

# Specifying Component Invariants with OCL

Rolf Hennicker

Hubert Baumeister

Alexander Knapp

Martin Wirsing

Ludwig–Maximilians–Universität München

{hennicke, baumeist, 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 [12]) 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 inheritance and component hierarchy aspects.

In Sect. 2 we describe the problems with non-local class-based OCL invariants. In Sect. 3 we propose our component-based approach. 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 consider a simple seminar system taken almost literally from 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 date. There are two invariants: the simple invariant for the class `Session` requires that the start date is before the end date. The invariant for the class `Instructor` requires that an instructor should only

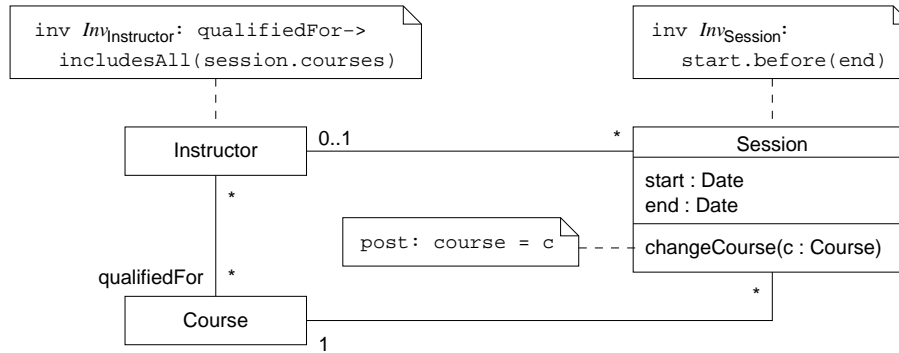


Figure 1: Annotated class diagram for the seminar example

teach sessions for courses he is qualified for. 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  $op(x_1 : D_1, \dots, x_n : D_n)$  of any class  $C$  of the diagram satisfies the class invariant  $Inv_C$  and the pre- and post-condition (cf. [10]), i.e., the Hoare formula

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

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

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

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

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

However, 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  $s.changeCourse(c)$  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 the following system state  $\sigma$  with two objects  $c_1, c_2$  of class `Course` and three objects  $s_1, s_2, s_3$  of class `Session` such that  $c_1$  is related with  $s_1$  and  $s_2$ , and  $c_2$  is related with  $s_3$ , see Fig. 2(a). Obviously, a Java call  $s_2.changeCourse(c_2)$  does not respect the invariants  $Inv'_{Course}[c_1/self]$  and  $Inv'_{Session}[s_2/self]$ , cf. Fig. 2(b). According to Hitz and Kappel [7, Sect. 6.2.1, p. 271–275], a correct Java implementation of `changeCourse` w.r.t. 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)
                course.rmSession(this);
            course = c;
        }
    }
}
```

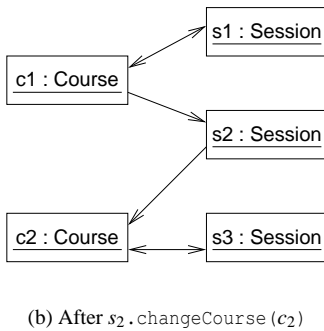
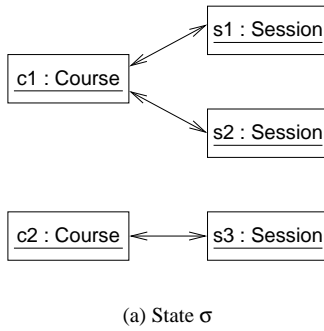


Figure 2: Sample states of the seminar system

```

        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.

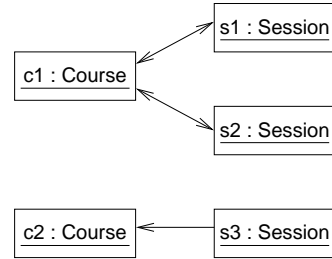


Figure 3: Sample state  $\sigma$  after  $c_2.rmSession(s_3)$

### 3 Component-Based Invariants

The problems described in the previous section can be solved by a component-oriented development methodology. Component-based approaches for software development have been advocated by many authors including [3, 11], 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.

For example, Catalysis components, the UML components of Cheesman and Daniels, or UML subsystems offer the required properties.

In our approach, 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` is a component invariant whereas the invariant  $Inv_{Session}$  of class `Session` is a class invariant. The invariants induced by bidirectional associations are component invariants.

Concerning the visibility of operations we distinguish between operations which are

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

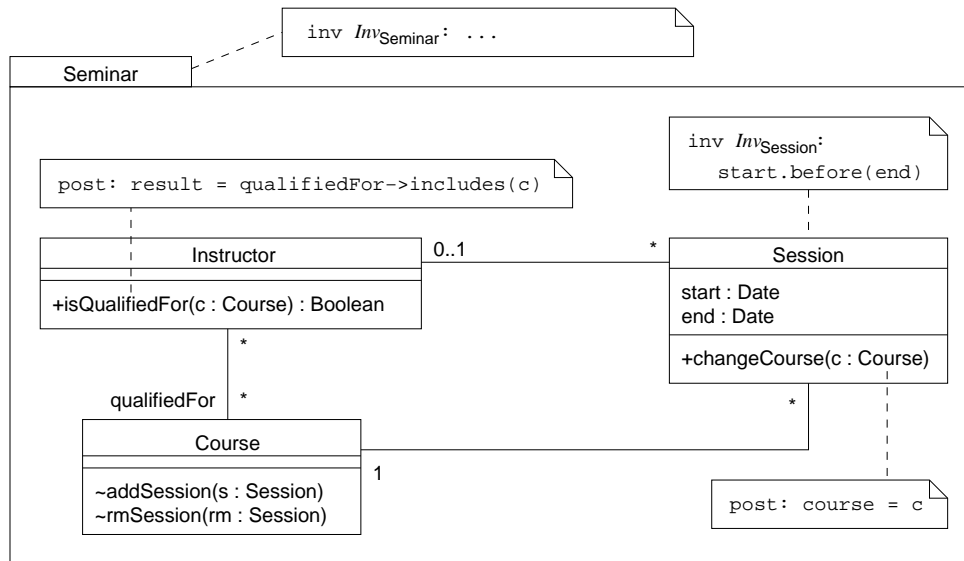


Figure 4: Components model of sample seminar system

class private —  
visible to a single class of the component

Components in the sense of Cheesman and Daniels or Catalysis can express these visibility requirements; in particular, due to the explicit notion of interface the visibility of component public operations can be modeled explicitly. A UML subsystem provides the following visibility correspondences for operations [9]: class private corresponds to private (–), component private to package (ˆ), and component public to public (+).

Fig. 4 shows the seminar system as a component where the component invariant  $Inv_{Seminar}$  reads:

```

context Seminar
  inv  $Inv_{Seminar}$ :
    Instructor.allInstances->forall(i |
      i.qualifiedFor->
        includesAll(i.session.course))
    and Session.allInstances->forall(s |
      s.course.session->includes(s))
    and Course.allInstances->forall(c |
      c.session->
        forall(s | s.course = c))
    ...

```

Based on our notion of component we can now define a realization relation which connects a given UML design component and its corresponding Java implementation model. As implementation model for components we use Java packages, classes in a design component are mapped to Java classes. This can be done by using trace dependencies as considered in [1] whereby in our case a

trace will always relate a class of the UML design component and a Java class as shown in Fig. 5(a).

We say that a trace dependency holds if the operations of the UML design class and the methods of the Java class coincide (up to an obvious syntactic modification of the signature), if all attributes of the design class are also attributes of the Java class and if for each role name at the end of a directed association the Java class 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.) These conditions guarantee that the OCL expressions used as constraints for the design model can be interpreted in the implementation model which is necessary to define the realization relation between a design model and an implementation model as shown in Fig. 5(b). Concerning visibility the correspondences are as follows [6]: Component public visible 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 with the following proviso: private Java attributes and methods are only called on *this*. The constraints on the Java implementation are represented by Hoare formulae and can be proven using the calculus presented in [10].

A realization relation between *DesignComponent* and *JavaPackage* expresses that the implementation model satisfies the requirements of the design model. This means that for each class *C* in the design model there is

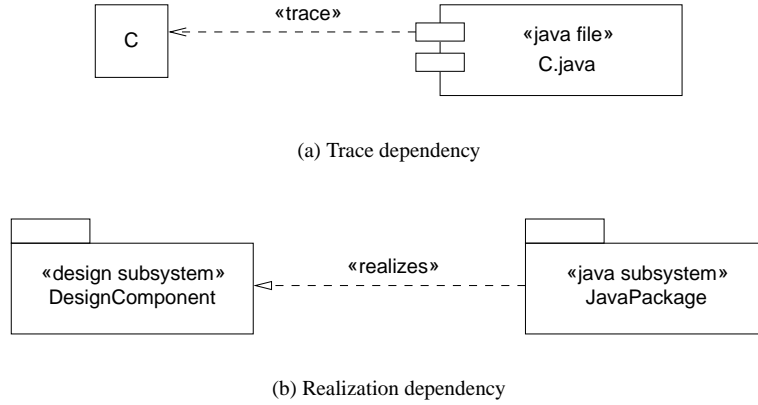


Figure 5: Component dependencies

exactly one class  $C.java$  in the implementation model that are related by a trace dependency. For each component public operation of  $C$  the corresponding method in  $C.java$  preserves the component invariant, preserves the class invariant of  $C$ , and satisfies its pre-/post-condition; similarly for component public constructors. For each component private operation of  $C$  the corresponding method in  $C.java$  preserves the class invariant of  $C$  and satisfies its pre-/post-condition; similarly for component private constructors. Finally, for each class private operation of  $C$  the corresponding method in  $C.java$  satisfies its pre-/post-condition; however, we require the constructors in  $C.java$  corresponding to class private constructors to establish the class invariant of  $C$  in addition to satisfying the pre-/post-condition.

More formally, let  $Inv_M$  denote the component invariant of  $DesignComponent$ , let  $Inv_C$  denote the class invariant of a class  $C$ , and let  $Inv_E$  be the conjunction of all class invariants, i.e.,

$$Inv_E = \bigwedge_T T.allInstances \rightarrow \text{forall}(x \mid Inv_T[x/self])$$

where  $Inv_T$  is the class invariant of  $T$  and where  $T$  ranges over all classes of the design model. Then  $JavaPackage$  is a correct realization of  $DesignComponent$  if for all classes  $C$  in the design model there is exactly one class  $C.java$  in the implementation model such that:

1.  $C$  and  $C.java$  are related by a trace dependency.
2. Let  $op$  be an operation of the design class  $C$  with constraint

```
context  $C::op(x_1 : D_1, \dots, x_n : D_n)$ 
pre:  $Pre_{op}$ 
post:  $Post_{op}$ 
```

- (a) If  $op$  is component public then the corresponding public method  $op$  in  $C.java$

- preserves the component invariant  $Inv_M$ :

$$\{Pre_{op} \text{ and } Inv_M \text{ and } Inv_E\}$$

$$op(x_1 : D_1, \dots, x_n : D_n)$$

$$\{Inv_M\}$$

- preserves the class invariant  $Inv_C$ :

$$\{Pre_{op} \text{ and } Inv_C\}$$

$$op(x_1 : D_1, \dots, x_n : D_n)$$

$$\{Inv_C\}$$

- satisfies the pre-/post-condition:

$$\{Pre_{op}\}$$

$$op(x_1 : D_1, \dots, x_n : D_n)$$

$$\{Post_{op}\}$$

- (b) If  $op$  is component private then the corresponding default visible method  $op$  in  $C.java$

- preserves the class invariant  $Inv_C$ :

$$\{Pre_{op} \text{ and } Inv_C\}$$

$$op(x_1 : D_1, \dots, x_n : D_n)$$

$$\{Inv_C\}$$

- satisfies the pre-/post-condition:

$$\{Pre_{op}\}$$

$$op(x_1 : D_1, \dots, x_n : D_n)$$

$$\{Post_{op}\}$$

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

$$\begin{array}{l} \{Pre_{op}\} \\ op(x_1 : D_1, \dots, x_n : D_n) \\ \{Post_{op}\} \end{array}$$

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

$$\begin{array}{l} \text{context } C::C(x_1 : D_1, \dots, x_n : D_n) \\ \text{pre: } Pre_C \\ \text{post: } Post_C \end{array}$$

- (a) If the constructor  $C$  is component public then the corresponding public constructor  $C$  in  $C.java$

- preserves the component invariant  $Inv_M$ :

$$\begin{array}{l} \{Pre_C \text{ and } Inv_M \text{ and } Inv_E\} \\ x = \text{new } C(x_1 : D_1, \dots, x_n : D_n) \\ \{Inv_M\} \end{array}$$

- establishes the class invariant  $Inv_C$ :

$$\begin{array}{l} \{Pre_C\} \\ x = \text{new } C(x_1 : D_1, \dots, x_n : D_n) \\ \{Inv_C[x/self]\} \end{array}$$

- satisfies the pre-/post-condition:

$$\begin{array}{l} \{Pre_C\} \\ x = \text{new } C(x_1 : D_1, \dots, x_n : D_n) \\ \{Post_C[x/self]\} \end{array}$$

- (b) If the constructor  $C$  is class private or component private then the corresponding private or default visible constructor  $C$  in  $C.java$

- establishes the class invariant  $Inv_C$ :

$$\begin{array}{l} \{Pre_C\} \\ x = \text{new } C(x_1 : D_1, \dots, x_n : D_n) \\ \{Inv_C[x/self]\} \end{array}$$

- satisfies the pre-/post-condition:

$$\begin{array}{l} \{Pre_C\} \\ x = \text{new } C(x_1 : D_1, \dots, x_n : D_n) \\ \{Post_C[x/self]\} \end{array}$$

Note that in general

$$\{ \} x = \text{new } C(x_1 : D_1, \dots, x_n : D_n) \{x.\text{oclIsNew}()\}$$

Given an operation call  $o.op(d_1, \dots, d_n)$  and a call to a constructor  $o = \text{new } C(d'_1, \dots, d'_m)$ , part (1) and (3) of the following lemma ensure that invariants of already existing objects other than  $o$  are preserved, part (2) and (4) guarantee that all objects created during the execution of the constructor or the operation  $op$  satisfy their 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  $T$  of the implementation model, object  $o$  of class  $C$ , method  $op$  of  $C$ , and object  $o' \neq o$  of class  $T$  in the state after executing  $o.op(d_1, \dots, d_n)$ :*

$$\begin{array}{l} \{Inv_T[o'/self]\} \text{ and} \\ T.\text{allInstances} \rightarrow \text{includes}(o') \} \\ o.op(d_1, \dots, d_n) \\ \{Inv_T[o'/self]\} \end{array} \quad (1)$$

$$\begin{array}{l} \{\text{not } T.\text{allInstances} \rightarrow \text{includes}(o')\} \\ o.op(d_1, \dots, d_n) \\ \{o'.\text{oclIsNew}() \text{ implies } Inv_T[o'/self]\} \end{array} \quad (2)$$

where  $d_1, \dots, d_n$  are some objects. Moreover, for any classes  $C$  and  $T$  of the implementation model, object  $o$  of class  $C$ , method  $op$  of  $C$ , and object  $o' \neq o$  of class  $T$  in the state after executing  $o = \text{new } C(d'_1, \dots, d'_m)$ :

$$\begin{array}{l} \{Inv_T[o'/self]\} \text{ and} \\ T.\text{allInstances} \rightarrow \text{includes}(o') \} \\ o = \text{new } C(d'_1, \dots, d'_m) \\ \{Inv_T[o'/self]\} \end{array} \quad (3)$$

$$\begin{array}{l} \{\text{not } T.\text{allInstances} \rightarrow \text{includes}(o')\} \\ o = \text{new } C(d'_1, \dots, d'_m) \\ \{o'.\text{oclIsNew}() \text{ implies } Inv_T[o'/self]\} \end{array} \quad (4)$$

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

*Proof sketch.* The claim is proved by induction on the execution trace of  $o.op(d_1, \dots, d_n)$  and  $o = \text{new } C(d'_1, \dots, d'_m)$  and crucially depends on our restrictions on class invariants, calls of private methods, and attributes: Class invariants have to be local, i.e., they must not employ navigation beyond the scope of a single object; private methods and attributes are only called on `this`; all attributes have to be private.  $\square$

For a correct realization of a design model, any non-class private operation preserves all class invariants and, as a consequence, any component public operation preserves all invariants:

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

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

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

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

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  $Inv_T[o'/self]$  be any class invariant of  $Inv_E$  where  $o'$  is an object of the state after executing  $o.op(d_1, \dots, d_n)$ . If  $o' = o$  and thus  $T = C$ , then  $Inv_T[o'/self]$  is ensured by the assumptions 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.*

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

Indeed, using this methodology, we obtain a correct realization of the seminar system by not declaring `rmSession` and `addSession` to be public but only to have default visibility, and changing the implementation of `changeCourse` in accordance with the component invariant  $Inv_{Seminar}$ :

```
public void changeCourse(Course c) {
    if (course != c) {
        if (instructor.isQualifiedFor(c) {
            if (course != null)
                course.rmSession(this);
            course = c;
            c.addSession(this);
        }
    }
}
```

## 4 Conclusions

We have shown that pure class diagram-based object-oriented software development has some drawbacks 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; e.g., sharing and behavioral subtyping or component hierarchies are other major issues [8]. However, OCL has proven to be a valuable tool for analyzing at least some deficiencies of 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, 2000.
- [5] D. F. D’Souza and A. C. Wills. *Object, Components, Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, Mass., 1998.
- [6] J. Gosling, B. Joy, G. Steele, and G. Brancha. *The Java Language Specification*. Addison–Wesley, Reading, Mass., 2<sup>nd</sup> edition, 2000.
- [7] M. Hitz and G. Kappel. *UML@Work*. dpunkt.verlag, Heidelberg, 1999.

- [8] P. Müller and A. Poetsch-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.
- [9] Object Management Group. Unified Modeling Language Specification, Version 1.4. Draft, OMG, 2001. <http://cgi.omg.org/cgi-bin/doc?ad/01-02-14>.
- [10] 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.
- [11] C. Szyperski. *Component Software*. Addison-Wesley, Harlow, 1998.
- [12] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, Reading, Mass., &c., 1999.