

# OCL Component Invariants<sup>\*</sup>

Hubert Baumeister

Rolf Hennicker

Alexander Knapp

Martin Wirsing

Ludwig–Maximilians–Universität München

{baumeist, hennicke, knapp, wirsing}@informatik.uni-muenchen.de

## Abstract

*The “Object Constraint Language” (OCL) offers a formal notation for constraining model elements in UML diagrams. OCL consists of a navigational expression language which, for instance, can be used to state invariants and pre- and post-conditions in class diagrams. We discuss some problems in ensuring non-local, navigating OCL class invariants, as for bidirectional associations, in programming language implementations of UML diagrams, like in Java. As a remedy, we propose a component-based system specification method for using OCL constraints, distinguishing between global component invariants and local class invariants.*

## 1 Introduction

During the last years the “Unified Modeling Language” (UML [2]) has become the de facto standard for object-oriented software development. The “Object Constraint Language” (OCL [14]) offers a formal notation to constrain the interpretation of model elements occurring in UML diagrams and therefore lends itself for systematic use in rigorous, UML-based software development methods, as shown, for example, in the Catalysis approach [5].

The OCL notation is particularly suited to constrain class diagrams since OCL expressions allow one to navigate along associations and to describe conditions on object states in class invariants and pre- and post-conditions of operations. However, by using the ability of describing navigational paths, a class invariant may be non-local in the sense that it also requires properties from other “remote” classes. This expressiveness and flexibility is appropriate in requirements specifications where the developer generally prefers a global view of the properties of the relationships between different classes. For design and implementation, however, such global requirements can be harmful since the implementation of a “remote” class would have to respect the non-local invariant of another class which is not mentioned anywhere in the “remote” class. Thus a programmer may not only have to check the validity of the invariant of the class he is implementing, but also the validity of invariants of other classes.

We first illustrate these problems with non-local class-based OCL invariants by simple examples, including the

conventional use of “setter” operations and, more interestingly, standard OCL formalizations and Java implementations of bidirectional associations. As a remedy, we propose a component-based approach which has the following two properties: it allows us to write non-local invariants at the global level of components instead of at the local level of classes and it allows us to control the visibility of operations. An operation can be *component public* and therefore visible for all classes *inside and outside the component*; or an operation can be *component private* and therefore visible for all classes *inside the component*; or an operation can be *class private* and therefore visible only for its *own class*. Non-local invariants have to be respected only by component public operations; local invariants have to be respected by component public and component private operations; class private operations do not have to respect any invariant. However, for simplicity, we omit component hierarchy aspects and inheritance between different components.

In Sect. 2 we describe the problems with non-local class-based OCL invariants. In Sect. 3 we propose our component-based approach. In Sect. 4 we discuss how components can be realized in Java and we show some properties of correct realizations in Sect. 5. Throughout the paper we assume that the reader is familiar with UML class diagrams and the OCL notation.

## 2 Non-Local Class Invariants

For exhibiting the problems with non-local class-based invariants, we model a simple seminar system inspired by a similar example in the Catalysis book [5, Sect. 2.5.1, p. 67], see Fig. 1. In this system a course consists of several sessions. Each session has at most one instructor and each instructor may be qualified for several courses. Each session has a start and an end time. A session may also be public, where external participants have to pay a certain amount in order to be admitted. There are three invariants: the simple invariant for the class `Session` requires that the start time is before the end time; the invariant for class `PublicSession` states that the price for such a session is at least 10\$. The invariant for the class `Instructor` requires that an instructor should only teach sessions for courses he is qualified for. The class `Session` shows an initialization operation `setup` setting default start and end times for a session; this operation is overridden in `PublicSession`.

<sup>\*</sup>Partially supported by the DFG project InOpSys, ref. WI 841/6-1.

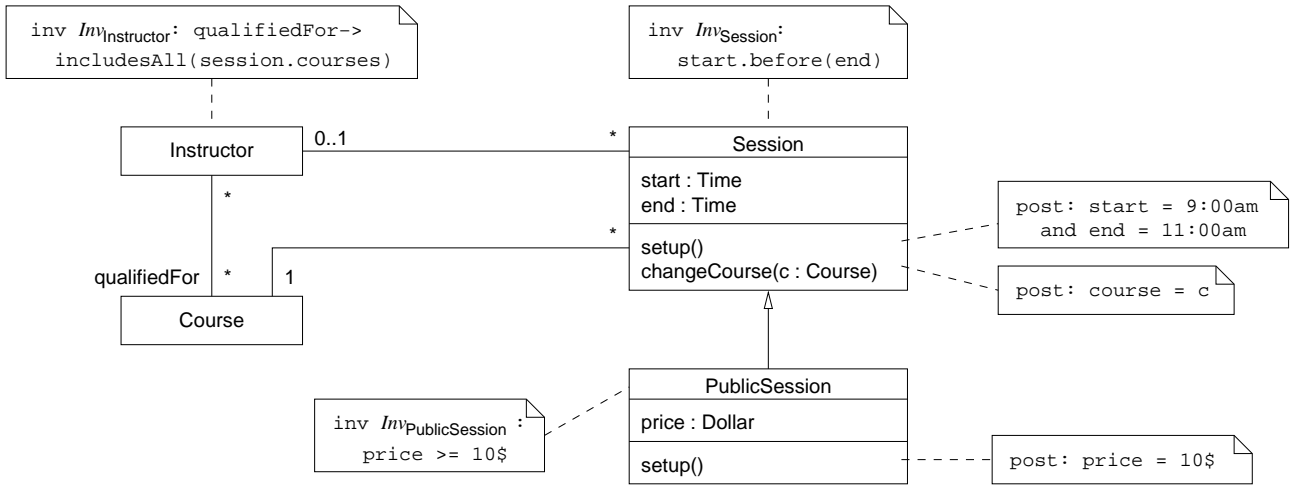


Figure 1: Annotated class diagram for the seminar example

sion initializing additionally the price for the public session. Moreover, the class `Session` has a “setter” operation `changeCourse` which allows to assign a new course to a session; the post-condition just requires to reassign the new course to the actual session.

For a correct implementation of this system in Java, one would like to require that any operation  $C :: op(x_1 : D_1, \dots, x_n : D_n)$  of any class  $C$  of the diagram preserves the invariant  $INV_C$ , obtained by the conjunction of the class invariant of  $C$  and the invariants of all its super-classes (if any), and satisfies the pre- and post-condition provided for  $op$ . As formalized in [12], this means that the Hoare formula

$$\{Pre_{C::op} \text{ and } INV_C\} \\ C :: op(x_1 : D_1, \dots, x_n : D_n) \\ \{Post_{C::op} \text{ and } INV_C\}$$

should be valid. In the example, any implementation of the operation `changeCourse` of class `Session` should satisfy the Hoare formula

$$\{\text{start.before(end)}\} \\ \text{Session} :: \text{changeCourse}(c : \text{Course}) \quad (*) \\ \{\text{course} = c \text{ and } \text{start.before(end)}\}$$

(for the implicit requirements of the bidirectional associations see below). The following Java implementation satisfies (\*):

```
void changeCourse(Course c) {
    course = c;
}
```

The problem is that, although `changeCourse` does not involve any attribute or role of class `Instructor`, it may destroy the invariant  $Inv_{Instructor}$  of class `Instructor`, e.g., when being called via  $s.\text{changeCourse}(c)$  for a session  $s$  having instructor  $s.instructor$  who is not qualified for the course  $c$ .

Another problem stems from making explicit the semantic constraints of bidirectional associations by expressing them

in OCL. Consider for example the one-to-many association between the class `Course` and the class `Session`. The semantic constraint requires that any object  $c$  of class `Course` is related to a set of objects of class `Session` in such a way that each of these objects is related to  $c$ ; thus navigating from  $c$  to any object of `Session` and back to class `Course` yields the original object  $c$ . Similarly, the sessions of the course of a `Session` object  $s$  must include the original object  $s$ . In OCL, one may try to formalize this using the following two class invariants of `Session` and `Course`:

```
context Course
inv Inv'_Course :
    self.session->
        forAll(s | s.course = self)

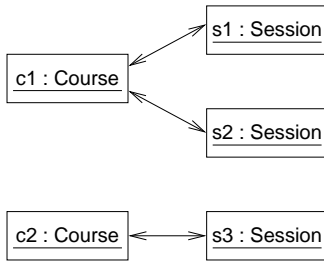
context Session
inv Inv'_Session :
    self.course.session->includes(self)
```

Now consider a system state  $\sigma$  showing two objects  $c_1, c_2$  of class `Course` and three objects  $s_1, s_2, s_3$  of class `Session` such that object  $c_1$  is related with objects  $s_1$  and  $s_2$ , and object  $c_2$  is related with object  $s_3$ , see Fig. 2(a). Obviously, a Java call  $s_2.\text{changeCourse}(c_2)$  does not respect the invariants  $Inv'_{Course}[c_1/self]$  and  $Inv'_{Session}[s_2/self]$ , cf. the object diagram in Fig. 2(b).

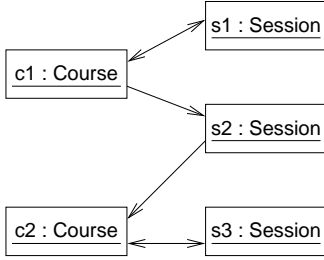
According to Hitz and Kappel [7, Sect. 6.2.1, p. 271–275], a correct Java implementation of `changeCourse` with respect to the bidirectional association can be given using two Java operations `addSession` and `rmSession` in the following way:

```
public class Session {
    private Instructor instructor;
    private Course course;

    public void changeCourse(Course c) {
        if (course != c) {
            if (course != null)
```



(a) State  $\sigma$



(b) State after  $s_2.changeCourse(c_2)$

Figure 2: Sample states of the seminar system

```

        course.rmSession(this);
        course = c;
        c.addSession(this);
    }
}
...
}

public class Course {
    private Vector session;

    public void addSession(Session s) {
        if (!session.contains(s)) {
            session.addElement(s);
            s.changeCourse(this);
        }
    }

    public void rmSession(Session s) {
        session.removeElement(s);
    }

    ...
}

```

The operations `changeCourse` and `addSession` indeed preserve both of the invariants  $Inv'_{Course}$  and  $Inv'_{Session}$  but (as Hitz and Kappel also mention in their book [7]) calling `rmSession` may lead to an illegal state; e.g., see Fig. 3, calling  $c_2.rmSession(s_3)$  in state  $\sigma$  leads to a state where the invariant  $Inv'_{Session}[s_3/self]$  does not hold.

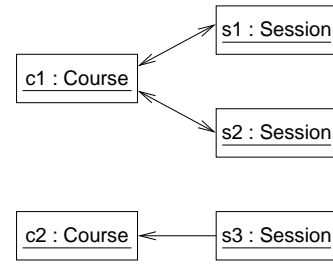


Figure 3: State after  $c_2.rmSession(s_3)$

### 3 Component-Based Invariants

The problems described in the previous section are due to the fact that class invariants of one class are based on properties of objects of another class; in other words, the invariant expressions navigate to objects of other classes. For instance, the invariant  $Inv_{Instructor}$  navigates to the sessions of an instructor and requires that all these sessions are for courses the instructor is qualified for. Hence it is obvious that the class invariant for instructor can easily be destroyed by changing the course of a session. To overcome these problems we use a component-oriented development methodology. Component-based approaches for software development have been advocated by many authors including Broy [3] and Szyperski [13], or, in the context of UML and OCL, by Catalysis [5] and Cheesman and Daniels [4].

We do not propose a new notion of component; almost any of the notions for components in the literature is suitable for our approach provided that a component is composed of classes (and possibly local components) and that the following two requirements are satisfied:

1. It is possible to require invariants globally for the whole component and also locally for the elements of a component.
2. An operation can be declared to be visible either inside and outside the component, or only inside the component, or only inside a single class of the component.

More precisely, we distinguish between class invariants and component invariants: A *class invariant* is an invariant for describing properties concerning a single class (i.e. its attributes and association roles without navigation) and a *component invariant* is an invariant for describing properties concerning two or more classes. For example, the invariant  $Inv_{Instructor}$  of class `Instructor` cannot be used as a class invariant but has to be included in the component invariant for the seminar system (cf. Fig. 4). The invariants induced by bidirectional associations are also included in the component invariant  $Inv_{Seminar}$ . The invariants  $Inv_{Session}$  and  $Inv_{PublicSession}$  of classes `Session` and `PublicSession`, however, are class invariants.

Concerning the visibility of operations we distinguish between operations which are

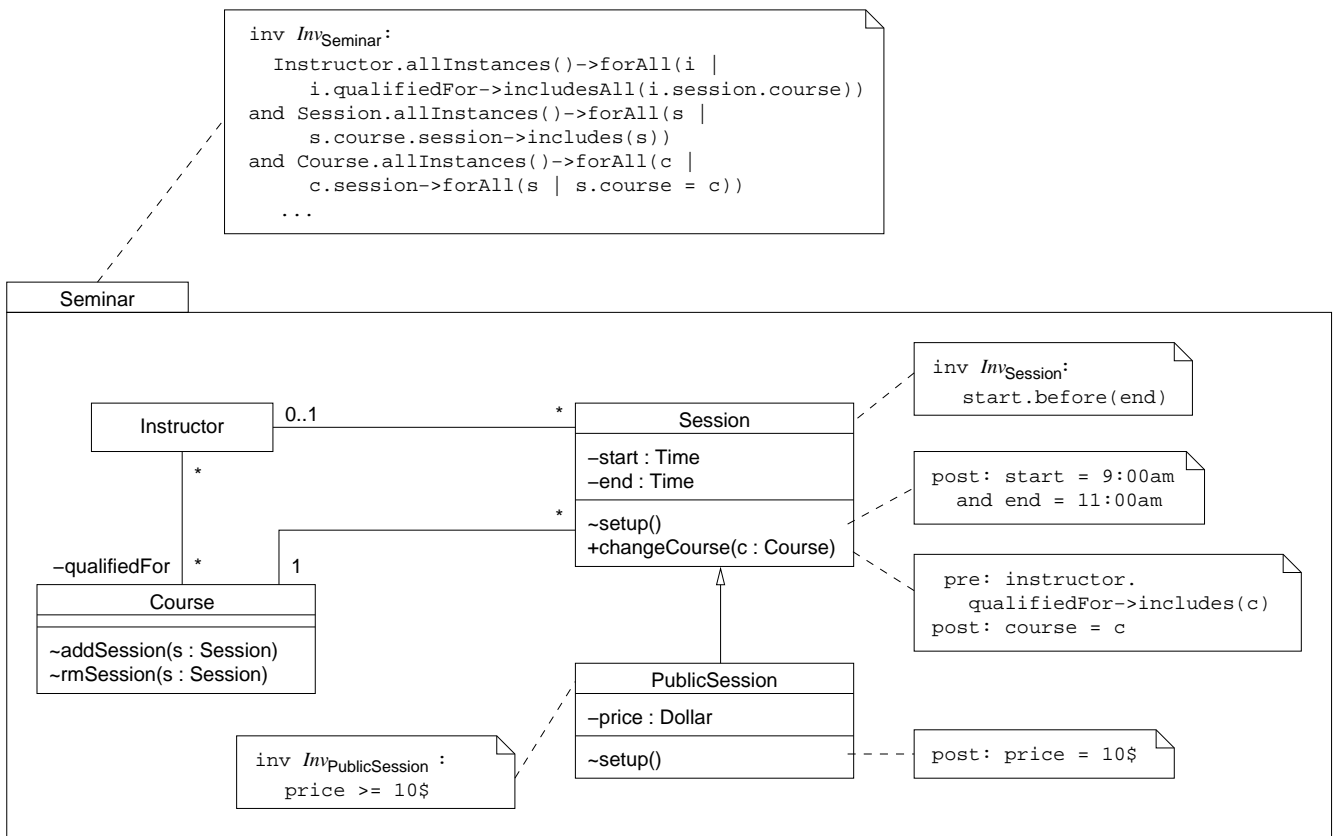


Figure 4: Component model of the seminar system

- component public — visible at the interface of the component
- component private — visible to all classes of the component
- class private — visible to a single class of the component

Each component public operation has to preserve the component invariant, the class invariant where the operation is declared, and the class invariants of all super-classes, and satisfy their pre-/post-condition. Each component private operation has to preserve its class invariant and the class invariants of all super-classes and satisfy its pre-/post-condition. Finally, each class private operation has to satisfy its pre-/post-condition. Since class private operations are auxiliary operations which can be (internally) applied to an object in a “non-stable” state, they need not preserve invariants; cf. also [9].

Concerning the visibility of attributes we assume a good style of design: all attributes have to be visible only to a single class.

Components in the sense of Cheesman and Daniels, Catalysis, or UML subsystems can express these visibility requirements and also show notations for invariants. In particular, due to the explicit notion of interfaces used in Catalysis and by Cheesman and Daniels the visibility of component public operations can be modeled explicitly. A UML subsystem

provides the following visibility correspondences for operations [11]: class private corresponds to private (–), component private to package (~), and component public to public (+). Similarly, an attribute that is only visible to a single class has visibility private (–).

Fig. 4 shows the seminar system as a UML subsystem component. We choose to declare `rmSession` and `addSession` not to be component public but only to be component private. Hence the component invariant  $Inv_{Seminar}$  needs not to be respected by these operations, but has to be respected by the component public operation `changeCourse`. In order to be able to fulfill the post-condition of `changeCourse` and, simultaneously, to ensure the preservation of the component invariant  $Inv_{Seminar}$ , we have to add a pre-condition to `changeCourse`, requiring the replacing course to be compatible with the abilities of the teaching instructor. In fact, the detection of this pre-condition is greatly facilitated by making the component invariant explicit.

## 4 Realization of Components

Based on our notion of component we define a realization relation which connects a UML design component and a Java implementation model. As implementation model for components we use Java packages, classes in a design compo-

nent are mapped to Java classes. However, we put an additional restriction on Java classes: `private` Java attributes and methods are only called on `this` which is good programming practice for encapsulating object states.

We employ UML trace and realization dependencies as considered in [1] to relate UML classes and Java classes, and UML design components and Java packages. A realization relation between a UML design component and a Java package expresses that the implementation model satisfies the requirements of the design model. In particular, the Java methods corresponding to a component public, component private, or class private operations in a class of the UML design component have to preserve the component invariant, the class invariant (and the class invariants of all super-classes), and satisfy its pre-/post-condition according to the requirements described in Sect. 3. Similarly, we have to consider component public, component private, and class private object constructors, where we always require that constructors establish class invariants. Trace dependencies guarantee that the OCL expressions used as constraints for the design model can be interpreted in the implementation model. The constraints on the Java implementation are represented by Hoare formulae and can be proved using the calculus presented in [12].

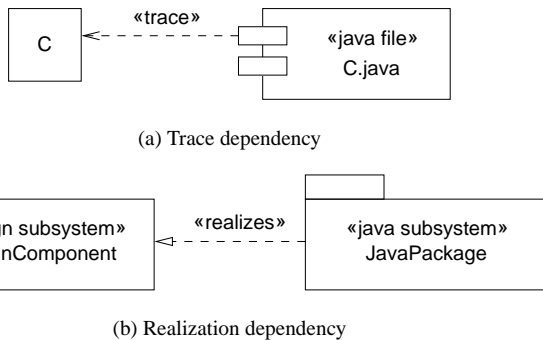


Figure 5: Component dependencies

A trace dependency holds between a UML design class  $C$  and a Java class  $C.java$ , see Fig. 5(a), if the direct super-classes of  $C$  and of  $C.java$  coincide, if the operations of  $C$  and the methods of  $C.java$  coincide (up to an obvious syntactic modification of the signature), if all attributes of  $C$  are also attributes of  $C.java$ , and if for each (explicit or implicit) role name at a navigable association end  $C.java$  contains a corresponding reference attribute with the same name. (Note that standard types may be slightly renamed according to the Java syntax and that role names with multiplicity greater than one map to reference attributes of some container type.) Concerning visibility the correspondences are as follows [6]: Component public operations correspond to public methods (of public classes) in Java, component private operations to Java default visible methods, and class private operations to private Java methods; private attributes in UML correspond to private attributes in Java.

A realization dependency holds between a UML design

component  $M$  and a Java package  $P$ , see Fig. 5(b), if the following conditions are satisfied: Let  $Inv_M$  denote the component invariant of the design component  $M$  and let  $Inv_C$  denote the class invariant of class  $C$  in  $M$ . Let  $INV_C$  denote the inherited class invariant of class  $C$ , i.e., the conjunction of the class invariant of  $C$  and the class invariants of the super-classes of  $C$ :

$$INV_C = \bigwedge_{D \geq C} Inv_D$$

1. For all classes  $C$  in  $M$  there is exactly one class  $C.java$  in  $P$  such that  $C$  and  $C.java$  are related by a trace dependency.
2. Let  $op$  be an operation declared in the design class  $C$  with constraint

```
context C::op(x1 : D1, ..., xn : Dn)
pre: PreC::op
post: PostC::op
```

- (a) If  $op$  is class private then its corresponding private method  $op$  in  $C.java$  satisfies the pre-/post-condition (but no invariants):

```
{PreC::op}
C::op(x1 : D1, ..., xn : Dn)
{PostC::op}
```

- (b) If  $op$  is component private then the corresponding default visible method  $op$  in  $C.java$  preserves the inherited class invariant  $INV_C$  and satisfies the pre-/post-condition:

```
{PreC::op and INVC}
C::op(x1 : D1, ..., xn : Dn)
{PostC::op and INVC}
```

Moreover, any call of  $op$  for an object of a class  $D$  that is a sub-class of  $C$ , has to preserve the inherited invariant of  $D$  and satisfy the inherited pre-/post-condition for  $op$ :

```
{PreC::op and INVD}
D::op(x1 : D1, ..., xn : Dn)
{PostC::op and INVD}
```

- (c) If  $op$  is component public then the corresponding public method  $op$  in  $C.java$  fulfills the requirements for component private operations and additionally preserves the component invariant  $Inv_M$ :

```
{PreC::op and INVC and InvM}
C::op(x1 : D1, ..., xn : Dn)
{InvM}
```

3. Let  $C(x_1, \dots, x_n)$  be a constructor of the design class  $C$  with constraint

```
context  $C :: C(x_1 : D_1, \dots, x_n : D_n)$ 
  pre:  $Pre_C$ 
  post:  $Post_C$ 
```

- (a) If the constructor  $C$  is class private or component private then the corresponding private or default visible constructor  $C$  in  $C.java$  establishes the inherited class invariant  $INV_C$  and satisfies the pre-/post-condition:

```
{ $Pre_C$ }
   $x = \text{new } C(x_1 : D_1, \dots, x_n : D_n)$ 
  { $Post_C[x/\text{self}]$  and  $INV_C[x/\text{self}]$ }
```

- (b) If the constructor  $C$  is component public then the corresponding public constructor  $C$  in  $C.java$  fulfills the requirements for component private constructors and additionally preserves the component invariant  $Inv_M$ :

```
{ $Pre_C$  and  $Inv_M$ }
   $x = \text{new } C(x_1 : D_1, \dots, x_n : D_n)$ 
  { $Inv_M$ }
```

We say that a Java package is a correct realization of a UML design component  $M$  if a realization dependency holds between  $M$  and  $P$ . The requirements for a correct realization follow the conditions for class correctness in [9] or [8] which here are extended to take into account visibilities and component invariants. Note that, by requirement (2b) Liskov's substitution principle is satisfied with respect to pre-/post-conditions of inherited operations.

For the seminar system introduced in Sect. 2, we can establish a realization dependency between the UML design component in Fig. 4 and the Java implementation in Sect. 2 changing the public visibilities of `addSession` and `rmSession` into default visibilities. The proof obligations for setup originating from requirement (2b) are:

```
{start.before(end)}
  Session::setup()
{start = 9:00am and end = 11:00am and
  start.before(end)}
{start.before(end) and price >= 10$}
  PublicSession::setup()
{start = 9:00am and end = 11:00am and
  start.before(end) and price >= 10$}
{start.before(end) and price >= 10$}
  PublicSession::setup()
{price = 10$ and
```

```
start.before(end) and price >= 10$}
```

Obviously, the last two requirements can be easily combined into a single requirement.

Similarly, the proof obligations for `changeCourse` originating from requirement (2b) and (2c) are:

```
{instructor.qualifiedFor->includes(c) and
  start.before(end)}
  Session::changeCourse(c: Course)
{course = c and start.before(end)}
{instructor.qualifiedFor->includes(c) and
  start.before(end) and price >= 10$}
  PublicSession::changeCourse(c: Course)
{course = c and
  start.before(end) and price >= 10$}
{instructor.qualifiedFor->includes(c) and
  start.before(end) and  $Inv_{Seminar}$ }
  Session::changeCourse(c: Course)
  { $Inv_{Seminar}$ }
```

## 5 Properties of Correct Component Realizations

In a correct Java realization of a UML design component, method and constructor calls cannot destroy the class invariants of alien objects, i.e. of any object different from the object the method is called upon or different from the newly created object; additionally, all objects created during the execution of an operation or a constructor satisfy their class invariants.

**Lemma 1.** *Consider a correct realization of a design model. Assume that for any terminating method call  $o.op(d_1, \dots, d_n)$  and any terminating constructor call  $o = \text{new } C(d'_1, \dots, d'_m)$  the pre-condition of any method called during the evaluation of  $o.op(d_1, \dots, d_n)$  and  $o = \text{new } C(d'_1, \dots, d'_m)$  is satisfied. Then for any classes  $C$  and  $D$  of the implementation model, object  $o$  of class  $C$ , method  $op$  of  $C$ , and object  $o' \neq o$  of class  $D$  existing in the state after executing  $o.op(d_1, \dots, d_n)$ :*

```
{ $Inv_D[o'/\text{self}]$  and
   $D.allInstances() \rightarrow \text{includes}(o')$ }
   $o.op(d_1, \dots, d_n)$ 
  { $Inv_D[o'/\text{self}]$ }
  (1)
```

```
{not  $D.allInstances() \rightarrow \text{includes}(o')$ }
   $o.op(d_1, \dots, d_n)$ 
  { $o'.oclIsNew()$  and  $Inv_D[o'/\text{self}]$ }
  (2)
```

where  $d_1, \dots, d_n$  are some objects. Moreover, for any classes  $C$  and  $D$  of the implementation model, object  $o$  of class  $C$ , method  $op$  of  $C$ , and object  $o' \neq o$  of class  $D$  existing in the state after executing  $o = \text{new } C(d'_1, \dots, d'_m)$ :

$$\begin{aligned} & \{Inv_D[o'/\text{self}] \text{ and} \\ & D.\text{allInstances}() \rightarrow \text{includes}(o')\} \\ & o = \text{new } C(d'_1, \dots, d'_m) \\ & \{Inv_D[o'/\text{self}]\} \end{aligned} \quad (3)$$

$$\begin{aligned} & \{\text{not } D.\text{allInstances}() \rightarrow \text{includes}(o')\} \\ & o = \text{new } C(d'_1, \dots, d'_m) \\ & \{o'.\text{oclIsNew}() \text{ and } Inv_D[o'/\text{self}]\} \end{aligned} \quad (4)$$

where  $d'_1, \dots, d'_m$  are some objects.

*Proof sketch.* The claims are proved simultaneously by induction on the depth of the execution tree of method calls and object creations.

Each method call and each object creation is associated with an execution trace consisting of attribute assignments  $o.f = v$ , method calls  $o.op(v_1, \dots, v_l)$ , and object creations  $o = \text{new } C(v_1, \dots, v_l)$ .

We define the degree of a method call, an object creation, and an attribute assignment as follows:  $\text{deg}(o.f = v) = 0$ ,  $\text{deg}(o.op(d_1, \dots, d_n)) = 1 + \max_{1 \leq i \leq n} \{\text{deg}(s_i)\}$  where  $s_1 \dots s_k$  is the execution trace of  $o.op(d_1, \dots, d_n)$ , and  $\text{deg}(o = \text{new } C(v_1, \dots, v_l)) = 1 + \max_{1 \leq i \leq l} \{\text{deg}(s_i)\}$  where  $s_1 \dots s_k$  is the execution trace of  $o = \text{new } C(v_1, \dots, v_l)$ .

Let the assumptions of the claim hold and let  $o'$  be as in the claim. For any method call and any object creation, we proof the claim by induction on the degree of the method call or object creation and by sub-induction on the length of the execution trace.

*Case 1.* Let  $s_1 \dots s_k$  be the execution trace of a method call  $o.op(d_1, \dots, d_n)$ . Let the pre-condition  $Inv_D[o'/\text{self}]$  and  $D.\text{allInstances}() \rightarrow \text{includes}(o')$  hold before execution of  $s_1 \dots s_k$ .

*Case 1.1.* Let  $\text{deg}(o.op(d_1, \dots, d_n)) = 1$ . If  $k = 0$  then  $Inv_D[o'/\text{self}]$  holds trivially after  $s_1 \dots s_k$ . Let  $k > 0$  and let  $Inv_D[o'/\text{self}]$  hold before the execution of  $s_k$ . Then  $s_k$  has necessarily the form  $o.f = v$ , since all attributes have to be private and attributes are only called on this. Hence,  $Inv_D[o'/\text{self}]$  holds after the execution of  $s_k$  since  $o' \neq o$  and all class invariants have to be local, i.e., they must not employ navigation beyond the scope of a single object.

*Case 1.2.* Let  $\text{deg}(o.op(d_1, \dots, d_n)) = m + 1$ . If  $k = 0$  then  $Inv_D[o'/\text{self}]$  holds trivially after  $s_1 \dots s_k$ . Let  $k > 0$  and let  $Inv_D[o'/\text{self}]$  hold before the execution of  $s_k$ . Then there are three cases:

*Case 1.2.1.* If  $s_k = o.f = v$ , then the same argument as in case 1.1 applies.

*Case 1.2.2.* Let  $s_k = o'.op'(v_1, \dots, v_l)$ . If  $o'' = o'$  then  $op'$  cannot be private by our restriction on the programming

style of Java classes. Hence, by proof obligations (2b–c) and the pre-condition assumption,  $Inv_D[o'/\text{self}]$  holds after execution of  $s_k$ . If  $o'' \neq o'$  then  $Inv_D[o'/\text{self}]$  holds after execution of  $s_k$  by the main induction hypothesis, since  $\text{deg}(s_k) \leq m$ .

*Case 1.2.3.* Let  $s_k = o'' = \text{new } C'(v_1, \dots, v_l)$ . Then  $o'' \neq o'$ . Thus,  $Inv_D[o'/\text{self}]$  holds after execution of  $s_k$  by the main induction hypothesis, since, again,  $\text{deg}(s_k) \leq m$ .

An analogous argument holds for all remaining cases.  $\square$

From this observation, it follows that in a correct Java realization of a UML design model, any non-class private operation preserves all class invariants and, moreover, any component public operation preserves all invariants including the component invariant. In order to prove these properties let  $INV_M$  denote the environment invariant of design component  $M$ , i.e., the conjunction of all class invariants of the classes in  $M$  for all instances:

$$INV_M = \bigwedge_C C.\text{allInstances}() \rightarrow \text{forall}(x \mid Inv_C[x/\text{self}])$$

**Theorem 2.** Consider a correct realization of a design model  $M$ . Assume that for any terminating method call  $o.op(d_1, \dots, d_n)$  and any terminating constructor call  $o = \text{new } C(d'_1, \dots, d'_m)$  the pre-condition of any method called during the evaluation of  $o.op(d_1, \dots, d_n)$  and  $o = \text{new } C(d'_1, \dots, d'_m)$  is satisfied. Then for any class  $C$  of the implementation model and any non-class private method  $op$  of  $C$

$$\begin{aligned} & \{Pre_{C::op} \text{ and } INV_M\} \\ & C :: op(x_1 : D_1, \dots, x_n : D_n) \\ & \{INV_M\} \end{aligned}$$

holds; moreover, for any constructor  $C(x_1, \dots, x_n)$  of  $C$

$$\begin{aligned} & \{Pre_{C::C} \text{ and } INV_M\} \\ & x = \text{new } C(x_1 : D_1, \dots, x_n : D_n) \\ & \{INV_M\} \end{aligned}$$

holds.

*Proof.* Let  $o.op(d_1, \dots, d_n)$  be a terminating call of a non-class private method  $op$  of class  $C$  of the implementation model. Let  $Inv_D[o'/\text{self}]$  be any class invariant of  $INV_M$  where  $o'$  is an object existing in the state after executing  $o.op(d_1, \dots, d_n)$ . If  $o' = o$  and thus  $D \leq C$ , then  $Inv_D[o'/\text{self}]$  is ensured by the assumptions (2b) on preservation of invariants of non-class private methods in correct realizations of the design model. Otherwise, i.e. if  $o' \neq o$ , apply Lemma 1.

The claim on constructors is proved analogously.  $\square$

**Corollary 3.** With the assumption as in the theorem, any call of a component public operation or constructor preserves all

the invariants, i.e.

$$\{Pre_{C::op} \text{ and } INV_M \text{ and } Inv_M\}$$
$$C :: op(x_1 : D_1, \dots, x_n : D_n)$$
$$\{INV_M \text{ and } Inv_M\}$$
$$\{Pre_{C::C} \text{ and } INV_M \text{ and } Inv_M\}$$
$$x = \text{new } C(x_1 : D_1, \dots, x_n : D_n)$$
$$\{INV_M \text{ and } Inv_M\}$$

## 6 Conclusions

We have emphasized that pure class diagram-based object-oriented software development has some drawbacks related to invariants which can be overcome by using a component-based approach. Of course, the problems presented here are not the only problems in object-oriented and component-based software development. In particular, we have omitted a discussion of component hierarchies and associations between components, which may be treated by repeating the component invariant approach at all hierarchy levels. Inheritance between an element of one component and an element of another component seems to pose some more subtle problems. For instance, the interplay between inherited operations and component invariant preservation is not obvious. The issue of sharing between components [10] may have similar effects.

However, OCL has proven to be a valuable tool for analyzing well-known implementation schemata for associations between classes. In our opinion, OCL is well-suited as a constraint language for UML and presents a further positive step towards rigorous object-oriented software development.

## References

- [1] M. Bidoit, R. Hennicker, F. Tort, and M. Wirsing. Correct Realizations of Interface Constraints with OCL. In R. B. France and B. Rumpe, editors, *Proc. 2<sup>nd</sup> Int. Conf. UML*, volume 1723 of *Lect. Notes Comp. Sci.*, pages 399–415. Springer, Berlin, 1999.
- [2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Mass., &c., 1998.
- [3] M. Broy. Towards a Mathematical Concept of a Component and its Use. *Software — Concepts and Tools*, 18:137–148, 1997.
- [4] J. Cheesman and J. Daniels. *UML Components*. Addison Wesley, Boston, &c., 2000.
- [5] D. F. D’Souza and A. C. Wills. *Object, Components, Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, Mass., &c., 1998.
- [6] J. Gosling, B. Joy, G. Steele, and G. Brancha. *The Java Language Specification*. Addison-Wesley, Reading, Mass., &c., 2<sup>nd</sup> edition, 2000.
- [7] M. Hitz and G. Kappel. *UML@Work*. dpunkt.verlag, Heidelberg, 1999.
- [8] K. Huizing and R. Kuiper. Verification of Object-Oriented Programs Using Class Invariants. In T. Maibaum, editor, *Proc. Intl. Conf. Fundamental Approaches to Software Engineering 2000*, volume 1783 of *Lect. Notes Comp. Sci.*, Berlin, 2000. Springer, Berlin.
- [9] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, &c., 1988.
- [10] P. Müller and A. Poetzsch-Heffter. Modular Specification and Verification Techniques for Object-Oriented Software Components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, Cambridge, 2000.
- [11] Object Management Group. Unified Modeling Language Specification, Version 1.4. Draft, OMG, 2001. <http://cgi.omg.org/cgi-bin/doc?ad/01-02-14>.
- [12] B. Reus, M. Wirsing, and R. Hennicker. A Hoare Calculus for Verifying Java Realizations of OCL-Constrained Design Models. In H. Hußmann, editor, *Proc. 4<sup>th</sup> Int. Conf. Fundamental Approaches to Software Engineering*, volume 2029 of *Lect. Notes Comp. Sci.*, pages 300–317. Springer, Berlin, 2001.
- [13] C. Szyperski. *Component Software*. Addison-Wesley, Harlow, &c., 1998.
- [14] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, Reading, Mass., &c., 1999.