Available online at www.sciencedirect.com

ScienceDirect

Information and Software Technology 49 (2007) 871–891

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Object-based and class-based composition of transitive mixins

Uwe Zdun [a,*], Mark Strembeck [b,*], Gustaf Neumann [b]

[a] *Distributed Systems Group, Information Systems Institute, Vienna University of Technology, Austria*
[b] *New Media Lab, Institute of Information Systems, Vienna University of Economics and BA, Austria*

Received 17 February 2006; received in revised form 14 September 2006; accepted 3 October 2006
Available online 27 November 2006

Abstract

In object-oriented composition, classes and class inheritance are applied to realize type relationships and reusable building blocks. Unfortunately, these two goals might be contradictory in many situations, leading to classes and inheritance hierarchies that are hard to reuse. Some approaches exist to remedy this problem, such as mixins, aspects, roles, and meta-objects. However, in all these approaches, situations where the mixins, aspects, roles, or meta-objects have complex interdependencies among each other are not well solved yet. In this paper, we propose transitive mixins as an extension of the mixin concept. This approach provides a simple and reusable solution to define "mixins of mixins". Moreover, because mixins can be easily realized on top of aspects, roles, and meta-objects, the same solution can also be applied to those other approaches.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Mixins; Mixin classes; Software composition; Object-oriented composition

1. Introduction

In many object-oriented approaches, the (multiple-)inheritance relationship and the type concept are modeled via the same construct, the class. However, (multiple-)inheritance primarily aims at the reusability of classes, whereas a class primarily defines the type of its instances, the objects (see also [8]). These two goals are often contradictory, as, on the one hand, a unit of reuse should be small and flexibly composable with arbitrary kinds of other classes, and, on the other hand, an object's type needs to be defined completely and requires a fixed place in the class hierarchy. Mixins are proposed as a way to solve this problem (see, e.g. [3,5,6,26,29,41]). A mixin is a small unit of composition that is not necessarily defined completely. It can be *mixed* into a given class hierarchy at arbitrary places.

An open issue in mixin-based composition is the composition of multiple mixins in dependency to each other, i.e., how to define "mixins of mixins". As mixins can be used in arbitrary places of a class hierarchy, it is hard to define the interdependencies between them in the context of compositions in a generic way.

Consider a simple example: an access control handler is conditionally composed with application logic classes in a server. The access control handler depends on a number of other classes: for example the remote objects which have to be protected, the users and/or roles whose access requests to objects must be controlled, the permissions of each particular user or role, and the context constraints for these permissions.[1] As these classes are all together defining the access control handler type, and each of them should be flexibly composable and reusable in many situations, it seems to be a good choice to model each of them as a mixin. In a "flat" mixin model, however, we are not able to model the interdependencies among these mixins. Example problems in such models are that conditional composition based on runtime state is not possible, the composition order cannot be specified, multiple roles of one type cannot have different instance-specific permissions, or all mixins would

* Corresponding authors.
E-mail addresses: zdun@infosys.tuwien.ac.at (U. Zdun), Mark.Strembeck @wu-wien.ac.at (M. Strembeck), Gustaf.Neumann@wu-wien.ac.at (G. Neumann).

[1] The access control example will be discussed in detail in Section 5.2.

be applied to the user or role instances. Instead, we would like to be able to explicitly model a kind of "mixin of mixin" relationship: a role mixin might only be configured for users, a permission mixin only for roles, and a context constraint mixin only for permissions. In such cases, the problem arises how these interdependencies among the classes can be properly modeled while still retaining the reusable type relationship offered by the mixin concept.

In this paper, we propose the transitive composition of mixin classes as a solution to this problem. In particular, in our basic concept, which is called *transitive mixin chains*, each mixin can transitively have other mixins itself, to model (and arbitrarily refine) mixin-based compositions. This way, multiple class hierarchies, expressing orthogonal concerns, can be (dynamically) composed with the application logic in a transitive fashion. Moreover, we also introduce the more elaborate *transitive mixin delegation* concept. It allows each transitive mixin to have its own (object-specific) state. These concepts are defined in a generic way using Horn clauses in Section 3.

Our approach applies mixins as a simple basic concept for reusable types. We used this approach because of the generality of the mixin concept. Similar concepts are present in many recent adaptation techniques, including aspect-oriented programming, meta-object protocols, roles, message interceptors, interpreters, virtual machines, etc. That is, our approach can also be implemented as an extension to these other techniques, and hence we expect a wide applicability of our concepts.

Section 4 presents a proof-of-concept implementation of our approach that is based on XOTcl mixins [29,30]. Subsequently, in Section 5, we illustrate the practical use of the concepts with two case studies, a persistence manager component and a role-based access control framework. We present these details, because we feel that – even though the concept in general and its use are quite simple and straightforward – the implementation details are not obvious. We illustrate the general problems in implementing transitive mixin classes by explaining the design challenges and decisions of our implementation, as well as the corresponding case studies. Of course, many design challenges can be solved quite differently in other implementations of our concepts. In Section 6 we evaluate our findings and Section 7 concludes the paper.

2. Discussion of related work

In addition to the related work regarding the area of mixin-based composition [3,5,6,26,41], mentioned in the previous section, various other extensions and implementation concepts for mixins have been proposed.

A number of approaches suggest to add mixins in a type safe framework. For instance, Flatt, Krishnamurthi, and Felleisen present a mixin approach for Java [11] that is conceptually similar to mixin-based inheritance.

Van Hilst and Notkin describe an implementation technique for roles using C++ templates [40]. Here, roles are composed using inheritance, and the superclass of a role is specified as a template argument. This approach can be seen as a form of mixin classes that are statically composed (i.e., composed before runtime).

Smaragdakis and Batory simplified and extended the idea of role mixins into the mixin layers concept [35]. Mixin layers group multiple mixin classes into a container class. The parameter (superclass) of the outer mixin determines the parameters (superclasses) of inner mixins. Thus, the approach is more structured compared to Van Hilst's and Notkin's approach, and it uses a more simple instantiation style. In both approaches, however, it can be challenging to understand how the pieces compose together (mainly due to the use of templates and other complex C++ language features).

Traits [34] support the reuse of method collections over several classes. They are groups of methods that act as units of reuse from which classes are composed. Thus, traits are pure units of reuse consisting only of methods (similar to the per-class mixins presented below).

In all mixin approaches, explained so far, mixins are not supported as explicit mixin entities, but rather seen as pure extensions of the inheritance relationship. That means, both stateful composition of mixin roles and object-specific composition of mixins is not supported. Moreover, transitive mixin composition is not supported in any of the approaches. In this paper, we will extend those other mixin concepts to support each of these facets as an option that can be chosen by the developer.

A number of more dynamic, object-oriented environments, such as CLOS [3], Smalltalk [15], and Self [39], provide both a programming environment and a runtime environment, allowing to influence the language behavior from within a program. For this purpose, different language constructs are supported, such as computational reflection [25,36], meta-object protocols (MOP) [19], meta-classes [12], dynamic classes, or delegation constructs [2]. With these constructs, a given composition and even the composition mechanism itself can be manipulated and adapted to a given context. The above mentioned techniques provide a great expressive power to the developer. Yet, they also impose a high complexity. To understand an expression, the current runtime definition of the environment has to be understood (including class relationships, meta-objects and meta-classes, and even (re-)definitions of language elements). As there is no standard way to express interdependencies between class relationships, manipulations of these interdependencies are often hand-built. Thus, they look different in different applications and are not easy to understand for the developer.

To limit the complexity, but still allow for powerful software adaptations, different approaches have been proposed. A number of these approaches can be classified as aspect-oriented programming (AOP) [21] approaches. AspectJ [20], for instance, allows to declaratively provide "pointcuts" which are performing adaptations for a number of predefined "joinpoints". Joinpoints are specific, well-defined events in the control flow of the executed program. Aside from AspectJ, there are many other aspect composition frameworks. They have in common that they are easier to understand and apply than meta-programming or reflection. Also they provide a runtime indirection layer [42], so that an aspect can react on context changes at runtime. However, as these mechanisms focus on static adaptation techniques, they cannot be applied directly for runtime changes of the aspect configuration. Therefore, a number of extensions offer dynamic aspect composition. For instance, Prose [33] and Steamloom [4] modify the Java Virtual Machine to allow for dynamic configuration of aspects.

JBoss AOP [7] introduces a simple notion of mixins into an aspect-oriented programming framework. In particular, a mixin class and a number of additional interfaces are added to a class using byte-code manipulation. The mixin class provides the implementation of the methods introduced using the additional interfaces. At runtime, an instance of the mixin class is created for each instance of the class that is extended with the respective mixin. JBoss AOP mixin classes introduce methods that can be used by interceptor methods. This feature resembles AspectJ's inter-type declarations [20].

The AOP approaches discussed above lack a clear solution for interdependencies of aspects. Aspects of aspects can be realized by a few research prototypes, such as Hyper/J [38] or EAOP [10]. In most AOP implementations, however, it is difficult to compose aspects of other aspects, because the aspects are most often composed in a linear chain with a predefined order. Inter-aspect dependencies are thus often hard to model (i.e., only with complex workarounds), and resulting solutions are complex and hard to understand. Furthermore, when aspects have interdependencies among each other, it is difficult to determine which aspects are applied to which composition units in what order.

The composition of roles has been studied in a number of approaches. An object is allowed to possess or play one or more roles. Typically, an object plays the roles which are associated with the class from which the object was instantiated (see e.g. [1,16,32]). Some of these role approaches distinguish class and role hierarchies, as for instance [1,16]. This way, roles are differentiated into role types and can be further specialized. This concept is similar to mixin class hierarchies but offers no concept for transitive interdependencies of roles.

Kristensen and Østerbye [23,24] extend the earlier role concepts with the notion of "roles of roles". This concept is transitively applicable, however, it requires manual casting to a role's context. For instance, if an instance of a class `C` has a role `R1`, and `R1` itself has a role `R2`, then the methods of `R1` and `R2` are not directly available to clients of the instance of `C`, but first this instance must be classified to `R1` or `R2`, respectively. A problem of this approach is that changes to clients are necessary to acquire the mixin behavior, which, again, hinders unanticipated evolution and reuse.

Zhao and Foster propose to model roles using the Cascade pattern [46]. Cascade uses a tree structure to represent roles. Each Cascade layer is a Composite pattern [13]. Through the repeated use of the Composite pattern on different levels, the Cascade pattern achieves an explicit semantic layering and ordering in whole-part relationships. Zhao's and Foster's approach can be used to model class interdependencies of Cascade layers, but manual forwarding through the Cascade hierarchy is required.

In the Object Teams approach [17], role concepts are combined with concepts from AOP. Here, a Team is a class that contains direct inner classes that each implement a role. Instances of a base class will always be associated to instances of the role classes in the team. A particular base class instance can be associated to multiple role objects in multiple teams. The concept allows for method bindings between a class and its roles, which enable automated method forwarding in both directions (so-called CallIns and CallOuts).

Most approaches explained above provide some special language construct, such as a dedicated mixin construct, meta-object, role, or aspect. Even though the composition approaches are slightly different, they all can be applied to extend or compose type relationships with the ordinary inheritance hierarchy. We can distinguish the following differences: in some approaches, the composition can be changed at runtime, others do not provide this feature. Some approaches apply the composition per individual instance (per-object), others do it for classes (per-class). When the composition is applied, it can retain the identity of the object onto which it is applied, or there might be a separate compositional instance (like a role or mixin instance) having its own (unique) identity. Moreover, the details of how the composition is actually applied vary, e.g.: the ordering of the composition; the application of compositions before, after, or around the execution of an actual method invocation; the resolution of ambiguities in the class graph.

In the approach presented in this paper, we aim to remedy the composition problems, identified in the related work especially with regard to transitive mixin composition. Even though the composition is realized in different ways, our approach can be applied – with moderate effort – on top of most of the other approaches. In Section 6 we will compare our approach to those other approaches.

3. Transitive mixin classes: concepts

Our goal is to extend the mixin concept with support for transitive mixins ("mixins of mixins") – as a way to model the interdependencies among different mixins. We have chosen mixins as a conceptual foundation for our approach because mixin implementations and approaches exist for many environments, and the mixin concept can be realized in or on top of many other adaptation techniques, including aspect-oriented programming, meta-object protocols, roles, message interceptors, interpreters, virtual machines, etc. Thus, there is a wide potential for a common applicability of our results.

In the remainder of this section we define the declarative semantics of two novel mixin concepts, *transitive mixin chains* and *transitive mixin delegation*. We present our concepts in a generic form, expressed via Horn clauses. The goal is to express a general model that can be used with many of the approaches explained in the preceding section. Even though there are slight differences in these approaches, our transitive mixin approach can be implemented on top of other existing approaches, such as mixin, role, aspect, and MOP approaches. It may, however, require some modifications of concrete mixin implementations to fully realize our concepts.

To make our approach as general as possible, we do not presume that a special language construct, such as a dedicated mixin construct, meta-object, or aspect, must be used to implement mixins. Instead, in our approach, a mixin must just have the properties of an ordinary class. In our proof-of-concept implementation explained below, we indeed use ordinary classes for implementing mixins, however, this is again no prerequisite. Of course it is also possible to use more advanced constructs to implement mixins, such as dedicated mixin constructs, meta-objects, or aspects.

3.1. Basic type relationships

In this section, we provide declarative semantics for *transitive mixin chains* and *transitive mixin delegation*. We first define the basic type relationships that we presuppose for the following mixin concepts. Therefore, we define the following facts describing basic object-oriented constructs:

- Classes are specified via *is_class*(*C*).
- Superclass relationships are specified via *superclass* (*C*,*S*). It is not specified if a class *C* can have only a single superclass *S* or multiple superclasses, to cover single as well as multiple inheritance.
- Instances of classes (i.e., objects) are defined via *instance_of*(*O*, *C*).
- The methods provided by a particular class *C* are specified via *provides_method*(*C*, *M*).

Based on these facts, we define a set of clauses describing relationships that are common in most object-oriented languages (see Fig. 1): each class *C* defines a (custom) type for its instances (Clause 1). In presence of a class hierarchy, a class *C* also provides all types defined by its superclasses (Clause 2). An object *O* is said to be of a type *T*, if this type *T* is provided by a class *C*, and *O* was instantiated from *C* (Clause 3). This definition is sufficient for single and multiple inheritance relationships alike. Clause 4 defines which methods can be invoked on a particular object based on the predicates *is_type_of* and *provides_method.*

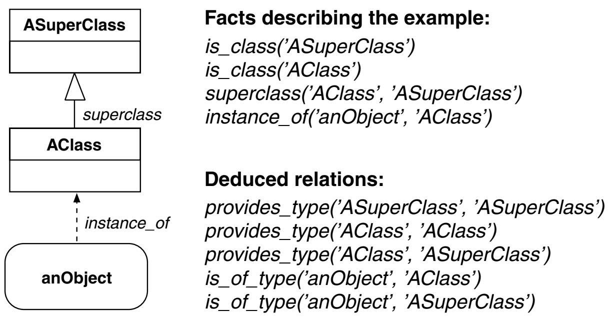
Fig. 2 depicts a simple example including two classes, *ASuperClass* and *AClass*, and an instance of *AClass* named *anObject*. Furthermore, the figure shows the facts needed to describe the example and the relations that can be deduced via the clauses defined in Fig. 1.

3.2. Declarative semantics of mixins and transitive mixin chains

As an extension to the basic relationships we now define the semantics of mixins. As motivated above, we simplify the type semantics of existing mixin concepts, and, at the same time, make them work in a transitive fashion. For this reason we assume that a mixin is an ordinary class, supporting all the relationships defined in Fig. 1. Again, we first define some basic facts that are then used to define additional mixin related clauses: the fact *has_per_object_mixin*(*O*, *P*) specifies that an object *O* has a per-object mixin *P*. And the fact *has_per_class_mixin*(*C*, *P*) defines that class *C* has a per-class mixin *P*.

Fig. 3 specifies that per-object and per-class mixins extend the *is_of_type* and *provides_type* predicates defined in Section 3.1:

- *Per-object mixins* are classes that are applied as mixins for an individual object, i.e., for an instance of a class (see Clause 5). They extend the types of an object with (one or more) per-object mixin classes. Fig. 4 shows an example of how an object *anObject* acquires two new



Facts describing the example:
is_class('ASuperClass')
is_class('AClass')
superclass('AClass', 'ASuperClass')
instance_of('anObject', 'AClass')

Deduced relations:
provides_type('ASuperClass', 'ASuperClass')
provides_type('AClass', 'AClass')
provides_type('AClass', 'ASuperClass')
is_of_type('anObject', 'AClass')
is_of_type('anObject', 'ASuperClass')

Fig. 2. Class and superclass relationship: example.

$$\forall\, C : is\_class(C) \longrightarrow provides\_type(C, C) \tag{1}$$

$$\forall\, C, T\ \exists\, S : superclass(C, S) \wedge\ provides\_type(S, T) \longrightarrow provides\_type(C, T) \tag{2}$$

$$\forall\, O, T\ \exists\, C : instance\_of(O, C) \wedge provides\_type(C, T) \longrightarrow is\_of\_type(O, T) \tag{3}$$

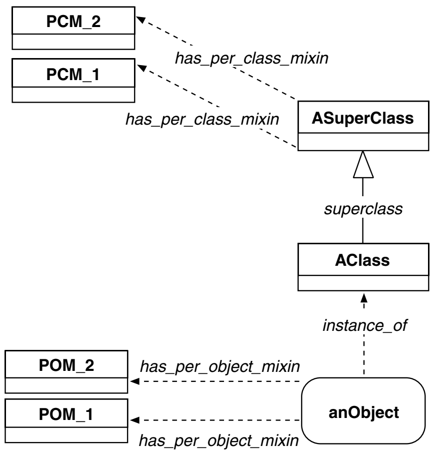
$$\forall\, O, M\ \exists\, T : is\_of\_type(O, T) \wedge provides\_method(T, M) \longrightarrow can\_invoke(O, M) \tag{4}$$

Fig. 1. Class and superclass relationship.

$$\forall O,T \; \exists P : has\_per\_object\_mixin(O,P) \wedge provides\_type(P,T) \longrightarrow is\_of\_type(O,T) \qquad (5)$$

$$\forall C,T \; \exists P : has\_per\_class\_mixin(C,P) \wedge provides\_type(P,T) \longrightarrow provides\_type(C,T) \qquad (6)$$

Fig. 3. Mixin relationships (1): transitive mixin chains.



Facts describing the example:

is_class('ASuperClass')
is_class('AClass')
is_class('PCM_1')
is_class('PCM_2')
is_class('POM_1')
is_class('POM_2')
superclass('AClass', 'ASuperClass')
instance_of('anObject', 'AClass')
has_per_class_mixin('ASuperClass', 'PCM_1')
has_per_class_mixin('ASuperClass', 'PCM_2')
has_per_object_mixin('anObject', 'POM_1')
has_per_object_mixin('anObject', 'POM_2')

Deduced relations:

provides_type('ASuperClass', 'ASuperClass')
provides_type('AClass', 'AClass')
provides_type('PCM_1', 'PCM_1')
provides_type('PCM_2', 'PCM_2')
provides_type('POM_1', 'POM_1')
provides_type('POM_2', 'POM_2')
provides_type('ASuperClass', 'PCM_1')
provides_type('ASuperClass', 'PCM_2')
provides_type('AClass', 'ASuperClass')
provides_type('AClass', 'PCM_1')
provides_type('AClass', 'PCM_2')
is_of_type('anObject', 'AClass')
is_of_type('anObject', 'ASuperClass')
is_of_type('anObject', 'PCM_1')
is_of_type('anObject', 'PCM_2')
is_of_type('anObject', 'POM_1')
is_of_type('anObject', 'POM_2')

Fig. 4. Per-class and per-object mixins: example.

types POM_1 and POM_2 via the corresponding per-object mixins.

- *Per-class mixins* are classes that are applied as mixins for a class. Per-class mixins are types for all direct and indirect instances of this class (see Clause 6). Fig. 4 shows an example of how *anObject* as an instance of *AClass* acquires two additional types PCM_1 and PCM_2, if these classes are registered as per-class mixins for *ASuperClass*.

Most mixin concepts can be used to realize at least one of the two relationships, per-object mixins or per-class mixins, and each of the two relationships can be used to "simulate" the other. From a practical point of view, however, it makes sense to define both relationships because they both occur frequently in design situations. Simulating the one with the other is tedious and error-prone.[2]

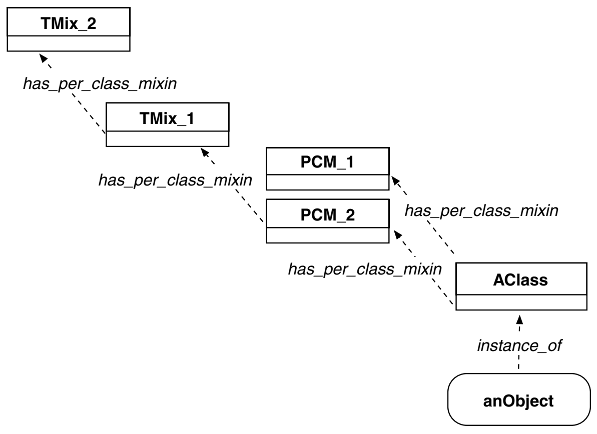
While the example in Fig. 4 only illustrates non-transitive mixins, the clauses provided in Fig. 3 also describe transitive mixin chains. Transitive mixin chains result from mixin classes which themselves have one or more per-class mixins, as illustrated in Fig. 5. In particular, the example in Fig. 5 shows a mixin class PCM_2 which has itself a per-class mixin $TMix\_1$, and $TMix\_1$ again has a per-class mixin $TMix\_2$. The transitive mixin chains concept transitively applies per-class mixins registered on a mixin class for all corresponding objects. With regard to the example in Fig. 5 this means that the instance *anObject* acquires all types obtained by per-class mixins of one of its mixins.

To realize a transitive mixin chain for a per-object mixin, we need to define one or more per-class mixins for the respective per-object mixin class. This is only possible because per-object mixins are assumed to be classes, and hence per-class mixins can be defined on them. In other words, a transitive mixin chain, like the one depicted in Fig. 5, may also be attached to a class that is used as a per-object mixin.

The definitions given in this section introduce mixins as classes that support transitive mixin chains, both on a per-object and per-class basis. The result is a powerful and simple concept for transitive mixin composition. This concept has an important characteristic that needs some more consideration: as a mixin is itself a class of the object, the mixin retains the identity of the object when it is applied. Thus, both per-object and per-class mixins extend the type relationship of the object. Therefore, when a method defined on a mixin is invoked, the instance to which this method is applied must be the same object on which the original method call was invoked. The concept of transitive mixin chains is thus applicable for all mixin implementations that allow to retain the identity of the instance the mixin is registered for. In other words, the identity of an object (that often can be referred to using a language construct called `self` or `this`) does not change when passing a message call to the mixin classes of the corresponding object.

This is a particular strength of the transitive mixin chain concept for many typical application scenarios of mixins because it enables developers to transparently extend an object using mixins. That is, neither the object itself nor the class the object was instantiated from need to be altered to

[2] This problem has been observed in our early XOTcl case studies. Initially, our XOTcl prototype (see Section 4.1) did only support per-object mixins. While this construct was very useful for some design situations, in other design situations we frequently ran into problems when we wanted to apply a mixin for all instances of a class. Hence, we introduced the per-class mixin feature to avoid writing per-object mixin generation code into the constructors of each of these classes.



Facts describing the example:

is_class('AClass')
is_class('PCM_1')
is_class('PCM_2')
is_class('TMix_1')
is_class('TMix_2')
instance_of('anObject', 'AClass')
has_per_class_mixin('AClass', 'PCM_1')
has_per_class_mixin('AClass', 'PCM_2')
has_per_class_mixin('PCM_2', 'TMix_1')
has_per_class_mixin('TMix_1', 'TMix_2')

Deduced relations:

provides_type('AClass', 'AClass')
provides_type('PCM_1', 'PCM_1')
provides_type('PCM_2', 'PCM_2')
provides_type('TMix_1', 'TMix_1')
provides_type('TMix_2', 'TMix_2')
provides_type('TMix_1', 'TMix_2')
provides_type('PCM_2', 'TMix_1')
provides_type('PCM_2', 'TMix_2')
provides_type('AClass', 'PCM_1')
provides_type('AClass', 'PCM_2')
provides_type('AClass', 'TMix_1')
provides_type('AClass', 'TMix_2')
is_of_type('anObject', 'AClass')
is_of_type('anObject', 'PCM_1')
is_of_type('anObject', 'PCM_2')
is_of_type('anObject', 'TMix_1')
is_of_type('anObject', 'TMix_2')

Fig. 5. Transitive mixin chains: example.

extend the object with the new behavior provided by the mixin, it is sufficient to add a new (mixin) relation.

There are other composition scenarios (such as the one described in the case study in Section 5.2), however, in which the transitive mixin chain relationship is not suitable for expressing complex compositions of mixins because we require the object identity to change, when a mixin is applied. This issue also occurs for ordinary object-oriented composition by inheritance: the inheritance relationship also retains the object identity. If the identity should or must change, in ordinary object-oriented approaches, delegation is applied instead of inheritance. Analogously, in case of a mixin-based composition we apply the concept of transitive mixin delegation (explained in the following section) in such situations. In other words: transitive mixin delegation provides a concept for transitive mixin composition based on the concept of delegation.

3.3. Declarative semantics of transitive mixin delegation

A typical example in which the identity of the object should change when a mixin is applied are stateful roles modeled as mixins. A stateful role can be seen as a collection of new *behavior* added to an object *plus* some additional per-role *state*. It should be possible to add different roles independently, and the developer of a certain class/object cannot foresee all possible role extensions, therefore a mixin implementing a role must somehow realize the per-role state.

Unfortunately, it is cumbersome to realize such stateful roles with (transitive) per-class mixins. In particular, we would need to instantiate helper objects to hold the per-role state, and there would be no common concept for such helper objects. Thus, each developer of a stateful mixin would have to realize this concern from scratch, which is likely leading to code that is hard to reuse, understand, and maintain.

For this reason, we introduce the concept of *transitive mixin delegation*. This concept has similar type semantics as the transitive mixin chains explained before, and additionally it defines how to automatically *delegate* the mixin invocation to an object holding the mixin's state. The additional declarative semantics is defined in Fig. 6.

The transitive mixin delegation relationship enables an object to invoke additional methods via delegation (defined on one or more of its mixins). First, we must define the general semantics of "invoking a method via delegation". The *can_invoke* predicate (from Clause 4) is extended in Clause 7 and 8. The *can_invoke* predicate with four parameters specified in Clause 7 defines that an object (first parameter) can invoke a method (fourth parameter) on another object (third parameter) using a delegation that happens when one of its own methods (second parameter) is invoked. Clause 7 extends the *can_invoke* predicate from Clause 4 to the four parameter version. Next, Clause 8 defines how a delegated invocation works: if a method $M1$ can be invoked on object $O1$ and a method $M2$ can be invoked on object $O2$, and the fact $delegate(O1, M1, O2, M2)$ is defined, then the object $O1$ can invoke $O2$'s $M2$ via its own method $M1$. In other words, $M1$ includes a delegation to $M2$. Again, the delegation can happen before, after, or around the invocation of $M1$.

$$\forall O, M : can\_invoke(O, M) \longrightarrow can\_invoke(O, M, O, M) \tag{7}$$

$$\begin{aligned}\forall O1, M1, O2, M2 : delegate(O1, M1, O2, M2) \wedge can\_invoke(O1, M1) \wedge can\_invoke(O2, M2)\\ \longrightarrow can\_invoke(O1, M1, O2, M2)\end{aligned} \tag{8}$$

$$\begin{aligned}\forall O, M1, P, M2 : has\_per\_object\_mixin(O, P) \wedge transitive\_mixin\_delegation(P, M1, M2)\\ \longrightarrow delegate(O, M1, P, M2)\end{aligned} \tag{9}$$

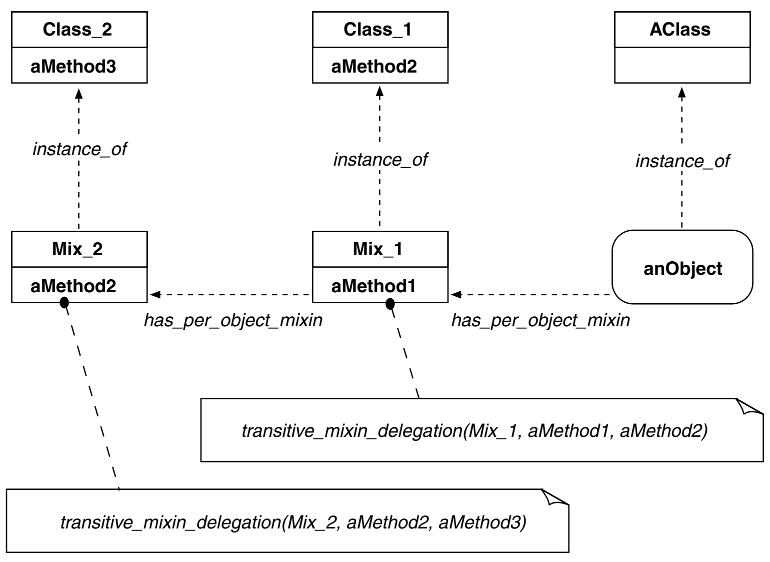
Fig. 6. Mixin relationships (2): transitive mixin delegation.

To define transitive mixin delegation, we assume the additional fact *transitive_mixin_delegation*$(P, M1, M2)$. This fact means that a transitive mixin delegation is defined for a particular per-object mixin P, between methods $M1$ and $M2$. If an object O has a class P registered as a per-object mixin, and *transitive_mixin_delegation* is defined for P's methods $M1$ and $M2$, then a delegation (defined via the *delegate* predicate) between the method $M1$, invokable on O, and the method $M2$, invokable on P, can be deduced. This relation is expressed in Clause 9. Here, P is assumed to be a per-object mixin of the object O because we want to perform an object-specific extension, and in our mixin concept per-object mixins are used to model object-specific extensions (see Section 3.2).

In Clause 8 we assume *can_invoke*$(O1, M1)$ and *can_invoke*$(O2, M2)$. For transitive mixin delegation *can_invoke*$(O1, M1)$ is trivially fulfilled because the object O from Clause 9 of course can invoke its own method $M1$. The second part from Clause 8, *can_invoke*$(O2, M2)$, means that – in Clause 9 – the per-object mixin P itself can invoke a method $M2$. This is a central assumption made by the concept of transitive mixin delegation: *mixins themselves must be able to receive method invocations*. In other words, a mixin must either be an object itself, or it must be represented by some (proxy) object. In the mixin definitions provided above, mixins were assumed to be classes, however (see Section 3.2). Thus, we require classes that can receive method invocations.

There are many ways to realize this assumption. In our proof-of-concept implementation described below (see Section 4) we use the concept of class objects: a *class object* is an object representing the class at runtime while additionally containing a per-class state (that can also be used to realize a per-mixin state). Class objects, however, are not supported by all mainstream programming languages. Thus, in such situations we use some other object to hold the per-mixin state. One simple solution are helper objects for holding a mixin's state. This solution is equally powerful, but less elegant than class objects, because it requires some additional central management facility for the helper objects. As an alternative, we can "simulate" the class object approach: for example, there are many patterns describing how to implement dynamic object systems where classes act as objects, such as Object System Layer [14] or Type Object [18]. These patterns can be realized in almost any object-oriented programming language. Our approach, however, does not assume any of these implementation variants, the only assumption made by our concept is that mixins are themselves able to receive method invocations.

The consequence of the semantics of transitive mixin delegation (defined in Fig. 6) is that per-object mixins can be used to express stateful composition of mixins (like stateful roles for instance) and all its direct and indirect relationships (i.e., ordinary type relationships and other mixin relationships). To illustrate this feature, consider the example diagram in Fig. 7. In this example, an object *anObject* has a per-object mixin *Mix*_1 providing a method *aMethod*1. For this method a transitive mixin delegation to a method *aMethod*2 is defined which is dispatched on the class hierarchy the mixin object was instantiated from (here the method is defined on *Class*_1). Moreover, the mixin class *Mix*_1 has itself a per-object mixin *Mix*_2, which has a transitive mixin delegation from *aMethod*2 to *aMethod*3 (here the method is provided by *Class*_2). As a



Facts describing the example:

is_class('AClass')
is_class('Mix_1')
is_class('Mix_2')
is_class('Class_1')
is_class('Class_2')
instance_of('anObject', 'AClass')
instance_of('Mix_1', 'Class_1')
instance_of('Mix_2', 'Class_2')
provides_method('Mix_1', 'aMethod1')
provides_method('Mix_2', 'aMethod2')
provides_method('Class_1', 'aMethod2')
provides_method('Class_2', 'aMethod3')
has_per_object_mixin('anObject', 'Mix_1')
has_per_object_mixin('Mix_1', 'Mix_2')
transitive_mixin_delegation('Mix_1', 'aMethod1', 'aMethod2')
transitive_mixin_delegation('Mix_2', 'aMethod2', 'aMethod3')

Deduced relations:

provides_type('AClass', 'AClass')
...
can_invoke('anObject', 'aMethod1')
can_invoke('Mix_1', 'aMethod2')
can_invoke('Mix_2', 'aMethod3')
can_invoke('anObject', 'aMethod1', 'anObject', 'aMethod1')
can_invoke('Mix_1', 'aMethod2', 'Mix_1', 'aMethod2')
can_invoke('Mix_2', 'aMethod3', 'Mix_2', 'aMethod3')
delegate('anObject', 'aMethod1', 'Mix_1', 'aMethod2')
delegate('Mix_1', 'aMethod2', 'Mix_2', 'aMethod3')
can_invoke('anObject', 'aMethod1', 'Mix_1', 'aMethod2')
can_invoke('Mix_1', 'aMethod2', 'Mix_2', 'aMethod3')
can_invoke('aObject', 'aMethod1', 'Mix_2', 'aMethod3')

Fig. 7. Example for transitive mixin delegation.

consequence, three delegated invocations can be deduced for *anObject* (using the clauses defined in Figs. 1, 3, and 6):

- *Delegation from aMethod1 to aMethod2*: An invocation of *aMethod*1 on *anObject* is automatically delegated to *aMethod*2 on the object *Mix*_1 (*Mix*_1 is an instance of *Class*_1 and can therefore invoke *aMethod*2 provided by *Class*_1).
- *Delegation from aMethod2 to aMethod3*: An invocation of *aMethod*2 on *Mix*_1 is automatically delegated to *aMethod*3 on the object *Mix*_2 (*Mix*_2 is an instance of *Class*_2 and can invoke *aMethod*3).
- *Transitive delegation from aMethod1 to aMethod3*: An invocation of *aMethod*1 on *anObject* is automatically delegated to *aMethod*3 on the mixin *Mix*_2. This transitive delegation is conducted via *aMethod*2 that can be invoked by *Mix*_1 (see Fig. 7). The state of the mixins is introduced through the object-specific delegation between the mixins. Each mixin is implemented as a class object, and hence it has its own mixin-specific state.

3.4. A decision tree for the modeling of mixin composition

Above we have introduced two novel concepts for modeling mixin interdependencies, transitive mixin chains and transitive mixin delegation, as well as a number of variations for the application of these concepts (per-object vs. per-class; directly applied vs. transitively applied). Now we take a look at the “big picture” and illustrate when which of these variants is applicable. To assist developers in a systematic decision, Fig. 8 provides a decision tree when to apply which of the concepts.

The class primarily defines the type of its instances. Hence, for defining ordinary types the standard class relationship (*instance_of*) or ordinary inheritance (*superclass*) should be used.

In turn, if flexible reusability of classes is the goal, a mixin class should be applied. Thus, if an ordinary object or class is to be extended, we use directly applied mixin classes. Per-object mixins are used for object-specific extensions that apply to an individual object only, and per-class mixins for class-specific extensions that apply for all instances of a particular class.

We also use mixin classes for refining mixin compositions. In this case, however, we apply them transitively (see Fig. 8). We have to further decide whether the object identity should be retained or not (i.e., whether stateless or stateful mixins are needed). If the object identity should be retained, we apply transitive mixin chains, otherwise we choose transitive mixin delegation (see Fig. 8).

As transitive mixin delegation is always object-specific, it requires the usage of per-object mixins. Transitive mixin

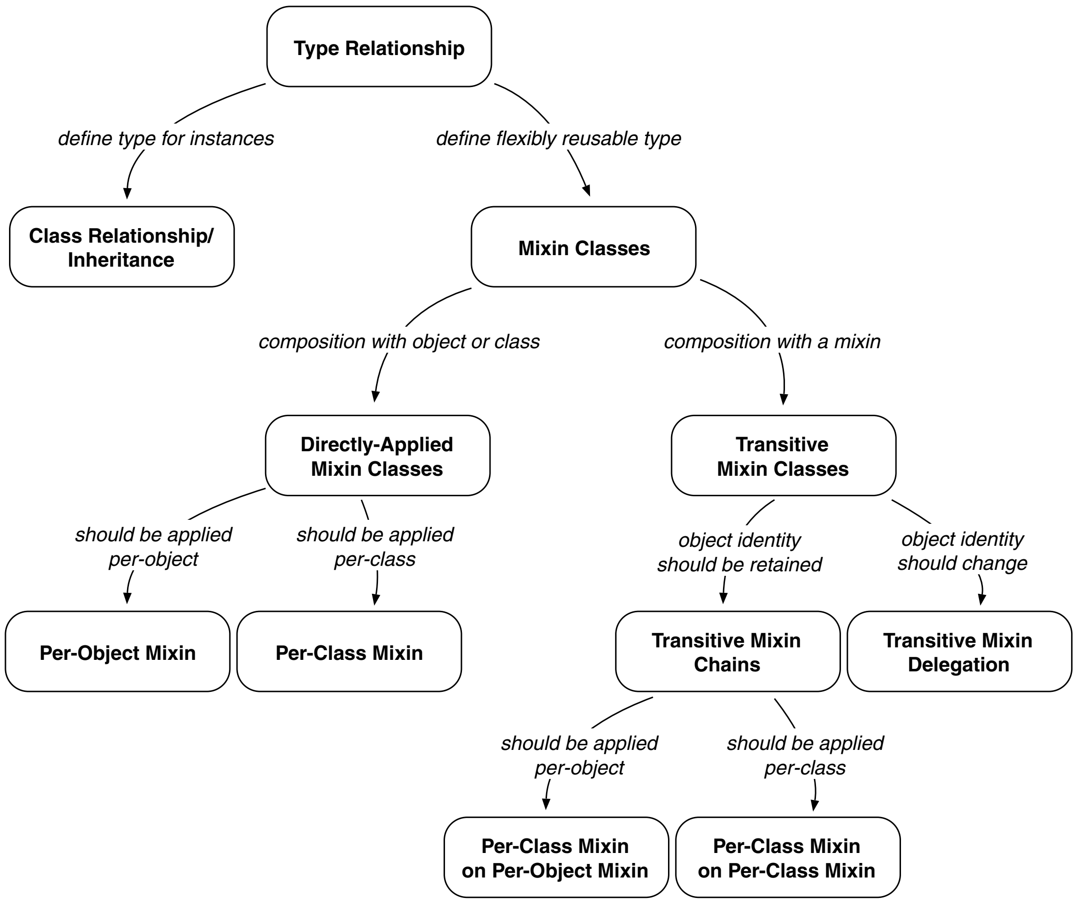

Type Relationship
define type for instances
define flexibly reusable type
Class Relationship/ Inheritance
Mixin Classes
composition with object or class
composition with a mixin
Directly-Applied Mixin Classes
Transitive Mixin Classes
should be applied per-object
should be applied per-class
object identity should be retained
object identity should change
Per-Object Mixin
Per-Class Mixin
Transitive Mixin Chains
Transitive Mixin Delegation
should be applied per-object
should be applied per-class
Per-Class Mixin on Per-Object Mixin
Per-Class Mixin on Per-Class Mixin

Fig. 8. Decision tree for mixin composition.

chains are always class-specific and are thus applied as per-class mixins. There is, however, the choice whether an object-specific mixin relationship should be extended (per-class mixin on a per-object mixin), or if a class-specific mixin relationship is to be extended (per-class mixin on a per-class mixin). The options are summarized in the decision tree depicted in Fig. 8.

4. Proof-of-concept implementation: transitive mixins in XOTcl

As discussed in Sections 1 and 2, many variations of the mixin concept exist. Most of them can, in principle, be used to implement the concept of transitive mixin classes as described in Section 3. In our examples we use XOTcl mixin classes for illustration and as a proof-of-concept implementation. In this section, we provide the essential implementation details, because we found them to be non-obvious and still necessary for a successful realization of the concepts. Our implementation is close to the concepts defined in Section 3. However, other implementations based on other existing frameworks (like existing mixin implementations or AOP frameworks, see Section 2) may of course choose other ways to implement these concepts. For instance, we follow the distinction of per-object and per-class mixins. Most other realizations of mixin concepts do not support both variants. Thus, one of the two has to be simulated using the other, before the concepts presented in Section 3 can be fully realized.

4.1. Proof-of-concept implementation and XOTcl details

For our proof-of-concept implementation we have used an object-oriented extension of the scripting language Tcl [31], called XOTcl (eXtended Object Tcl) [29]. XOTcl is a C-library that can be dynamically loaded into every Tcl compatible environment such as `tclsh` or `wish` and is embeddable in C programs. As a Tcl extension, all Tcl commands [31] are directly accessible in XOTcl. XOTcl is open source and publicly available from [30].

The code for our proof-of-concept implementation, described below, is implemented in C (about 3000 lines of code of the XOTcl C implementation are relevant for the mixin and transitive mixin implementations). In this paper, we will, however, not explain the details of the C reference implementation, rather we describe transitive XOTcl mixins and their application in software development situations – please refer to [30] for the XOTcl source code and a language reference.

Moreover, to show the generality of our results, we have done a second implementation of our concepts in a Java extension, called Frag [44,45]. Basically, this implementation extends the Java-library Jacl [9] with the same mixin concepts as XOTcl. Here, AspectJ [20] is used to connect the dynamic mixin-based object system to existing Java classes (see [43] for details). We will not explain the details of Frag because the XOTcl implementation of the transitive mixin concepts is more advanced and from a user-level view both are very similar. Only the internal implementation details of the XOTcl C implementation and the Frag implementation differ. The source code of the Frag reference implementation can be obtained from [44].

XOTcl is based on the object system of OTcl [41]. This object system enables us to define objects, classes, and meta-classes. Here, classes are special objects with the purpose of managing other objects. In this context, "managing" means that a class controls the creation and destruction of its instances and that it contains a repository of methods accessible for the instances. Every object may be enhanced with object-specific methods.

XOTcl supports single and multiple inheritance. All relationships in XOTcl, including class and superclass relationships, are completely dynamic. Furthermore, XOTcl offers a rich introspection mechanism which allows to inquire nearly all characteristics of XOTcl objects and classes at runtime.

Through the superclass-relation classes are arranged in a directed acyclic graph. XOTcl defines a linearized precedence order for class and mixin hierarchies to avoid potential conflicts during the name resolution (for details see Section 4.3).

In XOTcl every object (and class) may contain other objects (or classes). Objects can be aggregated dynamically by another object at runtime. An aggregation constitutes a part-of-relationship between the corresponding objects.

4.2. XOTcl mixin classes

XOTcl mixin classes are a dynamic message interception technique based on the general mixin concept. They allow to define extension classes in addition to the inheritance hierarchy of the target object a mixin is registered for. For method resolution, mixin classes are searched prior to searching the object's class itself (and the corresponding inheritance hierarchy). XOTcl supports both, per-object mixins (POM) and per-class mixins (PCM), following the generic semantics defined in Section 3.

In XOTcl any "ordinary" class can be registered as a mixin. This design is chosen because developers should not have to learn new features of advanced constructs (such as aspects, meta-classes, or meta-objects) to use mixins. Additionally, this also eases the composition of any existing (e.g. third party) classes, because – provided that there are no name conflicts on the classes – the classes can be composed as mixins without modification.

The predefined `instmixin`[3] method accepts a list of classes to be registered as per-class mixins, whereas the predefined `mixin` method registers classes as per-object mixins.

[3] "instmixin" is a short form of "instance mixin", meaning that a corresponding mixin is applied for all instances of the class the mixin was registered for. XOTcl uses a similar naming convention for methods: a method applying to all instances of a class is called "instproc".

XOTcl mixins may be dynamically added and removed at any time. To keep track of these dynamic relationships, `info instmixin` and `info mixin` provide introspection functions for mixins. Thus, at runtime one can always determine the current mixins of an object or class.

4.3. Method resolution order of class and mixin hierarchies in XOTcl

The concepts introduced in Section 3 allow for the definition of type extensions for an object using mixins. An important implementation facet, when realizing these concepts, is how mixins affect the method resolution order defined by the corresponding object-oriented language – i.e., which methods of which classes are dispatched in what order. In general, many solutions are possible and it is advisable to use a solution that follows the method resolution rules of the respective object-oriented language as close as possible so that developers can apply mixins in a natural way without having to learn semantics that are significantly different from the rest of the language.

In XOTcl, the method resolution order is given by a simple linearization of the class and mixin hierarchies that apply for a certain object. Here, linearization means that duplicates in the method resolution order are eliminated. First, we illustrate this for ordinary class hierarchies and subsequently for mixin classes.

Conceptually, all methods in XOTcl are mixin methods, meaning that they can mix-in the next, same-named method on the class-graph. Let us consider an example: in Fig. 9, a method `aMethod` is dispatched, first on the object `anObject`,[4] then on its class (here: `AClass`), and finally on each of the corresponding superclasses (here: `ASuperClass`). Each method named `aMethod` that is found somewhere in this method resolution order is executed. An XOTcl code skeleton for this situation looks as follows:

```
Class ASuperClass
ASuperClass instproc aMethod args {
  ### code of aMethod
  …
}
Class AClass -superclass ASuperClass
AClass instproc aMethod args {
  ### code of aMethod
  …
}
# instantiation of anObject
AClass create anObject
# method invocation of aMethod
anObject aMethod
```

[4] In XOTcl objects might have object-specific methods which is a specialty of the XOTcl object system. For completeness, we show “methods defined on the particular object” in the method resolution order diagrams (see also Fig. 10), but this language feature is not relevant for realizing the concepts presented in this paper.

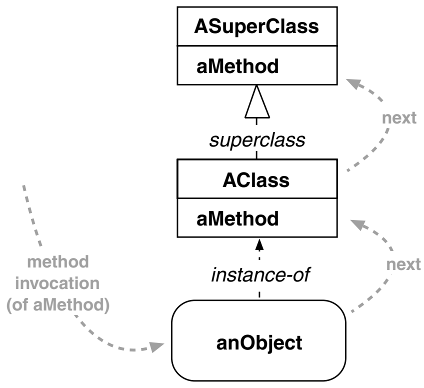


Fig. 9. Method resolution order for an invocation.

We can model before, after, and around behavior of a method call using the placement of the `next` command within the source code. That means, code after the invocation of `next` is executed *after* the invocation of the next same-named method (as shown in the example below), and code before the execution of `next` is executed *before* the invocation of the next same-named method. Before and after code is implemented as a variant of *around* behavior (as also provided in CLOS [3] or AspectJ [20] for example). When omitting `next`, the originally called method is the only method that is executed – i.e., the method call is not forwarded along the method resolution order. The following code is a skeleton of an XOTcl method with `next`:

```
AClass instproc aMethod args {
  ### instructions before ‘next’
  ### (might be omitted)
  …
  ### invocation of ‘next’
  ### (might be omitted)
  next
  ### instructions after ‘next’
  ### (might be omitted)
  …
}
```

XOTcl’s per-object and per-class mixins use the `next` command to forward messages to the mixin chain that is registered for a particular object and to finally pass it to the “original” class hierarchy the object was instantiated from. For instance, we can define two classes `POM_1` and `POM_2` (`POM_1` and `POM_2` are ordinary classes):

```
Class POM_1
POM_1 instproc aMethod args {…}
Class POM_2
POM_2 instproc aMethod args {…}
```

In XOTcl, any class may be assigned the role to act as a mixin class (simply by registration as a mixin class). For instance, we can register these classes as per-object mixins for the object `anObject`. This means that only this

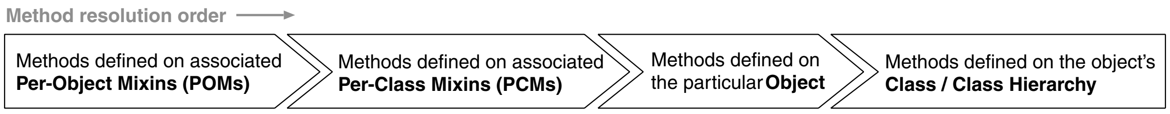

Method resolution order
Methods defined on associated Per-Object Mixins (POMs)
Methods defined on associated Per-Class Mixins (PCMs)
Methods defined on the particular Object
Methods defined on the object's Class / Class Hierarchy

Fig. 10. Method resolution order in XOTcl.

particular object is extended with the functionality of the two mixin classes:

```
anObject mixin {POM_1 POM_2}
```

In contrast to a per-object mixin (as motivated above), a per-class mixin operates on all instances of a class and all instances of its subclasses. For example, the following invocation registers two ordinary classes `PCM_1` and `PCM_2` dynamically for all instances of `ASuperClass` (which means that they are also applied for `anObject` which is an instance of `AClass` – see Fig. 10):

```
ASuperClass instmixin {PCM_1 PCM_2}
```

To avoid conflicts, XOTcl applies an unambiguous method resolution order: before the class hierarchy of an object is searched, XOTcl searches the mixins that are registered for this particular object. Moreover, per-object mixins are applied before per-class mixins (see Figs. 10 and 11). Subsequent to the mixins, the object's own heritage is searched in the following order: object, class, and superclasses (mixins are applied in the same order). All classes (mixins and ordinary classes) in the method resolution order are linearized, and each class may only appear once on a method resolution order because duplicates are eliminated. If a class can be reached more than once, the last occurrence in the linearized list is used.

Each time this method resolution order is used in a method invocation, the method resolution order is searched for the first implementation of the respective method. This particular implementation is then invoked by XOTcl's message dispatcher. Moreover, if this implementation invokes `next`, the next occurrence of this particular method is searched and mixed into the current invocation, and so on.

The registration lists of mixin classes are "order-sensitive", i.e., the order of the mixin classes determines the resulting method resolution order. Fig. 11 depicts the method resolution order for an invocation of a method called `aMethod` on an object `anObject` (using the example mixin registrations from above). The mixins `POM_1`, `POM_2`, `PCM_1`, and `PCM_2` can be reached from `anObject`, as well as its class `AClass` and the superclass `ASuperClass`.

4.4. Transitive mixin chains in XOTcl

As explained in Section 3, transitive mixin chains can be applied to define a mixin composition. For example,

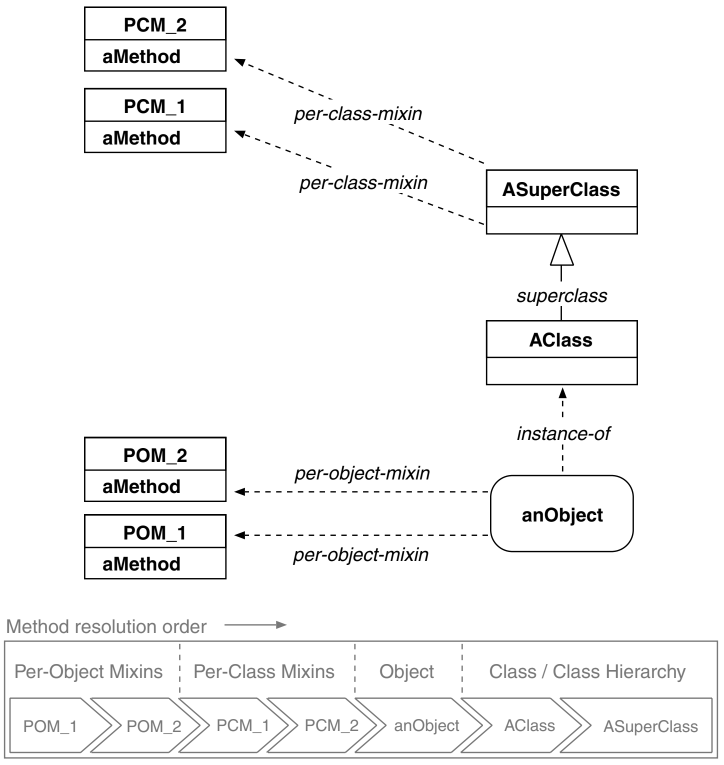


Fig. 11. Method resolution order with per-object and per-class mixins.

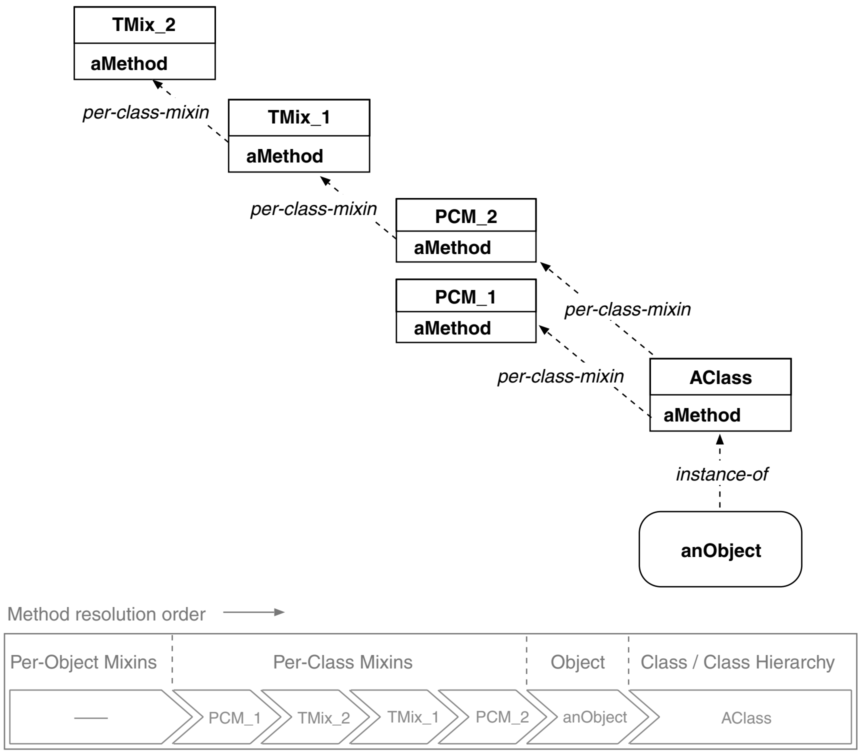


Fig. 12. Example of transitive per-class mixins.

consider a situation where we would like to configure the composition of a per-class mixin PCM_2 with a facet implemented in a class TMix_1,[5] while the original composition of mixins should stay unaffected. Moreover, TMix_1 itself should be refined by another mixin TMix_2 (see Fig. 12).

In XOTcl this is solved by adding the corresponding per-class mixins to the method resolution order of the affected object. This means that all per-class mixins of the mixin itself (and their superclasses) are searched before the method resolution order proceeds to the next mixin, resulting in a *transitive mixin chain* (see also Fig. 12). This scheme is applied recursively, because mixins might themselves have per-class mixins, which again might have per-class mixins, and so on.

In a transitive mixin chain the original target object of a method invocation – in XOTcl referred to as self – does not change. This means that a referral to self from any mixin contained in a specific transitive mixin chain refers to the target object of the original method invocation (for the example shown in Fig. 12 self would always refer to anObject).

Fig. 12 also shows the method resolution order resulting from a aMethod invocation to an object anObject which has two mixins (PCM_1 and PCM_2) registered as per-class mixins on its class AClass. Moreover, the per-class mixin PCM_2 has itself a per-class mixin TMix_1, and TMix_1 is again extended with another per-class mixin TMix_2.

[5] We use "TMix" as an abbreviation for "transitive mixin" in this example.

4.5. Transitive mixin delegation in XOTcl

To realize transitive mixin delegation, per-object mixins are used. As explained in Section 3, transitive mixin delegation is applied when the self reference should change in case a per-object mixin is applied. In XOTcl, this is solved by delegating the mixin invocation to the class object of the respective per-object mixin. Classes in XOTcl are objects with all object-specific characteristics (see also [29]). Thus, at runtime, a class can be treated as an instance (i.e., as an individual object). Class objects are defined using a special type of class, a so-called meta-class. In XOTcl, all objects need to have a class. A meta-class is a special kind of class whose instances are (ordinary) classes.

Meta-classes (see also [12]) are only one of many possible concepts to define the properties of classes. Other concepts that might be used equivalently are meta-object protocols (MOPs) [19], aspect-oriented programming [21], or patterns like Object System Layer [14] or Type Object [18]. To fully realize the concept of transitive mixin delegation, we still need to enable the transitive application of the mixin delegation relationship for the respective target object. For instance, in Fig. 13, the per-object mixin relationship of a class Mix2 to a class Mix1 refers to the corresponding target Mix1 only, and not to the object anObject. Thus, in the example in Fig. 13, an invocation to anObject is intercepted and automatically forwarded to Mix1, however it is not transitively sent to Mix2.

We solve this problem using a simple and automatically generated *delegator method*. This delegator method realizes

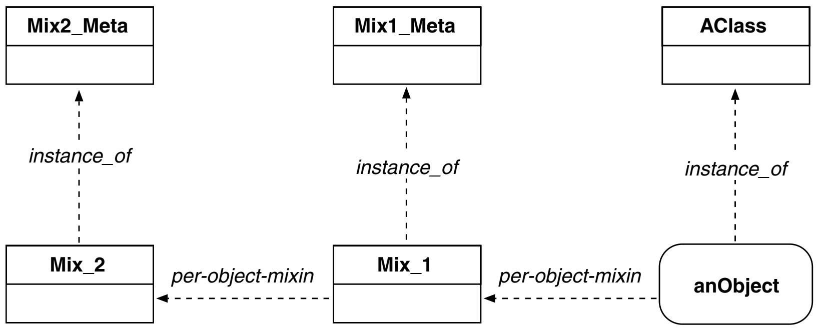


Fig. 13. Per-object mixins which are themselves configured using per-object mixins.

the *transitive_mixin_delegation* relationship as defined in Section 3.3. The resulting delegation behavior is defined by the meta-class.

Fig. 14 depicts the method resolution order resulting from the object-specific transitive mixin delegation in more detail. In order to forward a call of `aMethod` invoked on `anObject` from class `Mix_1` to its per-object mixin `Mix_2` (or to another transitive mixin), we need to invoke a corresponding method in `Mix_1` which is again forwarded along its own linearized method resolution order. Therefore, each mixin class (`Mix_1`,..., `Mix_n`) implements a *delegator method* for `aMethod` (or for any other method which should be forwarded during the method resolution using `next`). This delegator method simply forwards the call to the meta-class (which implements the respective method).

To realize this concept in XOTcl we define, in a first step, a meta-class (in XOTcl this is done by specifying `Class` as superclass) and implement `aMethod` on this class:

```
Class Mix1_Meta -superclass Class
Mix1_Meta instproc aMethod args {
  ### code of aMethod
  ...
}
```

The methods implemented on a meta-class define the methods applicable on all individual classes that are instantiated from this meta-class. We automatically generate a delegator method on the mixin classes (i.e., the meta-class' instances) for each method of the meta-class that should be (transitively) available to the objects which are associated with the respective mixin classes at runtime. Typically this is done in the constructor of the meta-class. The following code snippet contains a meta-class constructor (the `init` method) of the `Mix1_Meta` class that generates a delegator method `aMethod`:

```
Mix1_Meta instproc init args {
  next
  [self] instproc aMethod args {
    eval [self class] aMethod $args
    return [next]
  }
}
```

When instantiating `Mix1_Meta` the `[self]` call in `[self] instproc aMethod args` is replaced with the name of the new `Mix_1Meta` instance (see also Fig. 14). Within the delegator method, the `[self class]` call is replaced at runtime with the name of the mixin class instance. That means, when `aMethod` is invoked on `anObject`, the invocation is intercepted by the delegator method on `Mix_1` which then invokes the implementation on `Mix_1`'s class that is defined in `Mix1_Meta`.

At this point, the scheme described above gets automatically applied in a transitive fashion: before the invocation of `aMethod` is executed for `Mix_1`, all per-object mixins of `Mix_1` are invoked. As the same scheme is executed on `Mix_2`, `Mix_3`, and so on (see Fig. 14), a call of `aMethod` on `anObject` automatically invokes all (direct and transitive) per-object mixins associated with `anObject`.

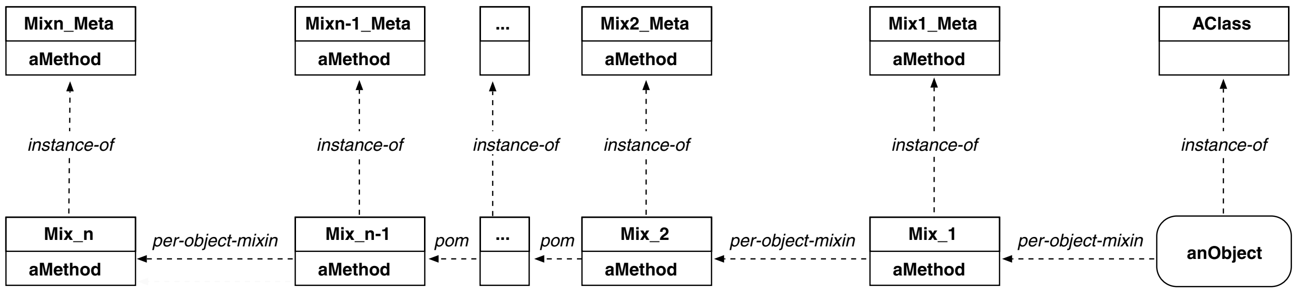
Even though this recursive scheme might look quite complex at first glance, the use of this mechanism is relatively simple (see the case study in Section 5.2 for instance). The developer only has to define the delegator method template.

Fig. 15 shows an example of the typical developer perspective on transitive mixin delegation. A number of arbitrary class hierarchies can be composed, and all inter-class hierarchy composition issues are automatically handled using transitive mixin delegation in conjunction with the `next` mechanism. In particular, this means the programmer is released from implementing a method/mixin lookup procedure on her own. Fig. 15 shows a characteristic example of a resulting method resolution order. In this example, an `aMethod` invocation is forwarded from `Mix_1` to the last mixin class in the transitive mixin chain (here: `Mix_n`). Subsequently, the respective invocation follows the method resolution order shown in Fig. 15 before it finally reaches `aMethod` provided through `AClass`.

5. Case studies

5.1. Transitive mixin chains: configuring a persistent storage

In XOTcl, every object can be made persistent using a simple API. In essence, this persistence property is added using a mixin class. For instance, the following code adds the eager persistence strategy to an object (here "eager" means that changes of variable values are directly written



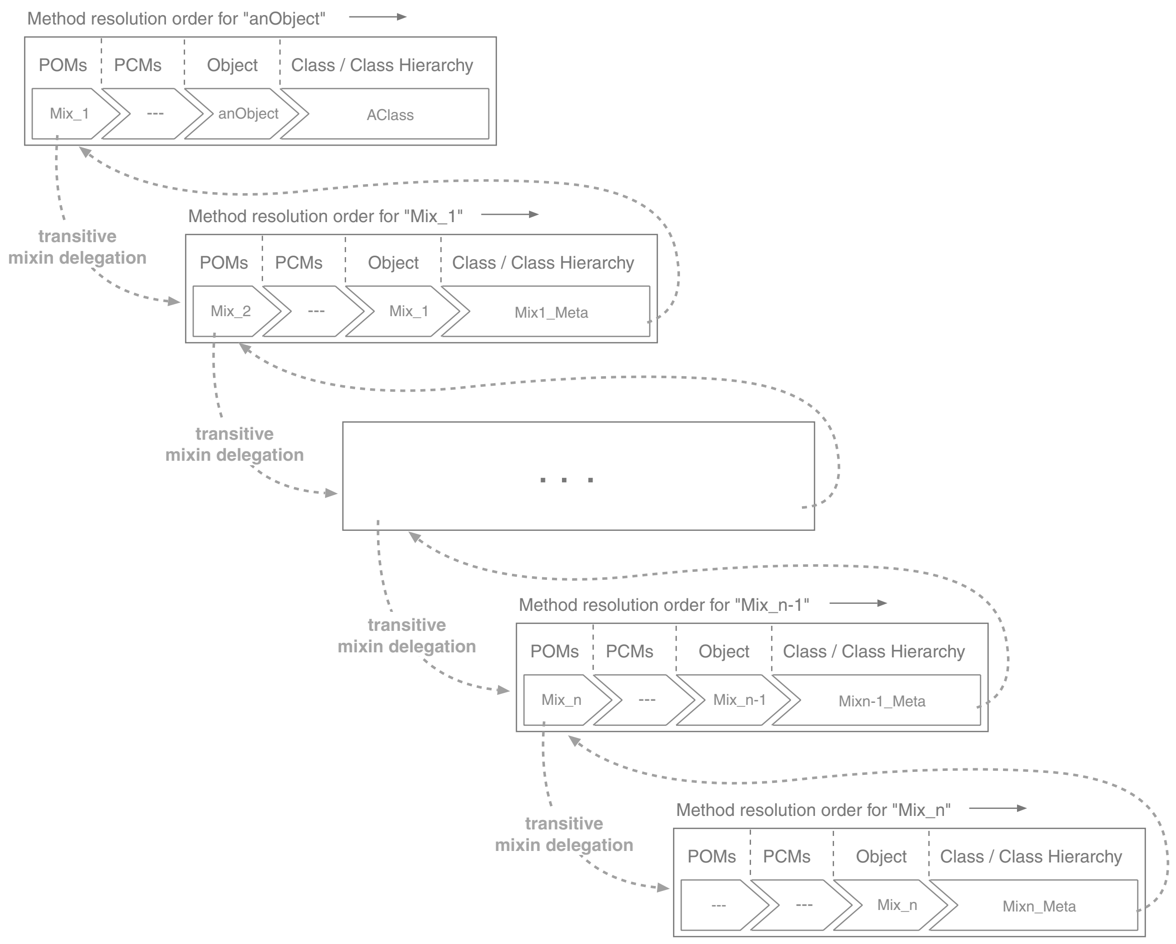

Method resolution order for "anObject"
POMs
PCMs
Object
Class / Class Hierarchy
Mix_1
---
anObject
AClass
transitive mixin delegation
Method resolution order for "Mix_1"
Mix_2
Mix_1
Mix1_Meta
. . .
Method resolution order for "Mix_n-1"
Mix_n
Mix_n-1
Mixn-1_Meta
Method resolution order for "Mix_n"
Mixn_Meta

Fig. 14. Method resolution order for transitive mixin delegation.

into the persistence store. XOTcl also implements a lazy persistence strategy):

```
anObject mixin PersistentEager
```

Using a per-class mixin we can add the same functionality to a class. Thus, the persistence mixin is applied for all instances of that class, for instance:

```
AClass instmixin PersistentEager
```

However, in this situation we face the problem that the persistence relationship needs to be further configured and refined: the persistence storage type has to be chosen and its functionalities need to be accessed. In XOTcl, multiple storage types are supported (a GDBM database, an SDBM database, a memory storage, and a plain file storage). All these storages can be accessed using a unified storage interface.

Thus, to access these storages from `anObject` or instances of `AClass` (which have the persistence logic mixed in using the per-class mixin on `AClass`), we only require the additional storage functionalities. This configuration is a stateless configuration that just adds the storage type behavior. That is, even though there might be multiple objects and classes that are made persistent,

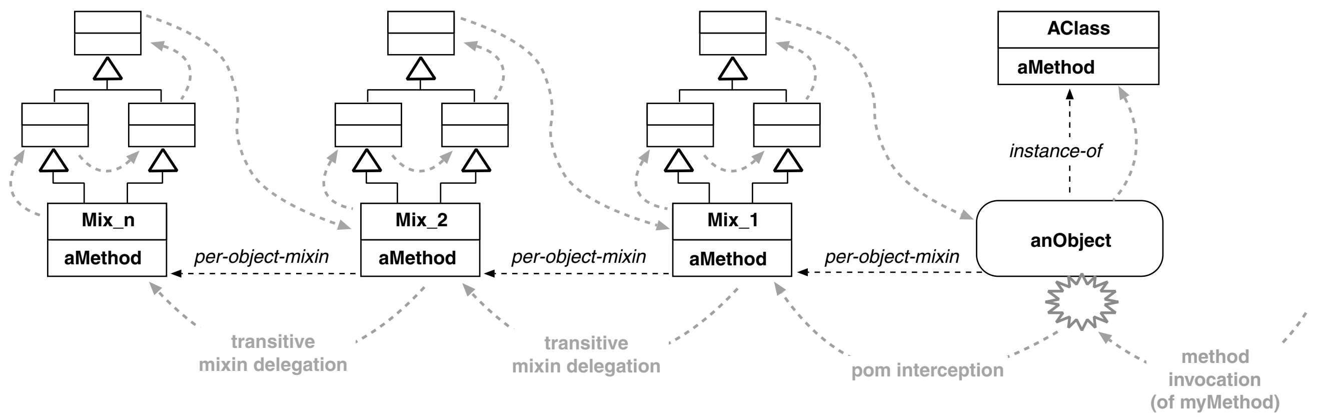


Fig. 15. Method resolution order for transitive mixin delegation with multiple mixin hierarchies.

usually all objects should be written into the same (type of) persistence store. In such cases, it is tedious to configure each object and class on its own. Instead, we can use the transitive mixin chain feature and configure the `PersistentEager` strategy with a certain storage type:

```
PersistentEager instmixin StorageGdbm
```

Now all objects are made persistent (using the eager strategy) and are written into a GDBM persistence store.

A strength of this approach is that it is still possible to further configure and refine persistence for individual objects if needed by an application. We can for instance configure the two mixin compositions above individually, by registering persistence as a second mixin.

```
anObject mixin StorageGdbm
…
AClass instmixin StorageMem
```

The central benefit of using transitive mixin chains for persistence storage configuration is the increased flexibility without compromising reuse or simplicity. Just consider the eight example configurations in Fig. 16, which all can be easily configured, without changes to any of the respective classes (these eight examples are shown for demonstration purposes and do not show all possible configurations). In different design situations, each of these configurations makes sense:

1. One specific object is made persistent with the eager strategy, using the GDBM storage.
2. One specific object is made persistent with the eager strategy, and all objects associated with the eager strategy are written to a GDBM storage.
3. All instances of a class are made persistent with the eager strategy using the GDBM storage.
4. All instances of a class are made persistent with the eager strategy, and all objects written eagerly are written to a GDBM storage.
5. One specific object is made persistent with the eager strategy, and all instances of that class, if they are made persistent, are written to the GDBM storage.
6. All instances of a class are made persistent with the eager strategy. The storage is configured object-specifically: for the example object GDBM is chosen.
7. All instances of a class are made persistent and written to the GDBM storage. All instances written to the GDBM storage are written eagerly.
8. One specific object is made persistent and written to the GDBM storage. All instances written to the GDBM storage are written eagerly.

Using transitive mixin chains the definition of other persistence configurations simply results in a different mixin registration, whereas in many other approaches, some of these variants would mean that internal changes are required to some of the classes. This is just a simple example with two mixins realizing one concern, persistence, configured on one object and one class. The transitive mixin chain works equally well for more behavioral concerns realized by a bigger number of mixins and applied for more complex hierarchies of classes and on arbitrary numbers of objects.

In the persistence example, ordering of the mixin classes does not matter. If the order of mixin classes matters (i.e., with respect to the example: whether `PersistenceEager` or `StorageGDBM` is applied first), then not all example configurations are exchangeable, because they yield different orders of the two mixins. In general, it is a strength of the transitive mixin chain approach that ordering can be controlled by the developer, if this is required.

5.2. Transitive mixin delegation: implementing the xoRBAC component

xoRBAC [27,28] is a software component that provides a role-based access control (RBAC) service. xoRBAC is implemented in XOTcl and, among other things, uses per-object mixins to implement the `checkAccess` method which renders xoRBAC access control decisions. We have

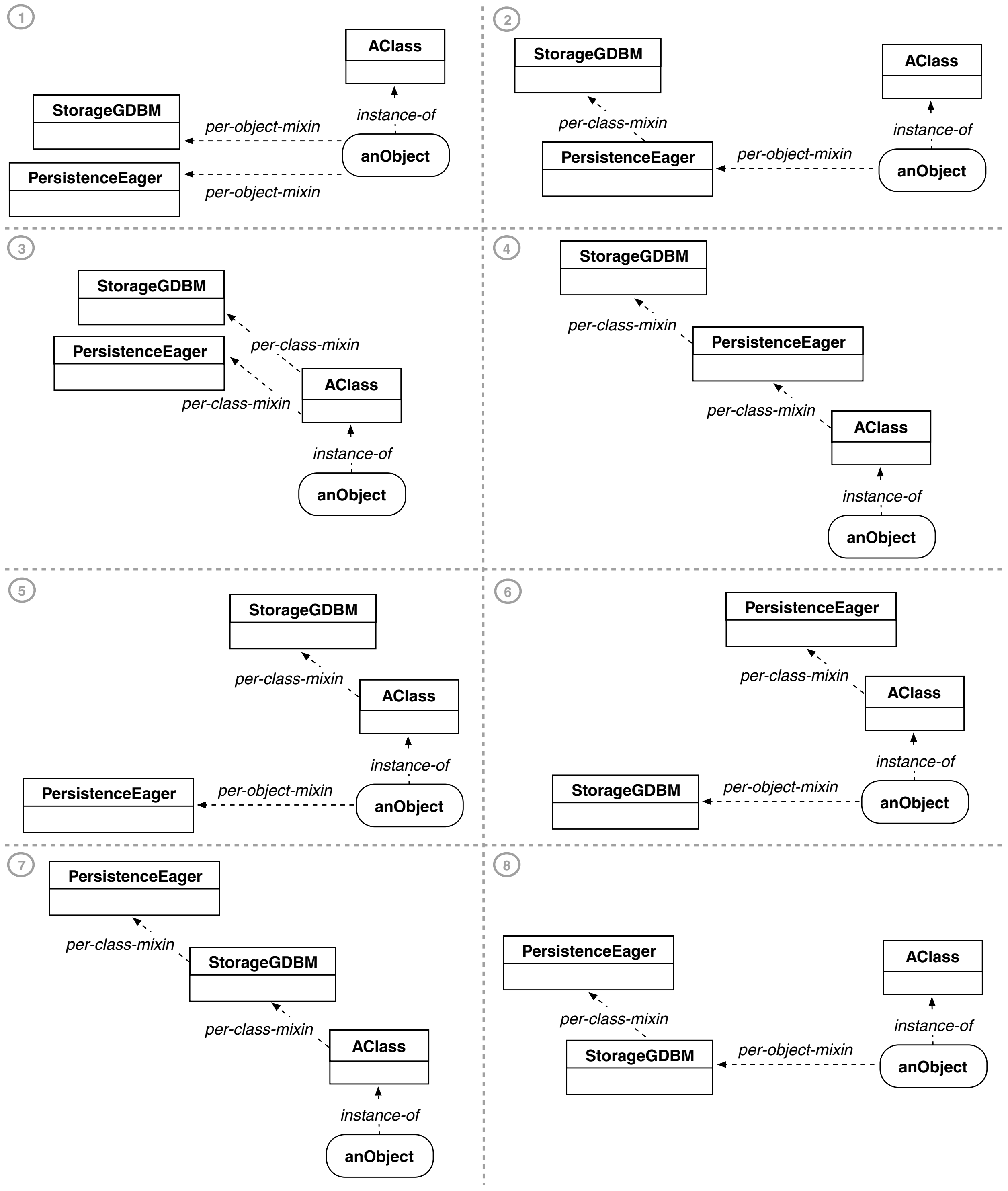


Fig. 16. Example configurations of two persistence mixin classes.

applied the transitive mixin delegation feature of XOTcl to facilitate the implementation of role-, permission-, and constraint-lookup procedures.

Fig. 17 depicts the high-level relations between xoRBAC objects: permissions are assigned to roles, roles are assigned to subjects, and roles may be arranged in a role-hierarchy (a directed acyclic graph). Furthermore, xoRBAC allows for the definition of context constraints [37]. A context constraint specifies a number of conditions that must hold simultaneously to grant a certain access request. On the implementation level, we use per-object mixins to associate subjects with roles, roles with permissions, and permissions with context constraints.

An example for the method resolution order of a `checkAccess` call is shown in Fig. 18. Here, `permission1`

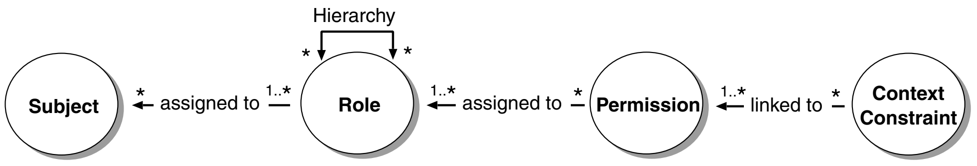


Fig. 17. High-level relations between xoRBAC objects.

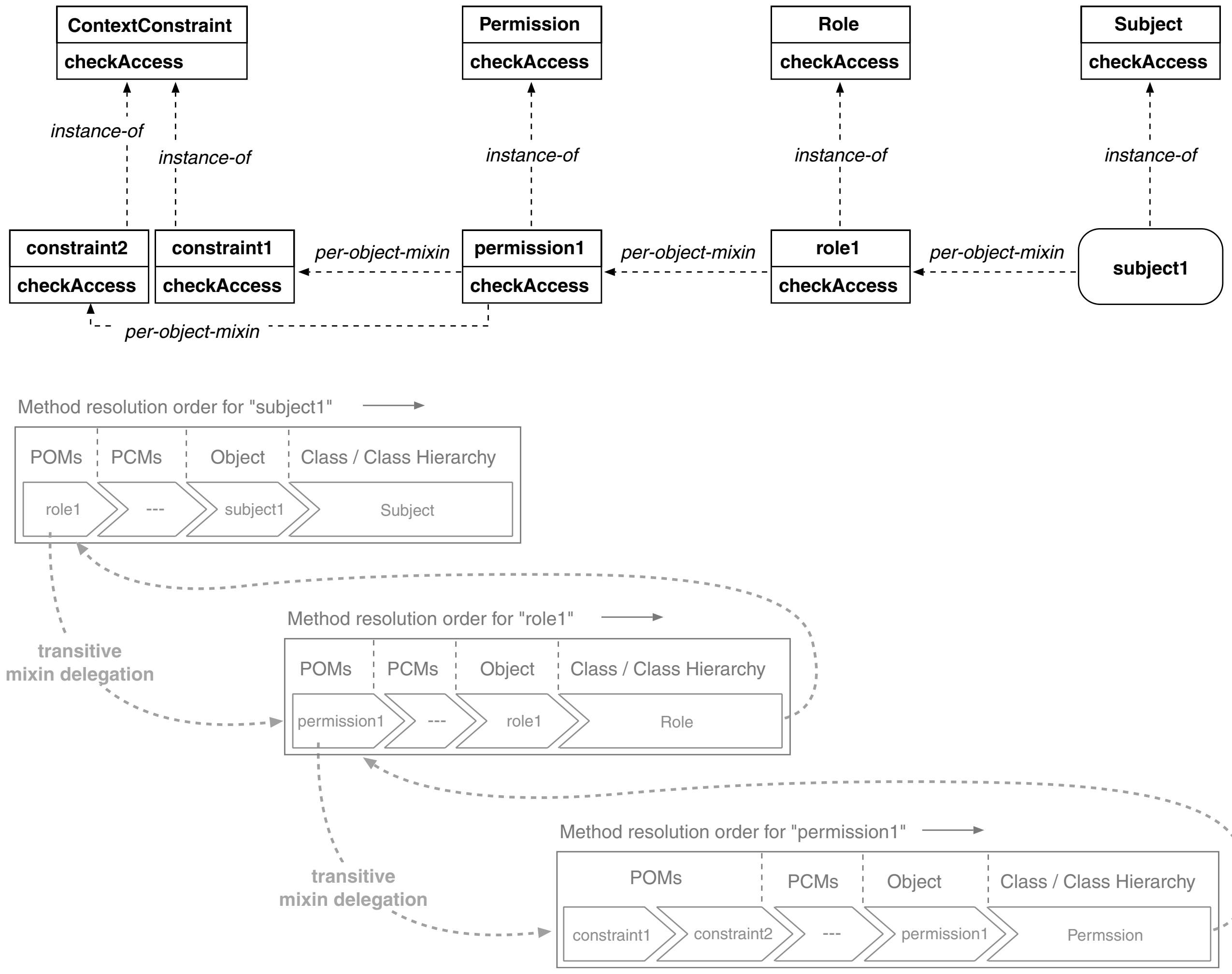


Fig. 18. Method resolution order for the call of checkAccess.

is assigned to `role1`, and `role1` is assigned to `subject1`. Moreover, `permission1` is linked to two context constraints `constraint1` and `constraint2`. Regarding the `checkAccess` method, the instances of `Role` and `Permission` (and `ContextConstraint`) form a Chain of Responsibility [13]. Thus, a `checkAccess` call is passed via the method resolution order until a `Permission` object declares itself responsible and grants the access request by returning `true`. If, however, the respective permission is associated with one or more context constraints (as in Fig. 18 for example), the permission must check its context constraints first. In other words, to grant a certain access request it is not sufficient for a subject to own a corresponding permission, but, at the same time, all context constraints associated with this permission must be fulfilled.

We chose transitive mixin delegation to associate context constraints with permissions, permissions with roles, and roles with subjects. The source code for the automatic generation of the `checkAccess` delegator method for `Role` objects is shown below. An access request is represented by the triple ⟨*Subject, Operation, Object*⟩ which again is represented through the `su op ob` parameters passed to the `checkAccess` method (the source code of the respective delegator method for `Permission` objects is quite similar, while context constraints, in contrast, return `false` if the constraint is violated and forward the call using `next` instead of returning `true` if the constraint is fulfilled).

```
Role instproc init args {
  next
  [self] instproc checkAccess {su op ob} {
    if {[[self class] checkAccess \
        $su $op $ob]} {
      return 1
    }else {
      return [next]
    }
  }
}
```

Fig. 19 shows a sequence diagram for the return of `false` (`checkAccess` returns `false` if the corresponding access request cannot be granted). The sequence diagram thus provides an alternative view of the action and event sequence resulting from a `checkAccess` call (see also Fig. 18).

Transitive mixin delegation offers a number of advantages in this case. One of the most important benefits is that the unambiguous method resolution order of the `checkAccess` method always includes all roles, permissions, and context constraints which are registered as per-object mixins on a specific `Subject` (directly as well as transitively). Thereby, xoRBAC does not need to implement separate lookup-methods for roles, permissions, or context constraints. Rather, a `checkAccess` method invocation follows the method resolution order to automatically visit all roles, permissions, and context constraints which are (potentially) relevant to the corresponding access request.

6. Evaluation

Our approach has a number of unique properties, compared to the other approaches discussed in Section 2. The main contribution of our approach is a clear concept for the transitive composition of mixins. This way we can express extensions to a class, superclass, or mixin using one and the same reusable programming technique: the transitive mixin class. From a conceptual point of view, mixin roles [40] and mixin layers [35] are heading to a similar direction as they also provide some additional composition mechanism using the mixin concept. However, the realization using static C++ templates is completely different and not well suited for expressing dynamic mixin interdependencies.

Even though some approaches, such as AOP and role concepts, can express class interdependencies quite well, it is usually difficult to apply these concepts transitively – like for instance "an aspect of an aspect". Aspects of aspects are only realized by a few prototypes, such as Hyper/J or EAOP. Our approach especially adds a clear precedence order that helps to easier understand aspect interdependencies. As aspects can be used to realize mixins, our concepts for transitive mixin composition can also be used as a concept to add transitivity to the other AOP approaches.

In a similar way, our approach can be used to extend role concepts with the notion of transitivity. Kristensen and Østerbye [23,24] have proposed a notion of "roles of roles" before. Nevertheless, as explained in Section 2, in their approach changes to clients are necessary to acquire the

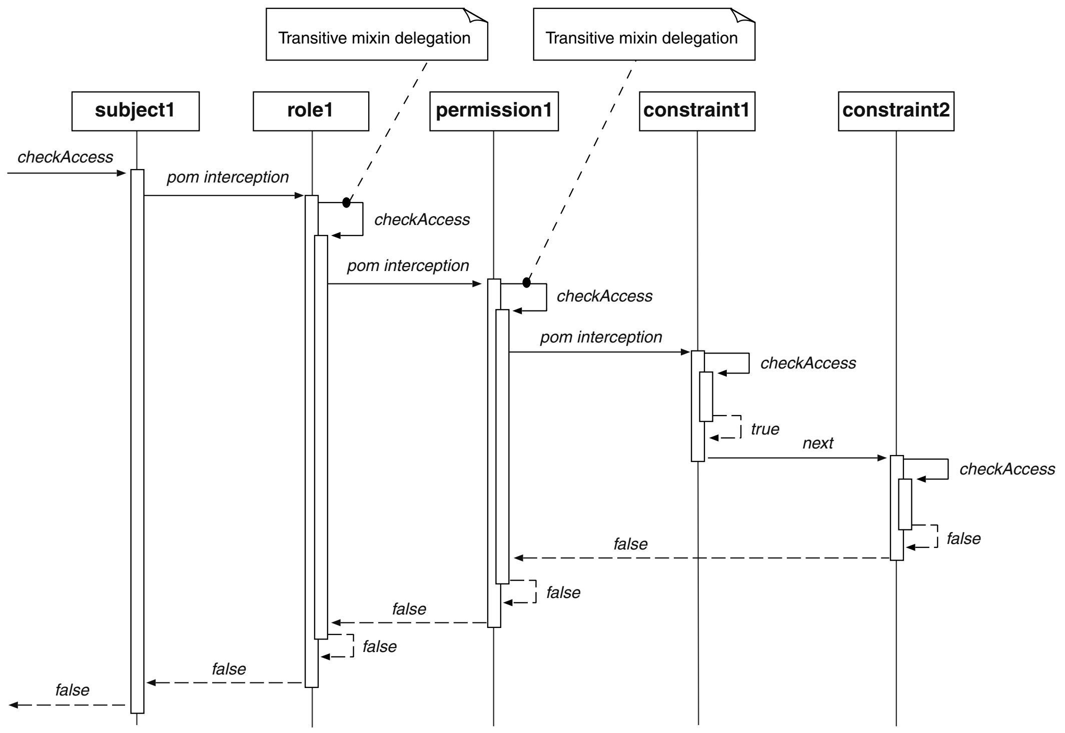


Fig. 19. Sequence diagram of a checkAccess call for the return of false.

mixin behavior. Our approach, in contrast, transparently composes transitive mixins. Thus, our approach is more suited for unanticipated evolution and reuse.

A similar problem occurs in Zhao's and Foster's approach [46]: manual forwarding through the Cascade hierarchy is required to compose Cascade layers (see also Section 2) to achieve the same effect as offered by our transitive mixin concepts. In our concept, automatic composition via the next-primitive and a linear precedence order are supported.

Object Teams [17] support automated method forwarding for method bindings between a class and its roles. That is, regarding transparent composition support, this concept is closer to our transitive mixins than the mentioned role approaches. However, Object Teams do not support transitive roles.

Our approach introduces one and the same construct for direct and transitive composition: the mixin class. Many of the related approaches introduce different constructs for mixin (or, for example, aspect/role/meta-object) and class. Thus, in our approach, developers only have to learn a single language feature to perform all kinds of composition. Only a few additional (implementation-dependent) facets have to be understood. Any class can be used as a mixin class through registration – without further modification of that class. This is supported by an automatic forwarding mechanism that also handles type conversions and argument passing transparently, the *method resolution order*. This results in a simple, unambiguous ordering scheme.

In AOP approaches, like AspectJ or JBoss AOP, mixins often need to be introduced as inter-type declarations. In contrast, our approach directly applies mixins as message interceptors (see also Section 4). As virtually all aspect composition frameworks support some kind of message interceptor (see [42] for a discussion) and some automatic forwarding mechanism (such as AspectJ's "proceed" [20] for instance), mixin classes can be realized using most AOP approaches with moderate efforts.

A major composition problem in many AOP approaches is the so-called fragile pointcut problem [22]. Many pointcuts have dependencies into the base program. Hence simple changes like renaming a method in the base program can break the pointcut. This problem is only a minor problem in our prototype, because we use explicit mixin registration on classes. Hence there is a direct relation between the mixin class and the base program that it extends. Only changes to base class names or method names that are intercepted by the mixin can potentially be the cause of a fragile mixin composition. In most cases, such a change directly causes an error (and can thereby easily be detected). If AOP approaches are used to implement our approach, however, fragile pointcuts might cause massive debugging problems, because transitive mixin composition leads to complex aspect interdependencies which might get hard to understand and trace if arbitrary pointcuts can be used. Our solution in the Frag prototype [44,45], which uses AspectJ to compose Frag mixin classes with Java base classes, is to use only simple, explicit pointcuts that are limited to the expressive power of mixin registration (see [43] for details). To ensure that this limitation is not violated by developers, it is advisable to use a program generator to automatically create the pointcuts.

In contrast to the model used in more dynamic object-oriented environments, such as CLOS, Smalltalk, or Self, as well as in programming techniques such as reflection, MOP, or meta-classes, transitive mixins provide a first-class entity for expressing the interdependencies of mixins. Method invocations on mixins are always resolved in an unambiguous, linear method resolution order – instead of a complex graph of meta-objects or other delegators with different responsibilities. Our mixins provide a very simple interface allowing for registration and introspection of mixins only. Thus, compared to more complex approaches such as MOPs they are very simple.

When configuring elements of an object-oriented software system, it is often not enough to provide configuration options on a per-class level. Many object-oriented adaptation techniques, however, perform adaptations on a per-class level only, making it cumbersome to apply these techniques for object-specific composition. On the other hand, when class-specific composition is required, having only an object-specific configuration option is tedious as well. Thus, our approach supports both variants: it can be applied using per-object and per-class mixins.

Some of the approaches discussed in Section 2 are static composition techniques meaning that the core composition mechanism cannot be used for dynamic composition. For instance, AOP approaches, like AspectJ or JBoss AOP, focus on static adaptation techniques. Therefore, in contrast to our approach, they cannot be directly applied for runtime changes of the aspect configuration. There are some workarounds to these problems (for instance, aspects that can be turned on and off using `thisJoinPoint` in AspectJ), but these are hand-built solutions that are not optimized for performance and without further composition support. This problem is resolved by dynamic AOP approaches. Our concept of transitive mixins can be applied in both a static as well as in a dynamic fashion, yet our examples (and prototype implementation) are focused on dynamic mixin configuration. The dynamic AOP approaches are closer to the examples in this paper than more static approaches like AspectJ. To implement our concepts on top of an AOP framework, it is thus advisable to reuse a dynamic AOP framework if possible, because this allows for the reuse of existing dynamic aspect composition means.

A sub-problem of dynamic composition is the dynamic ordering of aspects, which might be needed in some application scenarios. Mixin classes are dynamically composed and the order can be provided at runtime as a mixin list. Our mixin class concepts can also be used as a simple and intuitive conceptual foundation to add dynamics to static approaches.

Nevertheless, our approach is not limited to languages and environments that support mixins. The mixin concept is a rather simple extension of the basic object-oriented type concept and similar concepts can be found in many other adaptation techniques, such as aspect-oriented programming, meta-object protocols, roles, message interceptors, interpreters, virtual machines, etc. Therefore, our approach can be applied on top of those other approaches and usually reuse large parts of their implementation.

In our proof-of-concept implementation we describe the dynamic mixin classes of XOTcl. If dynamic composition is not required (i.e., if compile time or load time approaches are sufficient), the concepts presented in this paper can also be implemented using static mixin approaches. All implementation approaches for static mixins support some of the properties of transitive mixin classes. Essentially, to implement our concepts using one of these approaches, it is necessary to generate delegator methods to simulate the transitive "`next`" behavior and automatic forwarding (including type conversions, parameter adaptation, etc.). For programming languages without support for dynamic method generation, such as Java, many code generators exist that ease this task.

Even though our approach is easy to use and simple from a developer's perspective, the internal use of metaclasses and interceptors is far from being simple (as the discussion in Section 4 indicates). Thus, implementing our approach completely from scratch for another programming language or framework requires some effort that might be too much an effort for a small project.

Runtime composition techniques always impose an overhead in terms of runtime performance (for dynamic indirections). Even though XOTcl message interceptors are optimized for performance, they should not be applied for problems that do not require dynamic adaptations. Here, static techniques usually have a superior performance. However, this is, of course, only a potential drawback of our prototype implementation, not of the transitive mixin concepts in general.

7. Conclusion

In this paper, we have presented a practical approach to model mixin interdependencies. By applying mixin classes transitively, we are able to use the concept of mixin classes to define composition relationships of ordinary classes and mixins. Problems similar to the problem to define "mixins of mixins" are present in many other composition approaches as well – such as in aspect-oriented programming, meta-object protocols, roles, message interceptors etc. Hence, there is a broad applicability of the transitive mixin approach. The mixin concept is a rather simple extension to the basic type concepts of object-oriented languages, and is thus well suited to explore the problems of class relationships and interdependencies generally and conceptually – apart from the implementation details of the other composition approaches. We did two proof-of-concept implementations, XOTcl and Frag, which are both available as open source.

Moreover, our mixin concepts have been successfully applied in a number of projects (including the two case studies presented in this paper). As future work, we plan to implement the concepts as an extension of an existing AOP framework. In this paper, we focused on the extension of programming frameworks or languages by transitive mixin classes. As further work we also plan to provide modeling support for the concepts presented in this paper, for instance using a UML 2 extension.

References

[1] A. Albano, R. Bergamini, G. Ghelli, R. Orsini, An Object Data Model with Roles, in: Proceedings of the 19th International Conference on Very Large Data Bases (VLDB), Morgan Kaufmann Publishers Inc., 1993.

[2] L. Bettini, S. Capecchi, B. Venneri, Extending Java to dynamic object behaviors, in: Proceedings of the Workshop on Object-Oriented Developments (WOOD), Electronic Notes in Theoretical Computer Science (ENTCS), vol. 82, 2003.

[3] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, D.A. Moon, Common Lisp Object System Specification, ACM SIGPlan Notices 23 (SI) (1988).

[4] C. Bockisch, M. Haupt, M. Mezini, K. Ostermann, Virtual Machine Support for Dynamic Join Points, in: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, 2004.

[5] G. Bracha, W. Cook, Mixin-based inheritance, in: Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA), Proceedings of the European Conference on Object-oriented Programming (ECOOP), 1990.

[6] G. Bracha, G. Lindstrom, Modularity meets inheritance, in: Proceedings of IEEE International Conference on Computer Languages, 1992.

[7] B. Burke, JBoss Aspect Oriented Programming, <http://labs.jboss.com/portal/jbossaop/>, 2006.

[8] W.R. Cook, W. Hill, P.S. Canning, Inheritance is not subtyping, in: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 1990.

[9] M. DeJong, S. Redman, The Tcl/Java Project, <http://tcljava.sourceforge.net/>, 2006.

[10] R. Douence, M. Suedholt, A model and a tool for Event-based Aspect-Oriented Programming (EAOP). TR 02/11/INFO, Ecole des Mines de Nantes, french version accepted at LMO'03, second ed., December 2002.

[11] M. Flatt, S. Krishnamurthi, M. Felleisen, Classes and mixins, in: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), January 1998.

[12] I.R. Forman, S.H. Danforth, Putting Metaclasses to Work – A new Dimension to Object-Oriented Programming, Addison-Wesley, 1999.

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[14] M. Goedicke, G. Neumann, U. Zdun, Object system layer, in: Proceedings of the European Conference on Pattern Languages of Programs (EuroPlop), July 2000.

[15] A. Goldberg, D. Robson, Smalltalk-80: The Language, Addison-Wesley Longman Publishing, 1989.

[16] G. Gottlob, M. Schrefl, B. Röck, Extending object-oriented systems with roles, ACM Transactions on Information Systems 14 (3) (1996).

[17] S. Herrmann, Sustainable architectures by combining flexibility and strictness in Object Teams, IEE Proceedings Software 151 (2) (2004).

[18] R. Johnson, B. Woolf, Type object, in: R. Martin, D. Riehle, F. Buschmann (Eds.), Pattern Languages of Program Design 3, Addison-Wesley, 1998.

[19] G. Kiczales, J. des Rivieres, D. Bobrow, The Art of the Metaobject Protocol, MIT Press, 1991.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, Getting started with AspectJ, Communications of the ACM 44 (10) (2001).

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science (LNCS), vol. 1241, Springer-Verlag, June 1997.

[22] C. Koppen, M. Störzer, PCDiff: attacking the fragile pointcut problem, in: Proceedings of the European Interactive Workshop on Aspects in Software (EIWAS), September 2004.

[23] B. Kristensen, Object-Oriented Modeling with Roles, in: Proceedings of the International Conference on Object-Oriented Information Systems, Springer-Verlag, 1996.

[24] B. Kristensen, K. Østerbye, Roles: conceptual abstraction theory & practical language issues, Theory and Practice of Object Systems 2 (3) (1996).

[25] P. Maes, Concepts and experiments in computational reflection, ACM SIGPLAN Notices 22 (12) (1987).

[26] D. Moon, Object-oriented programming with flavors, in: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), SIGPLAN Notices, vol. 21, Portland, November 1986.

[27] G. Neumann, M. Strembeck, Design and implementation of a flexible RBAC-service in an object-oriented scripting language, in: Proceedings of the 8th ACM Conference on Computer and Communications Security (CCS), November 2001.

[28] G. Neumann, M. Strembeck, An approach to engineer and enforce context constraints in an RBAC environment, in: Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT), June 2003.

[29] G. Neumann, U. Zdun, XOTcl, an object-oriented scripting language, in: Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference, February 2000.

[30] G. Neumann, U. Zdun, XOTcl Homepage, <http://www.xotcl.org/>, 2006.

[31] J.K. Ousterhout, Tcl: an embeddable command language, in: Proceedings of the 1990 Winter USENIX Conference, January 1990.

[32] B. Pernici, Objects with Roles, in: Proceedings of the Conference on Office Information Systems, ACM Press, 1990.

[33] A. Popovici, T. Gross, G. Alonso, Just in time aspects: efficient dynamic weaving for Java, in: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, 2003.

[34] N. Schärli, S. Ducasse, O. Nierstrasz, A. Black, Traits: composable units of behavior, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science (LNCS), vol. 2743, Springer-Verlag, 2003.

[35] Y. Smaragdakis, D. Batory, Implementing layered designs with mixin layers, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science (LNCS), vol. 1445, Springer-Verlag, 1998.

[36] B. Smith, Reflection and semantics in lisp, in: Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), January 1984.

[37] M. Strembeck, G. Neumann, An integrated approach to engineer and enforce context constraints in RBAC environments, ACM Transactions on Information and System Security (TISSEC) 7 (3) (2004).

[38] P. Tarr. Hyper/J, <http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm/>, 2006.

[39] D. Ungar, R.B. Smith, Self: the power of simplicity, in: Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA), October 1987.

[40] M. VanHilst, D. Notkin, Using role components in implement collaboration-based designs, in: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1996.

[41] D. Wetherall, C.J. Lindblad, Extending Tcl for dynamic object-oriented programming, in: Proceedings of the USENIX Tcl/Tk Workshop, July 1995.

[42] U. Zdun, Pattern language for the design of aspect languages and aspect composition frameworks, IEE Proceedings Software 151 (2) (2004).

[43] U. Zdun, Using split objects for maintenance and reengineering tasks, in: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), March 2004.

[44] U. Zdun, Frag, <http://frag.sourceforge.net/>, 2006.

[45] U. Zdun, Tailorable Language for behavioral composition and configuration of software components, Computer Languages, Systems and Structures: An International Journal 32 (1) (2006).

[46] L. Zhao, T. Foster, Modeling roles with cascade, IEEE Software 16 (5) (1999).