Cite as follows: Bertrand Meyer, *The Dependent Delegate Dilemma*, in *Engineering Theories of Software Intensive Systems*, eds. Manfred Broy, J Gruenbauer, David Harel, C.A.R. Hoare, NATO Science Series II: Mathematics, Physics and Chemistry, vol. 195, Springer-Verlag, June 2005, to appear.

The Dependent Delegate Dilemma

Bertrand Meyer, ETH Zurich

ABSTRACT

A criticism of the object-oriented style of programming is that the notion of class invariant seems to collapse in non-trivial client-supplier relationships: a supplier ("*Dependent Delegate*") called from within the execution of a routine, where the invariant is not required to hold, may call back into the originating object, which it then catches in an inconsistent state. This is one of the problems arising from the application of assertion-based semantics to a model of computation involving references and the resulting possibility of dynamic aliasing.

This note suggests handling such cases by applying the basic non-object-oriented Hoare rule, instead of the version involving the invariant. It does not consider inheritance and dynamic binding.

1 OVERVIEW

A key concept of object-oriented programming, essential for reasoning about classes and their instances, is the class invariant. A class invariant expresses a consistency property applicable to all instances of a class. For example a class *PERSON* with a query *spouse* returning a *PERSON* and a boolean query *is_married* may include the invariant clauses:

```
is_married = (spouse /= Void)
is_married implies (spouse.spouse = Current))
```

In words: a person is married if and only if "he" has a spouse, and in that case the spouse of that spouse is the person himself (the "*Current*" object as talked about in the class).

Despite its name, the class invariant is, for all but non-trivial examples, not always satisfied; it only has to hold when the object is officially accessible to clients. During the execution of a routine of the class, the invariant may be temporarily violated. This is already clear from our example: any routine that affects *spouse* or *is_married*, for example a procedure *marry* (*p*: *PERSON*) that sets the spouse of the current person to *p* and *is_married* to True, will temporarily, in-between those two setting operations, falsify the invariant. This is considered acceptable since in such an intermediate state the object is not directly usable by the rest of the world — it is busy executing a routine, *marry* —, so it doesn't matter that its state might be inconsistent. What counts is that the invariant will hold before and after the execution of calls such as *Alice*.*marry* (*Bob*), executed by clients of the class *PERSON*.

The Dilemma of interest for this note arises when such a client is also a supplier, direct or indirect. A typical scheme is for a routine *r* (which could be the *marry* of our example) to pass the current object to a supplier:

```
r is
    do
        … Instructions_1 …
        some_supplier.some_work (Current)
        … Instructions_2 …
    end
```

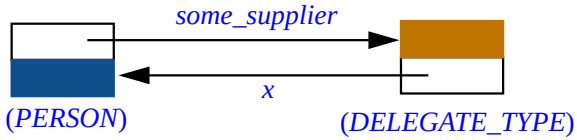
This tells another object *some_supplier* (the "*Dependent Delegate*") to do *some_work*, for which it may need to access the current object, passed to it as an argument. As a consequence, part of *some_work* could be a call (a "*Dependent Delegate Callback*") back into that same object:

```
some_work (x: PERSON) is
    do
        …
        x.some_operation
        …
    end
```

In this execution, *x* happens to be the former "current object" that is now waiting for the execution of *r* to terminate. But then the call to *some_operation*, back into that object, catches it unawares: there is no guarantee that the object will satisfy the invariant at that point, since the Instructions_1 might have invalidated that invariant, as they are entitled to do — without violating the correctness requirement of the original class — provided the Instructions_2 reëstablish it.

This is the Dependent Delegate Dilemma: you hope to delegate a certain task to a supplier, but discover that the supplier (the delegate) is dependent on you, soon coming back with requests for your own help. Since you didn't expect those requests — naïvely believing, like many a novice manager before you, that delegating a task means you can stop worrying about it and just wait for the delegate to come back with the work done — they may catch you in a state that doesn't satisfy the invariant (you may for example be dozing off between meetings with successive visitors).

The ultimate cause behind the Dilemma is the role of references in the object-oriented model of computation and the resulting possibility of dynamic aliasing. In our example the delegate object can, through *x*, keep a reference to the original *PERSON* object:



Such dynamic aliasing is part of the flexibility provided by the use of references, but complicates assertion-based reasoning about program behavior.

The next sections examine the Dilemma and suggest addressing it through proper application of Hoare-style specifications. It is important for this discussion to note the context in which the Dilemma may occur:

> The Dependent Delegate Dilemma arises when a supplier of a class is also — because it calls back one of its routines — a client of that class.

The case of a class that is both a client and supplier of another, introducing a cycle in the client class, is known to be delicate. For example:

- It prohibits a client relationship of the "expanded client" form where every instance of *A* contains a subobject of *B*, rather than the usual "reference client" form where each instance of *A* contains a (possibly void) reference to an object of type *B*. Eiffel's compile-time validity rules explicitly prohibit cycles in the expanded client relation **[5]**.
- Cyclic client relationships make invariants more difficult to enforce. Class *PERSON* may have a feature *residence*: *HOUSE* and the invariant clause *residence* /= *Void* **implies** *residence*.*resident* = *Current*, where *HOUSE* has *resident*: *PERSON*. Even if all the routines of class *PERSON* preserve that invariant, a routine of class *HOUSE* can violate it by assigning to *resident*. Looking at one of the classes alone will not reveal the error. This **Indelicate Delegate** problem is closely related to the Dependent Delegate Dilemma.

  This issue is discussed in **[6]** (11.14, "Class invariants and reference semantics") with the informal suggestion of adding a symmetric invariant: here, in class *HOUSE*, *resident* /= *Void* **implies** *resident*.*residence* = *Current*.

2 RULES FOR ROUTINE CALLS

A routine call stands for the execution of the corresponding routine body, with actual arguments if any substituted for the corresponding formals. This is captured by the traditional (non-O-O) Hoare rule for routines, which in a simplified form sufficient for this discussion we may write

$$\frac{\{P\}\ \ body\ \ \{Q\}}{\{P'\}\ \ call\ \ \{Q'\}}$$ **N_RULE**

where *P* and *Q* are assertions (precondition and postcondition), *body* is the body of a routine, *call* is a call to that routine, and priming (in *P'* and *Q'*) stands for substitution of actual for formal arguments. The rule states that, after such substitution, we may infer a property of any call to a routine from the corresponding property of the routine's body.

We call this rule **N_RULE** (**N** for non-object-oriented).

N_RULE applies to calls of the form

some_routine (*some_arguments*)	**[UNQUAL]**

This doesn't just includes routine calls in a non-O-O language, but also, in an O-O language, calls of the **UNQUAL** form executed by another routine in the same class as *some_routine*, which simply calls *some_routine* on the same object on which it is currently executing. The correctness of such calls, said to be **unqualified**, falls under **N_RULE**.

In an object-oriented language, we also have **qualified** calls of the form

some_object.*some_routine* (*some_arguments*)	**[QUAL]**

where *some_routine* must be exported to the appropriate class (a *client*) executing the qualified call. It is for such qualified calls that the class invariant intervenes, in the form of the modified rule

$$\frac{\{P \ \wedge INV\} \quad body \quad \{Q \wedge INV\}}{\{P'\} \quad call \quad \{Q'\}} \qquad \textbf{I\_RULE}$$

called "**I_RULE**" because it involves the invariant *INV* of the class. The invariant helps us reason about the class:

- Being added to the precondition, it facilitates writing the routine by allowing us to assume that it always finds the object in a consistent state.
- Being added to the postcondition, it imposes on the routine the extra requirement of ensuring the postcondition on exit.

For example, a class describing bank accounts may have an invariant clause stating *balance = deposits*.*total* – *withdrawals*.*total*: the current balance is consistent with the history of deposits and withdrawals. An exported routine that manipulates the account may assume this: it doesn't have to worry about finding an inconsistent object. It must, however, worry about avoiding that same inconsistency on return. So if for example it modifies the list of *deposits*, it must update the *balance* accordingly.

To complement **I_RULE** there's also a rule ensuring that, on creation, every object satisfies the invariant of its generating class. It reads

$$\frac{\{\ P \quad \wedge \mathit{Defaults}\ \}\quad \mathit{c\_body}\quad \{Q \wedge \mathit{INV}\}}{\{P'\}\quad \mathit{c\_call}\quad \{Q'\}}$$

C_RULE

and applies to a creation procedure ("constructor" in C++); *Defaults* denotes the result of default initializations. **C_RULE** is to **I_RULE** what the base step of an induction rule is to the induction step. It is not, however, essential to the present discussion.

For both **N_RULE** and **I_RULE** the inferred property of routine calls — the consequent of the rule — is the same: $\{P'\}\ \mathit{call}\ \{Q'\}$. The invariant figures only, in **I_RULE**, in the property of the routine body — the hypothesis that we must prove to allow the inference. This presents the invariant as an internal property of the class rather than one directly relevant to clients. Indeed, an invariant typically includes, along with official properties, corresponding to axioms of the corresponding abstract data type, a part known as the **representation invariant** **[3]** which involves secret features of the class and hence should not be visible to clients.

This discussion suggests a first definition of the correctness of a class:

Definition: Class Correctness (basic)

A class is correct if every routine *r* of the class satisfies the following properties:

1 • **N_RULE** if a routine of the class calls *r* unqualified.

2 • **I_RULE** if *r* is exported to at least one client.

3 • **C_RULE** if *r* is a creation procedure.

The three cases are not exclusive; a routine *r* that falls into more than one case must satisfy the associated clauses. In particular:

- In Eiffel, a procedure of the class may be available for normal calls $x.p\ (a)$, where clause 2 applies, as well as for creation calls **create** $x.p\ (a)$ which subject it to clause 3. (This is not the case in languages such as C++, Java and C# where constructors are special mechanisms distinct from the features of the class.)
- More directly relevant to this discussion, *r* may be both called by other routines of the class in unqualified form and available for qualified calls by clients, subjecting it to clause 1 as well as 2.

Clause 2 is stronger than needed since it would suffice to require that *r* satisfy **I_RULE** if some client actually calls it qualified, as in *x*.*p* (*a*). But then we couldn't check class correctness without knowing all the clients of a class; this would mean that the check is *global*, applying to an entire program ("system" in Eiffel terminology). As given, the rule is *modular*: enforceable at the level of individual classes.

Clause 2 makes **I_RULE** applicable to routines exported to "at least one client". In some object-oriented languages a feature is either secret or public; then the rule will apply only to exported features. In others, the policy is more fine-grained. For example C++ has a notion of "friend" classes and C# allows export to the "family" of a class or to its assembly. In Eiffel, it is possible to export a feature to specific classes, as in

```
class
    C
feature
    "Declaration of features f1, …"
feature {A, B}
    "Declaration of features g1, …"
feature {A, B, C}
    "Declaration of features h1, …"
feature {NONE}
    "Declaration of features i1, …"
end
```

The specifications determine whether a call of the form *x*.*f* (…), where *x* is declared of type *C* and *f* is one of *f1*, *g1* etc., is valid in a class *CLIENT*:

- For *f1*, *CLIENT* can be any class (the feature is public).
- For *i1* the call is never valid (*NONE* is, by convention, the bottom of the inheritance graph).
- For *g1* the call is valid only in a class *CLIENT* that is *A*, *B* or one of its descendants. (If we export a feature to a class we should also export it to its descendants.)

Because the rule applies to all qualified calls, *x*.*g1* (…) is not valid in class *C* itself, because the call uses the class as its own client. For the call to be valid, we must export the feature to the class itself, as with *h1* and of course *f1*. This policy distinguishes Eiffel from languages such as Java and C#, where a class may always use its own features. It follows from the principle that:

- A feature is always usable, within its own class, in unqualified calls.
- A qualified call, however, is only valid if it appears in a client to which the feature is exported. This means that if the client is the same as the supplier, it must export the feature to itself.

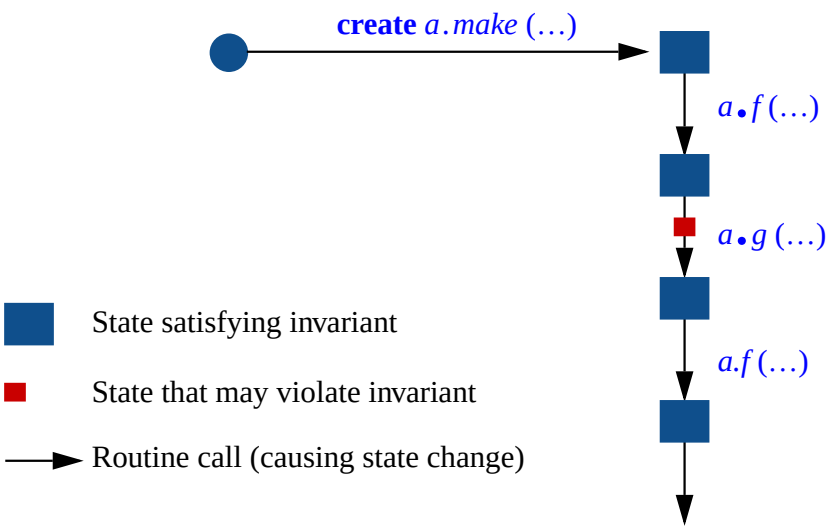
The rule that invariants (clause 2 of the rule) only matter for qualified calls also affects *run-time assertion monitoring*. Current Eiffel implementations do not yet support full proofs of correctness but offer optional run-time contract monitoring. With invariant monitoring turned on, invariant checks, on routine entry and exit, only take place for qualified calls. This means in particular (for example under Eiffel Software's EiffelStudio compiler) that the calls

```
f (…)
Current.f (…)
```

although equivalent for a correct program, differ in the presence of invariant monitoring: the second will cause the invariant to be checked, the first won't.

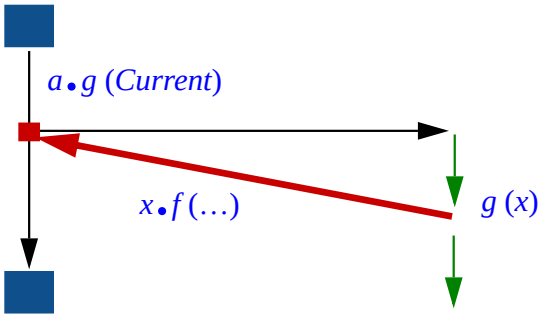
3 THE DEPENDENT DELEGATE RULE

The purpose of the Class Correctness rule is to ensure that clients get the promise of the class routines' contracts, based on the assumption that whenever an instance of the class is observable from the outside it will satisfy its invariant. We may informally picture the life of an object as follows **[6]**:



During the execution of qualified calls (as in the mark appearing in the execution of *g*) the invariant may temporarily be violated; but it will hold before and after the execution of these calls.

With the possibility of calls to "dependent delegates" the basic Class Correctness rule is no longer sufficient to ensure this property:



As illustrated, *g* may call back into the original object which it finds in a state violating the invariant.

If such a callback from a dependent delegate (a supplier that is also a client) occurs, it no longer suffices that the routine being called back, *f* in the figure, satisfy **I_RULE**, since it is called outside of invariant-satisfying states. The callback is similar to an unqualified call as may be executed from within the class, which the basic Class Correctness rule addresses through **N_RULE** (clause 1). It seems appropriate to address this case through the same formal device, yielding a new clause:

Definition: Class Correctness (extended)

In addition to the preceding clauses, *r* must satisfy:

4 • **N_RULE** if a supplier of the class calls *r* qualified.

This extra requirement appears to take care of the Dependent Delegate Dilemma in the absence of dynamic binding and is the contribution of the present note.

The added condition can be checked locally for each class (in other words, it is "*modular*"): while knowing the *clients* of a class requires access to the entire system (program), analyzing any class requires having access to its *suppliers* (the classes used as type for *x* in any calls of the form *x* • *f* (…) appearing in the class). Clause 4 arises when such a supplier (direct or indirect) is also a client, calling back into the object; checking it requires no more information than is needed anyway to analyze the class, independently of any specific system to which the class may belong.

4 AN EXAMPLE

To see how the Dependent Delegate Dilemma and the solution presented work out in practice, let us develop the “marriage” example sketched earlier:

```
class PERSON feature
        spouse: PERSON
                -- Spouse, if any
        is_married: BOOLEAN is
                -- Is this person married?
            do
                Result := (Spouse /= Void)
            end
        … Procedures such as marry (see below) …
invariant
        is_married = (spouse /= Void)
        is_married implies (spouse.spouse = Current))
end
```

Since there is no explicit creation procedure, instances of this class will be created through the default creation mechanism **create** *p* (for *p*: *PERSON*) which initializes all booleans such as *is_married* to False and all references such as *spouse* to void, ensuring that the instance satisfies the invariant, as per **C_RULE**.

Here is a first attempt at a procedure to marry the current *PERSON* object to another (omitting preconditions not relevant to the discussion, such as *p* not being *Current*):

```
        marry1 (p: PERSON) is
                -- Get married to p.                    -- [INCORRECT VERSION]
            require
                p /= Void
                not is_married
                not p.is_married
            do
                spouse := p
                is_married := True
                p.marry1 (Current)
            ensure
                is_married
                spouse = p
            end
```

This will not work since the call to *marry1* violates the second clause of the precondition: for this call *p* represents the current *PERSON* object, whose *is_married* attribute has just been set to True.

Note that in this example the Dependent Delegate of class *PERSON* is *PERSON* itself. The discussion can be directly transposed to the example of people's residence and houses' residents, which involves two distinct classes.

We might try reversing the order of instructions:

```
marry2 (p: PERSON) is
        -- Get married to p.                    -- [INCORRECT VERSION]
    require
        … As for marry1 …
    do
        p.marry2 (Current)
        spouse := p
        is_married := True
    ensure
        … As for marry1 …
    end
```

but the call causes infinite recursion.

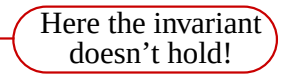
It seems inevitable to introduce a routine with fewer restrictions than *marry*, a plain "setter" procedure, which we call *get_engaged*:

```
feature {PERSON} -- Implementation
    get_engaged (p: PERSON) is
            -- Set spouse to p and is_married to True.
        -- No precondition!
        do
            spouse := p
            is_married := True
        ensure
            spouse = p
            is_married
        end
```

Procedure *get_engaged* is useful only for *PERSON*'s internal purposes; as a consequence it appears in a feature clause labeled **feature** {*PERSON*}, meaning it's exported only to *PERSON* itself, allowing routines of the class to use calls such as *p1*.*get_engaged* (*p2*) for *p1* and *p2* of type *PERSON*.

We may now write a correct version of *marry*:

```
marry3 (p: PERSON) is
        -- Get married to p.
    require
        … As for marry1 …
    do
        get_engaged (p)
        p.get_engaged (Current)
    ensure
        … As for marry1 …
    end
```



The call *p.get_engaged (Current)* is executed in a state that doesn't satisfy the invariant since *is_married* is now True but *spouse.spouse* is not *Current* (it's indeed the purpose of that call to set it to *Current*).

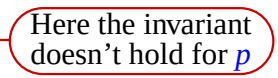
This call, *p.get_engaged (Current)*, is a dependent delegate callback on *p*. It doesn't actually cause a problem with the original Class Correctness rule since that at stage *p* satisfies the invariant (its *spouse* is void and its *is_married* is false). To illustrate a potentially damaging callback we replace *get_engaged* by two separate setter procedures:

```
feature {PERSON} -- Implementation
    set_spouse (p: PERSON) is
            -- Set spouse to p.
        -- No precondition!
        do
            spouse := p
        ensure
            spouse = p
        end
    set_married is
            -- Set is_married to True.
        -- No precondition!
        do
            is_married := True
        ensure
            is_married
        end
```

Then we may write the marrying procedure as

```
marry4 (p: PERSON) is
        -- Get married to p.
    require
        … As for marry1 …
    do
        set_married
        p.set_married

        set_spouse (p)
        p.set_spouse (Current)
    ensure
        … As for marry1 …
    end
```



where the last calls catches the object associated with *p* in a state that doesn't satisfy the invariant, since its *is_married* is true but (assuming *p* had just been created and initialized to the default) its *spouse* is still void. With only the basic Class Correctness rule this would make *marry4* incorrect; the extended rule, however, only requires *set_spouse* and *set_married* to satisfy **N_RULE**, which they do since they trivially ensure their postconditions.

5 IMPROVING RUN-TIME INVARIANT MONITORING

The extended Class Correctness rule appears to provide a basis for addressing the Dependent Delegate Dilemma, although this note does not address inheritance and dynamic binding, and does not provide a formal proof, which requires a mathematical model of O-O computation.

A practical consequence for today's Design by Contract support systems, enforcing run-time contract monitoring rather than proofs, is that **invariant monitoring should not apply to Dependent Delegate Callbacks**. As noted, invariant monitoring applies only to qualified calls; a Dependent Delegate Callback is qualified (*x.f*) but, like an unqualified call *f*, it may catch the object in a state that doesn't satisfy the invariant, without signaling any actual mistake in the system. Eiffel Software's EiffelStudio implementation **[1]** checks the invariant in this case; so do (as far as I know) other Eiffel compilers.

This policy should be corrected as it may lead to false alarms. Such situations, although very rare, do occasionally occur in practice; programmers address them through calls to library routines that disable invariant monitoring before the offending callback and reënable it after. Instead of forcing such ad hoc solutions on the programmer, compilers should take care of the problem by skipping the invariant check for dependent delegate callbacks.

Detecting that a qualified call is in fact a dependent delegate callback shouldn't be hard for compilers; this is, as noted, a local check, not requiring any more information than already needed to analyze and compile a class.

6 ABOUT THE INDELICATE DELEGATE PROBLEM

An earlier part of the discussion mentioned the Indelicate Delegate problem, causing a class invariant to be invalidated, through reference reassignment, beyond the control of the class itself. Although this note concentrates on the Dependent Delegate Dilemma, we may take a look at the relationship between the two issues.

The Indelicate Delegate problem is indeed lurking in our marriage example. All the *marry* procedures so far assumed the precondition clauses **not** *is_married* and **not** *p*.*is_married*. Assume we remove these clauses, to loosen the requirements by allowing remarriage. Then for non-void *p*, *q* and *r* a client may execute the successive calls

```
p.marry (q)
q.marry (r)
```

The second call remarries *q* to *r*. If *marry* is written — for example as *marry3* or *marry4* — to preserve the invariant in accordance with **I_RULE**, it will ensure *spouse*.*spouse* = *Current* for both *q* and *r*. But neither implementation updates any property of *p*; indeed, *spouse*.*spouse* will, for *p*, end up with value *r*! Such bigamous behavior on the part of *q* leads to moral outrage, the full punishment of the law and (the real scandal for this discussion) breach of software correctness.

[6], as already noted, suggested providing a symmetric invariant. This is clearly not the right approach here, since only one class is involved: the symmetric invariant is the same as the original, *spouse*.*spouse* = *Current* (under *is_married*).

What we seem actually to need here is *spouse*.*spouse*.*spouse* = *spouse*. Future work will address the issue in a more general setting.

7 SUMMARY AND CONCLUSION

This note has proposed a simple solution to the Dependent Delegate Dilemma, based on a simple correctness rule: requiring that any routine used in a dependent delegate callback satisfy, in addition to the object-oriented correctness property, the traditional routine rule. If this solution is right, it should be applied right away by run-time contract-monitoring options of current compilers.

8 ACKNOWLEDGMENTS

This note derives from an email message to Peter Müller and Rustan Leino (whose comments and criticism I gratefully acknowledge) during a discussion in November 2003, originally triggered by comments by Tony Hoare at the WG 2.3 meeting in Monterey in January 2002 (where he appears to have suggested the solution described here, although I did not realize it then). It was further fueled by remarks from Manfred Broy in Marktoberdorf in August 2004. Peter Müller provided further corrections.

REFERENCES

[1] Eiffel Software: EiffelStudio documentation, online at eiffel.com.

[2] C.A.R. Hoare: *Procedures and Parameters: An Axiomatic Approach*, in Symposium on the Semantics of Programming Languages, ed. Erwin Engeler, Lecture Notes in Mathematics 188, Springer-Verlag 1971, pages 103-116; reprinted in C.A.R. Hoare and C. B. Jones (eds.), *Essays in Computing Science*, Prentice Hall International 1989, pages [COMPLETE].

[3] C.A.R. Hoare: *Proofs of Correctness of Data Representations*, in *Acta Informatica*, vol. 1, 1972, pages 271-281; reprinted in C.A.R. Hoare and C. B. Jones (eds.), *Essays in Computing Science*, Prentice Hall International 1989, pages 103-115.

[4] K. Rustan Leino and Peter Müller: *Object Invariants in Dynamic Contexts*, in *European Conference on Object-Oriented Programming*, LNCS 3086, Springer-Verlag, June 2004, pages 491-516.

[5] Bertrand Meyer: *Eiffel: The Language*, 2nd printing, Prentice Hall, 1992.

[6] Bertrand Meyer, *Object-Oriented Software Construction, 2nd edition*, Prentice Hall, 1997.

[7] Peter Müller: *Modular Specification and Verification of Object-Oriented Programs*, LNCS 2262, Springer-Verlag, 2002.