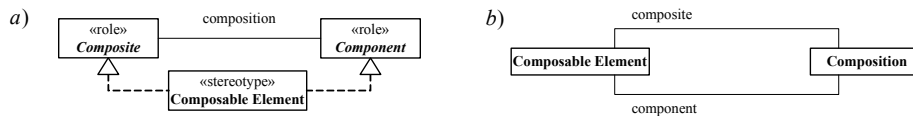


Figure 2. a) Pattern describing composition as a model abstraction and b) its transcription to UML's metamodel (see text).

As for classification and generalization, composition should not be bound to a particular kind of composable element. In fact, composition unfolds its true power only if it composes whole structures (involving model elements of different types) into composites. However, since UML (and object-oriented modelling in general) has no one notion of composition as a form of model abstraction (especially if compared to the rather settled notions of classification and generalization), we cannot assert or deny here the properties of encapsulation and structure-preservation as done above. We can observe, though, that they are naturally present in the world of hardware, a field that is notoriously envied by us software people for its successful record of composition; in Section 4, we will propose a notion of composition for UML that has very similar properties.

In hardware, components are routinely composed into larger units, the composites, which combine their components' functionality into something useful. The combination is also moulded in hardware: the circuit board for instance and the wires and connectors of an electronic device are, strictly seen, also components, although they are

³ For example, any particular document abstracts from the concrete signs it is made up of. Yet, the document is somewhat more than its parts; therefore, it would be considered an instance in its own right [8].

sometimes only referred to as the “glue” of a composition. Even though the hardware composite is usually identified as a distinct entity, when using a composite one always accesses one of its “public” components, the entity itself thus being inaccessible (and, from a practical standpoint, non-existent). From an abstraction point of view, however, the “published” properties of the components are regarded as the properties of the composite as an individual, but abstract entity; this explains why we are led to think of composites as abstract instances.

As can easily be verified, this kind of composition possesses the favourable property of being both encapsulating and structure-preserving, and this without raising the type level: the abstract (i.e., composite) instance combines all the context’s needed properties of the structure it abstracts from. It can therefore completely replace for this structure and drag all structural links to it to itself. Upon expansion (or refinement), the abstract instance vanishes, letting the elements it abstracts from take its place.

2.4 Comparing the Different Forms of Abstraction

The identified properties of the different forms of abstraction in object-oriented modelling are summarized in Table 1. Not incidentally, composition appears as the most complete form of abstraction; as will be detailed in the following sections, we suppose that the absence of a corresponding abstraction mechanism in UML accounts for much of the decried lack of scalability.

Table 1: The properties of the three different forms of abstraction

PROPERTY	CLASSIFICATION	GENERALIZATION	COMPOSITION
systematic	yes	yes	yes
invertible	yes	yes	yes
recursive	no*	yes	yes
encapsulating	yes**	mutually	yes
structure-preserving	yes**	exclusive***	yes
domain abstraction	weak	strong	strong
model abstraction	strong	weak	strong

* except for metamodeling

** if context is also subject to classification

*** unless in very special cases (see text)

Based on their different technical properties, the different forms of abstraction have different uses in modelling. Classification or, more characteristically, its converse instantiation serves the purpose of *model integration*: for example, it shows how an object diagram or a collaboration diagram on the instance level corresponds to the static structure of a system as declared in a class diagram (the model abstraction). Except perhaps for very special (meta-)modelling issues, instantiation does not capture domain information (beyond the trivial fact that the objects of a domain are grouped into certain classes, the domain abstraction).

Generalization on the other hand primarily helps *structure the modelled domain*: it is a means of expression given to the hands of the modeller, who can organize his classifiers in an abstraction (the generalization) hierarchy that reflects subsumption relationships of the problem domain (a domain abstraction). At the same time it serves as a basis of model integration, by inheriting the properties of a classifier shown in one diagram to its specializations shown in another. As has been argued above, however, its use as a model abstraction is rather limited.

Like generalization, the role of composition is twofold: it serves to capture domain abstractions (which elements of a model are meaningfully assembled into larger units), and it is a means of model integration (components in one diagram may be composites in another). Because of its combined capability of producing encapsulating and structure-preserving abstractions, however, its model abstraction capabilities go beyond that of generalization: as will be detailed below, it can serve *model management*.

Figure 1 raises the question whether generalization is a meaningful concept for instances, and whether composition can be applied to classes. Quite obviously, the generalization of instances could be introduced as the basis for instance-based inheritance (of prototype-based languages), even though the general instances are not necessarily abstractions. The second question is trickier: if classes were composed into a composite class, the composite would have to be an abstraction (but not a generalization!) of the composed classes, ideally one that is both encapsulating and structure-preserving. Although inner classes appear to be a possible interpretation, the consequences of such a view, especially as concerns the transfer of the composition relationship to the instances, need to be carefully investigated.⁴

3 Composition in UML

3.1 Composition as a Special Form of Aggregation

In UML, composition is a special form of aggregation, which in turn is a special form of association. While aggregation is a close to meaningless concept in UML (“The distinction between aggregation and association is often a matter of taste rather than a difference in semantics. [...] Think of it as a modelling placebo.” [13 p. 148]), composition has a much stronger semantics. Unfortunately, this semantics is owed to equating composition with joint memory allocation, and to claiming that for a class to have composition associations is equivalent to it having attributes [11, p. 2-66].

⁴ It is important in this context that declaration of composition and composition itself are not confused: although the composability of instances of certain classifiers into composite instances of another classifier can be declared on the type level, the actual composition occurs at the instance level. If composition of classes (as a form of abstraction) were to be declared, this would have to take place on the metalevel.

Whereas issues of memory management are clearly outside the scope of modelling, equating composition with having attributes degrades the components of an object to its properties. It seems that UML has no useful conceptual interpretation for composition; in fact, it does not even promote it as a domain abstraction. Its suitability as a model abstraction (usable for model management) is not conceivable from the standard; all in all, composition is not put forward as a means for the reduction of detail.

Composite Objects. UML defines a *composite object* as a notational shortcut for an object and its attribute objects (or an object and its parts connected to it by composition links) [11, p. 3-67]. Resembling a memory contour, this notation is suggestive of the memory allocation semantics of UML's composition definition; however, it hints at how composition might be used to hide away detail, albeit only for objects.

3.2 Components and the Component Diagram

In UML, “a component represents a physical piece of implementation of a system, including software code (source, binary, or executable) or equivalents, such as scripts or command files.” [11, p. 216]. In particular, components “package the code that implements the functionality of a system” [11] (p.216), but “a component does not have its own features (for example, attributes, operations), it acts as a container for other classifiers that are defined with features” [11] (3-171). Even though components are abstractions in the sense that they are model artefacts representing physical (i.e., real) entities, they are neither domain nor model abstractions; in fact, they suffer from the same shortcomings as packages (see below).

The home of components in UML is the component diagram, in which components can be placed in a context: “A component diagram is a graph of components connected by dependency relationships. Components may also be connected to components by physical containment representing composition relationships” [11 p. 3-170].

Although it is possible to express the composition of components, the composite (the outer component) is merely a container and cannot produce instances that abstract from the components and drag the relations to them (from the outside) to themselves. Instead, UML allows it that a classifier representing the externally visible logic and functionality associated with the component (also called the “dominant class” [11 p. 219-220]) can be introduced and marked with the *focus* stereotype. This corresponds to an application of the Facade pattern and hints on the closeness of component diagrams to the implementation level; in fact, it appears to be a technical workaround bringing component oriented semantics into a class oriented world, and is not compatible with our view of composition as a model abstraction.

3.3 Model Management

UML comes with at least two other forms of recursive (de)composition: the division of a model into packages, and the division of a system into subsystems. Both mecha-

nisms are intended to help manage the complexity of a model, by allowing the recursive grouping of model elements.

A UML package comprises a loose collection of other (subordinate) packages and arbitrary model elements owned by the package. This entails that the package structure is a forest enhanced by the possibility to import packages not directly or indirectly owned. A package has no semantics; although it can structure models (in a similar vein as paragraphs can structure a text), a package is not an abstraction of its contents. In particular, a package is not structure-preserving (since it cannot drag the structural embedding of its contents in a context to itself). Only by use of the facade stereotype is it possible to create a composite for a package that can be considered an abstraction of its content. However, such is really a design pattern and not a native modelling element of the language.

A UML subsystem constitutes an independent part of the whole system. Equipped with interfaces it “represents a behavioral unit in a physical system” [11, p. 2-192]. The model elements of a subsystem are divided into specification and realization elements. Apart from additional structure, “a subsystem is a package and has package properties” [11, p. 459]. Thus, as a model abstraction, it suffers from the same weak semantics.

4 Composition for Model Management in UML

The previous sections showed that UML has various notions of composition, but lacks one that takes the complexity out of models. With UML currently being rebuilt, we refrain here from suggesting changes to the language definition. Rather, we formulate two postulates for the introduction of composition to a future UML:

1. Composition should remain a domain abstraction, i.e., it should continue to tag certain elements of a model as composites (“wholes”) and others as components (“parts”), as found appropriate for the modelled domain.
2. At the same time, composition should become a model abstraction, i.e., it should be possible for model elements that are (meta-)classified as composable that they can be abstracted (together with their structure) into a composite. By taking on the external properties of the abstracted components, this composite should be able to completely replace for the composed model elements in the context of its occurrence.

These two postulates grant that — besides being a means of capturing certain abstractions identified by the modeller as structuring the modelled domain — composition/decomposition receives a status comparable to that of generalization/specialization and classification/instantiation, in particular that it comes with strong model management semantics. Specifically, composition should allow the expansion of single model elements into detailed views composed of other model elements interconnected to achieve the purpose of the composite, and this expansion mechanism should be invertible and uniform to all kinds of (de-)composable model elements. To illustrate this, we shall go through a set of concrete examples.

4.1 Composition at the Instance Level

Object Diagrams. Although not the most popular diagram type, object diagrams are very useful for identifying (and defining) composition relationships. For example, Figure 3a shows an exemplary set of interconnected objects constituting a parser. Note that according to the UML specification, aggregation has no special semantics (as regards model management): if a modeller decided to reduce detail by dropping the nodes and the symbols, the (indirect) connection between `aParseTree` and `aSymbolTable` is lost.

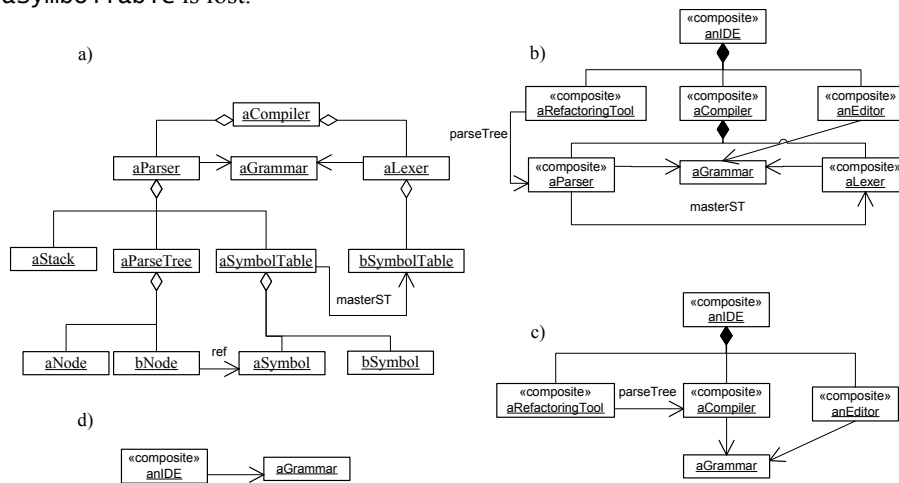


Figure 3. a) conventional object diagram describing the structure of a compiler using UML’s aggregation to designate its parts (standard composition could also have been used where deemed appropriate) b) composition diagram showing the compiler in a larger context, with some of the details abstracted away; the composition relationship has special semantics (see text) c) `aCompiler` collapsed with `parseTree` and relationships to `aGrammar` being redirected d) highest level of abstraction

Figure 3b is different in that it employs a form of composition with properties as postulated above, i.e., one that — besides serving as a valid domain abstraction — is an encapsulating and structure-preserving model abstraction. Note that, even though the (sub)components of `aParser` and `aLexer` have been dropped, the (indirect) relationship of `aParser` and `aLexer` is retained (a consequence of structure-preservation). In the same vein, the newly added `aRefactoringTool` needing access to the parser’s parse tree accesses `aParser` instead. When the components of `aCompiler`, `aParser` and `aLexer`, are subsequently abstracted away, the associations of Figure 3c are introduced to preserve the original structure.

Note that the diagrams of Figure 3b and 3c do not separate different levels of abstraction, since some composites are shown together with their components in one diagram (the corresponding compositions are expanded). Alternatively (and analogous to SDL), but too spacious to be shown here the components of the expanded

composite together with their interconnections and relationships to the outside world could be shown in a separate diagram, so that each diagram would cover only one level of abstraction. In this case, only the «`composite`» stereotype indicates at the composite level that there is something hidden underneath. Such is the case in Figure 3d: `anIDE`, a composite, hides all its internals (except for the fact that one or more of its components need access to `aGrammar`).

Abstract Instances. Indeed, it is not `anIDE` itself that holds a reference to `aGrammar`: this instance — like all other instances of Figure 3 stereotyped composite — is a mere abstraction artifact serving, at the given level of abstraction, as a placeholder for the object structure it abstracts from (that is, it is composed of). When expanding (or drilling down) the abstraction, the instance dissolves, with all references to and from the abstraction being redirected to the corresponding responsible parts.⁵ Of course the modeller is free to introduce a façade or focus object for each composite which takes on the position of the abstract instance at the component level. This façade object may realize the requests to the composition by collaborating with the other components, but the relationship between the façade and the other parts of the abstraction is not that of composition: it may be regarded the components’ *primus inter pares* (or “primary object” [19]), but it is not the composite. Thus, at the lowest possible level of abstraction no composite instances will be left, which is why we regard them as *abstract instances*.

Collaboration Diagrams. A collaboration is a structure of objects interconnected so that through interacting they can jointly fulfil a certain purpose. A collaboration specifies a composition of interacting objects, but the common abstraction, the composite, is not shown. It is useful however to regard the collaboration itself as an abstract (composite) instance that possesses the behaviour specified in the corresponding collaboration diagram(s).

With the whole collaboration diagram being regarded as the (partial) specification of an abstract instance, a recursive decomposition of collaboration diagrams reduces to assuming that one of its collaboration objects is itself a composite, i.e., abstract instance.⁶ The instance then stands for (expands to) a lower level collaboration as indicated above, and a call to an operation decomposes to the calls specified in the abstracted collaboration. In a way, the composition of collaboration diagrams introduces functional decomposition into UML.

Use Case Diagrams. Use cases are units of functionality; they can be related through generalization and dependency relationships. Adding the dimension of composition, a use case can be recursively decomposed into smaller-grained use cases, from which it abstracts. Even if semantically this is partially redundant to the other possibilities of

⁵ For instance, there need not be a concrete instance `aParseTree` when a reference to the root node of such a tree suffices. Note how this accords with the hardware analogy of Section 2.3.

⁶ Because a classifier role in a collaboration diagram stands for an instance of a certain type, we treat collaboration diagrams on the instance and on the specification level alike.

relating use cases, it reintroduces the traditional function tree to software modelling, complementing the functional decomposition resultant from nested collaboration diagrams.

4.2 Impact of Composition on the Type Level

As has been noted previously, we have tied composition to the instance level; yet the composability of objects of certain types is something that is declared on the type level. As indicated by Figure 1, instance and type level are related through classification, which is a systematic abstraction process (*cf.* Section 2). By applying this process to Figures 3a and 3b, Figures 4a and 4b can be derived as classification abstractions.⁷ The point of question here is whether and how abstraction through composition translates from the instance to the type level.

In mapping Figure 3a to the class level, the symbol tables of `aParser` and `aLexer` are abstracted to a single class `SymbolTable` (Figure 4a). Consequently, the reference from `aSymbolTable` to `bSymbolTable` maps to an association from `SymbolTable` to itself. At the same time, the information that a parser has access to the symbol table of a lexer is lost: the parser's symbol table could equally well hold a reference to itself (or some other symbol table not owned by any lexer). Thus, without further information abstraction through composition on the class level cannot faithfully restore or drop (abstract away) this association: Figure 4b cannot be derived deterministically from Figure 4a.

If on the other hand the abstracted object diagram of Figure 3b is mapped to the class level, the relationship from `Parser` to `Lexer` is naturally retained, as shown in Figure 4b. It is instructive to note that, although `Parser` is collapsed (hiding `ParseTree` and other classes delivering the components of a parser) `Stack` and its associations are still present; this is so because `Stack` is also referenced from some other class (the class not being shown), and since this reference may have nothing to do with parsing, it cannot be redirected to `Parser`. As a matter of fact, declaring a composition between `Parser` and `Stack` does not mean that the class `Parser` is an abstraction of the class `Stack`, only that instances of `Stack` may serve as components of instances of `Parser` (as well as composites of instances of many other component types). We therefore conclude that abstraction through composition on the class level is but a consequence of abstraction on the instance level (*cf.* Figure 1); performing it on the class level directly (if only for the purpose of model management) requires additional information, specifically information that pertains to the instance level.

One further issue needs mentioning. Replacing a component at an association's end by a composite (as, for instance, is the case in Figure 4b, where `Parser` and `Lexer` replace for `SymbolTable` at the ends of `masterST`) results in a typing problem: the composite — if represented by a class — is usually a different classifier than its components. Expanding or collapsing a composition would then entail the change of an association end's (or attribute's) associated type. However, this problem is not a problem if every such association connects to an interface (as opposed to a class), and if

⁷ For illustrative purposes, a reference to `Stack` has been added.

the component and the composite implement the same interface. In fact, since the composite class is abstract in the sense that its instances are abstract (i.e., physically non-existent, meaning that they do not survive to the lowest level of abstraction)⁸, it suffices to equate a »composite« classifier with a collection of behaviour specifications (analogous to the collaboration roles of a collaboration diagram, with no need for a dedicated class definition). That having associations end at interfaces is both natural and solving many problems of the UML specification so that it should be the rule rather than the exception is strongly argued for in [15].

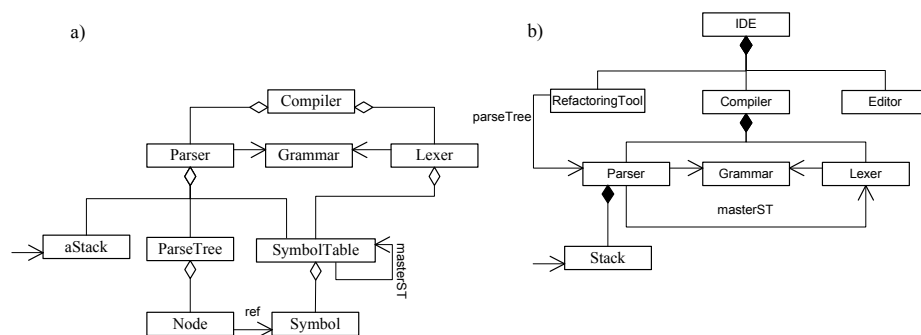


Figure 4. a) Class diagram abstracting (through classification) from Figure 3a. b) Classification of Figure 3b, which cannot be derived through composition from 4a (see text).

5 Related work

While composition as a means of assembling components (in the sense of component-based software engineering) is certainly an exhaustively covered topic, composition as a model abstraction mechanism appears not to be. Even though composition (under the name of aggregation) was already incorporated in the semantic data model [1], its potential impact on model management, in particular for mastering the complexity of real systems, has to our knowledge not been sufficiently exploited.

One of the few notable exceptions is the reference model of open distributed processing [6], which adopts a notion of composition very much in line with ours:

“Composition:

- a) (of objects) A combination of two or more objects yielding a new object, at a different level of abstraction. The characteristics of the new object are determined by the objects being combined and by the way they are combined. The behaviour of a composite object is the corresponding composition of the behaviour of the component objects.

⁸ Again, the modeller is free to introduce concrete façade classes, but these are no composites.

b) (*of behaviours*) *A combination of two or more behaviours yielding a new behaviour. The characteristics of the resulting behaviour are determined by the behaviours being combined and the way they are combined.*” [6] (p. 5)

However, the standard (as well as several other works relying on it, such as the UML Profile for EDOC [12]) leaves it open how to integrate composition thus defined into a modelling language such as UML.

Among the few approaches addressing this topic are those by Wienberg et al. [19], and by Cariou and Beugnard [3]. Wienberg et al. have proposed what they call *dynamic components* (runtime entities based on objects) as a generalization of component instances and nodes of the UML implementation diagrams [19]. They introduce a new association type (called *component link*, similar to aggregation) which designates the relationship between the component and its subcomponents (should be: the composite and its components). Composites are themselves objects; they are called the primary objects of a composition. Upon coarsening (zooming out) of a model, the primary objects are the only entities that remain of a composition; they inherit the structure and behaviour of the abstracted components as accessed from the outside. This is applied to both structural (object) and behavioural (collaboration and sequence) diagrams on the instance level. On the type level, component structure is expressed by a component aggregation stereotype.

Our major criticism of this approach is that the primary objects are concrete objects that in a collapsed view are specified with properties they do not possess, leading to unresolved inconsistencies between different views of the same model. In fact, the primary object is sometimes, and sometimes not the complete interface to the component. Also, Wienberg et al. present their work mainly as a set of notational extensions (stereotypes) to various diagram types, leaving the integration into the language core (in particular the delimitation between their newly introduced and existing model elements) open. Lastly, by considering only the instance level, the authors fail to address the specification of component types, one of the most interesting aspects of component-based modelling.

Cariou and Beugnard have proposed the reification of UML collaborations into interaction components, called *mediums* [3]. Mediums are special classes specifying the protocol covered by the collaboration. However, despite the fact that the medium is to represent the collaboration as a whole, it is dependent on additional interfaces, namely those of the components the medium interacts with, so that it appears that the medium is rather a controller coordinating the collaboration than a representative of the collaboration itself. Although certain parallels to our view of collaborations as compositions are observable, our work is much more general.

6 Conclusion

We have proposed to complement classification and generalization with composition as a third form of model abstraction. By identifying five technical properties of abstraction and their impact on the process of modelling, we have made plausible that

composition has particular potential to let the modeller recursively assemble and decompose models, maintaining manageability at all levels of abstraction.

References

1. S Abiteboul, R Hull „IFO: A formal semantic database model“ *ACM Transactions on Database Systems* 12:4 (1987) 525–565.
2. C Atkinson et al. *Component-based product line engineering with UML* (Addison-Wesley, 2002).
3. E Cariou, A Beugnard “The specification of UML collaborations as interaction components” in: JM Jézéquel, H Hußmann, S Cook (eds) *UML 2002 — Proceedings of the 5th International Conference* Springer LNCS 2460 (2002) 352–367.
4. EW Dijkstra “EWD227 Stepwise Program Construction” in: *Selected Writings on Computing: A Personal Perspective*, Springer (1982)
5. ISO/IEC 10027 *Information Technology — Information Resource Dictionary System (IRDS) Framework* (Genève 1990).
6. ISO/IEC 10746-1 *Information technology — Open distributed processing — Reference Model: Part 2*.
7. ITU *Specification and Description Language (SDL)* ITU-T Recommendation Z.100 (1995).
8. RE Fairley *Software engineering concepts* (McGraw-Hill, 1985).
9. B Henderson-Sellers, F Barbier “Black and white diamonds” in: *UML 1999* (1999) 550–565.
10. J Odell “Six different kinds of composition” *Journal of Object-Oriented Programming*, (January 1994)
11. OMG *Unified Modeling Language Specification* Version 1.4 (OMG, September 2001).
12. OMG *UML Profile for Enterprise Distributed Object Computing Specification* (OMG, February 2002).
13. J Rumbaugh, I Jacobson, and G Booch, *The Unified Modeling Language Reference Manual* (Addison Wesley, 1999).
14. B Selic, G Ramackers, C Kobryn “Evolution, not revolution” *CACM* 45:11 (2002) 72.
15. F Steimann „A radical revision of UML’s role concept“ in: E Evans, S Kent, B Selic (eds) *UML 2000: Proceedings of the 3rd International Conference* (Springer, 2000) 194–209.
16. F Steimann “The family pattern” *Journal of Object-Oriented Programming* (2001) 28–31.
17. S Strahringer *Metamodellierung als Instrument des Methodenvergleichs: Eine Evaluierung am Beispiel objektorientierter Analysemethoden* Dissertation (Darmstadt 1996).
18. C Szyperski *Component Software. Beyond Object-Oriented Programming* (Addison Wesley, 1998).
19. A Wienberg, F Matthes, M Boger “Modeling dynamic software components with UML” in: RB France, B Rumpe (eds) *«UML»’99 — Proceedings of the 2nd International Conference* Springer LNCS 1723 (1999) 204–219.