JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 2, March-April 2005

On theory and practice of Assertion Based Software Development

Herbert Toth, SIEMENS AG Austria, PSE KB

Abstract

It is common agreement that software engineering can meet its challenges only if disciplined reuse and composition mechanisms can be established in both theory and practice. In this paper we provide a thorough analysis of the percolation pattern and three alternatives to it. As result of this analysis we get that each of these alternative checking strategies ensures behavioral subtyping and therefore good reuse properties. However, none of them allows for modular reasoning due to missing success or failure conformance over class hierarchies.

1 INTRODUCTION

One of the real challenges for today's software engineers is that they usually are requested by their customers to produce what I would like to call "frency" software. (No, you are completely wrong: similarities in pronounciation are purely accidental!) Customers want programs to be flexible, robust, efficient, non-expensive, correct, and moreover to be ready "yesterday" – and all this regardless of all the mostly negative impacts of various other project relevant circumstances. As in other engineering disciplines, reuse of existing components with well defined interfaces is regarded to be the only realistic approach to meet the needs of software industry. No surprise that a considerable number of programming languages and development methods have been proposed during the last three decades to help software engineers create such reusable abstractions.

The concept of inheritance is one of the key features for the success of object-oriented progamming languages and design methods. Its main promise is to offer increased reusability and extendibility of software, and also supports the design of well-structured systems. However, these benefits do not come for free, and this is the reason why the following vision from the late sixties still has not become reality to a large extent:

> *"Software components (routines), to be widely applicable to different machines and users, should be available in families arranged according to precision, robustness, generality and time-space performance. Existing sources of components - manufacturers, software houses, users' groups and algorithm collections - lack the*

breadth of interest or coherence of purpose to assemble more than one or two members of such families, yet software production in the large would be enormously helped by the availability of spectra of high quality routines, quite as mechanical design is abetted by the existence of families of structural shapes, screws or resistors." [McIlroy69]

The software engineering community had to undergo a painful and long learning process to accept the fact that the specification of various kinds of pieces of software is not only a topic of theoretical interest but also one of practical importance. Therefore it seems quite natural that finally we can find a considerable number of research activities and published results starting from the mid-nineties with e.g. [Liskov94], and [Zaremski97]). On the practical side, the basic foundations have been laid by Bertrand Meyer with his concept of Design by Contract™ (DbC) as realized in the Eiffel language (see [Meyer92a] and [Meyer92b]).

Thus, what has been common in the area of hardware design for approximately twenty years under the name design for testability, viz. to enhance the product with means to increase the observability of its behavior, only recently seems to gain some attention also within software industry. This is all the more strange, as software has gained a still and rapidly increasing part of responsibility for the well functioning of almost all the (really or only seemingly) important things of every day life.

Some facts that certainly can be regarded as indicators of a growing attention on this kind of software development are collected in the following list:

- In the second half of the nineties we can find an increasing interest in Eiffel, with SmallEiffel[1] becoming the GNU Eiffel system;
- a steadily increasing number of papers dealing with the concepts and the practical use of assertions in general have been published during the last few years (have a look to the references for some examples);
- the Assertion Definition Language (ADL)[2] developed at Sun Labs;
- newer Methods like Syntropy [Cook94], the Eiffel-related BON[3] and Catalysis[4] [D'Souza99], provide means for the inclusion of assertions into their graphical models;
- the Unified Modeling Language[5] (UML) has as one of its parts the Object Constraint Language[6] (OCL), which has its root in the Syntropy method;
- emerging support for assertions for Java (from which even the ANSI C assert mechanism has been removed) and C++ (see e.g. the iContract tool[7] and Jass[8], or the overview in section 3.4 of [Maley00] for C++).

If we want to implement some contract mechanism, the first question that arises is "How to (re)build the Eiffel DbC mechanism in C++ or Java ?". Or, in a more pushing

[1] See http://smarteiffel.loria.fr/
[2] See http://adl.opengroup.org/
[3] See http://www.bon-method.com/book_main.htm
[4] See http://www.trireme.com/catalysis/ or http://www.catalysis.org
[5] See http://www-306.ibm.com/software/rational/uml/
[6] See http://www.klasse.nl/ocl/index.html
[7] See formerly at http://www.reliable-systems.com/tools/iContract/iContract.htm
[8] See http://semantik.informatik.uni-oldenburg.de/~jass/

ahead fashion: Are there any relevant new insights concerning the topic of assertion based programming during the last ten years that should be taken into account?

Note 1: Using the term 'assertion based programming' indicates that our focus is on the role of assertions in software artefacts, i.e. in the current paper we do not consider their way from requirements into code. Some more aspects of assertion based software development are considered in section 4.

Note 2: It was a definite goal to get this paper self-contained to a reasonable degree. Thus, expert readers will find some material already known to them but hopefully associated with some new aspects. The great benefit of this approach, however, is for newcomers to this area of software engineering who do not need to resort to other sources during their very first steps into the new domain.

The paper is organized as follows: Section 2 provides the general conceptual background used subsequently: the various kinds of assertions are introduced, as well as the notions of specification and specification matching. The Liskov Substitution principle is presented as one of the cornerstones of reasoning about object-oriented software. In Section 3 we have a more detailed look on assertions in various software contexts and what the concepts of section 2 do mean in practical life, whereas section 4 takes a broader perspective and presents some hints on specification activities beyond the limits of mere code throughout the whole development process. A concluding section follows.

I hope this paper will be said to have been written in the spirit of the both very wise and very tolerant view formulated by none other than Donald Knuth himself as follows [Knuth91]:

"The best theory is inspired by practice. The best practice is inspired by theory."

And I am sure, he is one of the few persons who must definitely know this from his own experience and work.

2 THE CONCEPTUAL FRAMEWORK

In order to provide a suitable bridge between the theoretical and practical aspects of assertion usage, this section is devoted to the presentation of the basic notions and the terminology that will be used during the rest of the paper.

Peace of Code (PoC) is a generic name that we use for denoting routines (methods), classes, modules, libraries, programs, and components. In using this general term we do, of course, not forget about the wide range of complexity and the different requirements and challenges inherent to the above mentioned kinds of software.

PoCs are made up of *features* (see e.g. [Mitchell02], p.21): Features are either *attributes* or *routines* (or *methods*). Some routines are *functions*, i.e. routines that return a value but do not change attributes. The other routines are *procedures*, some of them creators and others modifiers of objects. Attributes and functions together form the set of *queries* (return a result but do not change the visible properties of an object), procedures are the *commands* that clients can use to change or create objects. Commands do not return results.

Determining the behavioral relationship between PoCs is a central task for many software engineering activities such as reasoning about, reuse, extend or maintain code. Two basic notions in this context are behavioral subtyping and substitutability. The kind of behavioral relationship is determined by the assertions associated to the PoCs.

Assertions: What and why.

Software *assertions* are Boolean expressions that define the correct state of a program at a particular location in the code. Hence, assertions are not part of the working code, they say something about it, i.e. they are on a meta and not the coding level. Assertions can check method calls for proper invocation, method code for correct computation, class data states for consistency, and also individual statements for errors. Therefore, assertions may act in the following different roles:

- *Preconditions* express the requirements that clients must satisfy whenever they call a PoC, and are therefore evaluated at their entry point. Preconditions are obligations for the client, and benefits for the server.
- *Postconditions* inform about what the supplier (i.e. the PoC) guarantees on return, if the precondition has been satisfied on entry. They have to be evaluated at all(!) exit points of the PoC - if you allow more than one in your coding standard. Postconditions are obligations for the server, and benefits for the client.
- In case you are participating in an OO project: *Class Invariants* define the consistency conditions for the state space of a class and must be satisfied by every instance of the class whenever this instance is externally accessible, i.e. after creation, and before and after any call to a public routine. Class invariants have to be evaluated at the entry and all exit points of all externally visible methods of a class. (Note that an invariant (i) is universal across an entire class and (ii) has a normative impact not only on already existing, but also on all methods eventually added in the future.)
- *Data assertions* define conditions that must hold at a particular location in the code, whence they are evaluated only at their location in the code. You can take them for your own individually tailored and eventual only temporary tests. Data assertions do not contribute to software contracts in the sense outlined above and will not be further considered in this paper.
  (A special case of data assertions provided in the Eiffel language [Meyer92a] are loop invariants and loop variants: Like class invariants, which give a description of the internal consistency of the class as a whole, *loop invariants* characterize what must hold in each repetition of a loop. A *loop variant* is a strictly monotonic decreasing function that guarantees that the loop will terminate; its expression must remain positive, and it must decrease with each iteration.)

Specifications and specification matches

Following [Chen00] we choose the generic and well-known $\langle$requirement, offer$\rangle$ or $\langle$query, answer$\rangle$ or $\langle$problem, solution$\rangle$ situation as our general context of discussion because it covers all aspects we want to consider in more detail: Component retrieval uses the $\langle$query, component$\rangle$, software reuse the $\langle$client specification, method specification$\rangle$, and behavioral subtyping the $\langle$base class specification, derived class specification$\rangle$ pair. In fact, all three of them are examples of a general software reuse problem. Let us first recall some notions from [Chen00] in

Definition 1. (a) A *specification* is a pair of predicates $\langle P_{pre}, P_{post}\rangle$, where – as indicated - P_{pre} specifies the precondition, and P_{post} the postcondition of a party involved in a deal. E.g. in the situation of component retrieval $\langle Q_{pre}, Q_{post}\rangle$ would describe the specification of the query, and $\langle C_{pre}, C_{post}\rangle$ the one of component C.

(b) For a program or method C and a specification $\langle p, q\rangle$ the *correctness formula* (Hoare triple) $\{p\}C\{q\}$ is informally interpreted as the truth of "program C started with p satisfied will terminate in a state such that q holds" (total correctness).

(c) Given a query specification Q: $\langle Q_{pre}, Q_{post}\rangle$, a component *$C$ is reusable for implementing Q*, if $\{Q_{pre}\}C\{Q_{post}\}$ holds.

(d) Specification matching is a method for finding reusable components fulfilling a query by matching the component with the query specification. Formally, a *specification match* is a function M: **Spec** $\times$ **Spec** $\rightarrow$ {**true, false**}, and given a match M and two specification S_1 and S_2 we say "S_1 matches S_2 according to M" if $M(S_1, S_2)$ = **true**.

(e) A specification match M is *reuse ensuring*, i.e. can ensure that a component C satisfies a query Q, if and only if for any C and Q, $M(C,Q) \wedge \{C_{pre}\}C\{C_{post}\} \rightarrow \{Q_{pre}\}C\{Q_{post}\}$.

A number of specification matches have been proposed for defining behavioral sub-typing and component retrieval. *Figure 1* focuses on the first topic, whence the match functions are cast into the terminology of class and subclass (instead of query and component), denoted as C and SC (instead of Q and C), respectively; it is an extension of Figure 4 of [Penix99] which comprises the matches (4), (5), (6), (9) and (10) only. Here are the names of the specification matches:

(1) *exact pre/post* (as in [Zaremski97] and [Chen00])
(2) *exact pre*
(3) *exact post*
(4) *plug-in* (as in [Zaremski97])
(5) *weak plug-in* (as in [Penix99]; called *guarded plug-in* in [Zaremski97])
(6) *satisfies* (as in [Penix99]; called *relaxed plug-in* in [Chen00], *plug-in compatibility* in [Fischer97], and also used in [Liskov01], p. 176)
(7) *satisfies-and*
(8) *guarded generalized predicate* (as in [Chen00])
(9) *plug-in post*
(10) *weak post* (as in [Penix99]; called *guarded post* in [Zaremski97])

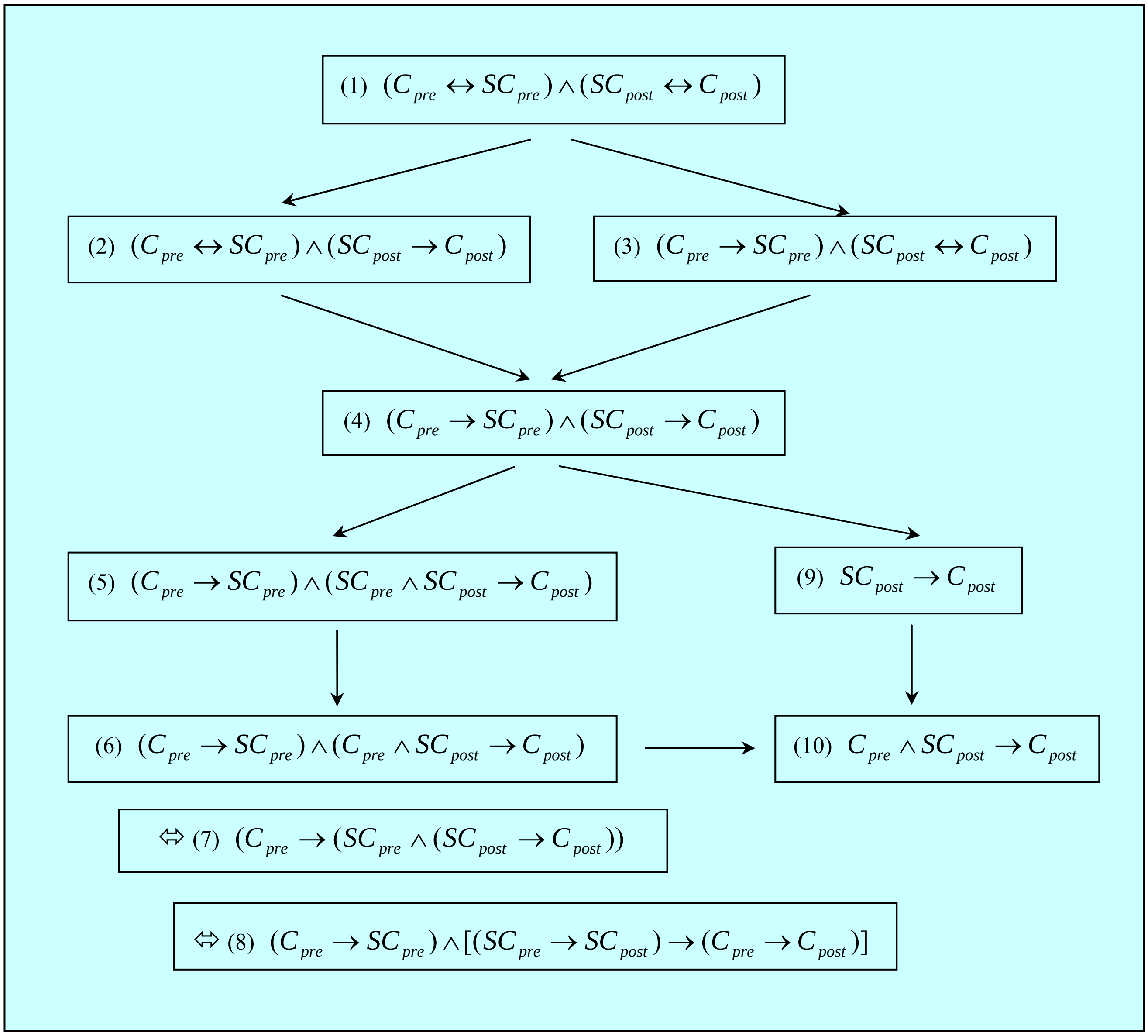
(1) $(C_{pre} \leftrightarrow SC_{pre}) \wedge (SC_{post} \leftrightarrow C_{post})$
(2) $(C_{pre} \leftrightarrow SC_{pre}) \wedge (SC_{post} \rightarrow C_{post})$
(3) $(C_{pre} \rightarrow SC_{pre}) \wedge (SC_{post} \leftrightarrow C_{post})$
(4) $(C_{pre} \rightarrow SC_{pre}) \wedge (SC_{post} \rightarrow C_{post})$
(5) $(C_{pre} \rightarrow SC_{pre}) \wedge (SC_{pre} \wedge SC_{post} \rightarrow C_{post})$
(9) $SC_{post} \rightarrow C_{post}$
(6) $(C_{pre} \rightarrow SC_{pre}) \wedge (C_{pre} \wedge SC_{post} \rightarrow C_{post})$
(10) $C_{pre} \wedge SC_{post} \rightarrow C_{post}$
$\Leftrightarrow$ (7) $(C_{pre} \rightarrow (SC_{pre} \wedge (SC_{post} \rightarrow C_{post}))$
$\Leftrightarrow$ (8) $(C_{pre} \rightarrow SC_{pre}) \wedge [(SC_{pre} \rightarrow SC_{post}) \rightarrow (C_{pre} \rightarrow C_{post})]$

Figure 1: Specification matches

Note: An arrow between two matches indicates that the match at the base of the arrow is stronger than (logically implies) the match at its end. The formal notation is abbreviated by dropping the quantifiers and variables for the predicates.

It has been shown in [Chen00] that *exact pre/post*, *plug-in*, *satisfies*, and *guarded generalized predicate* are reuse ensuring matches, and that *satisfies* and *guarded generalized predicate* are logically equivalent, i.e. (6) ⇔ (8).

Plug-in compatibility

As can be seen from the list above, *satisfies* or *plug-in compatibility* is a predominant match used in current literature. Furthermore, by Theorem 7 in [Chen00] it is a most general reuse-ensuring match, which is also shown by the arrow diagram in *Figure 1*.

What other arguments for and properties of *satisfies* have been given? In [Liskov01] the match is split up in two rules: The *precondition rule* ($C_{pre} \rightarrow SC_{pre}$) means a weakening of the precondition, i.e. the subtype method requires less from its caller than the supertype method does. Thus we can be sure that the call to a subtype method will be legal if the call to the supertype method is. The effect of a call is taken into account by the *postcondition rule* ($C_{pre} \wedge SC_{post} \rightarrow C_{post}$) which means a strengthening of the postcondition, i.e. the subtype method delivers more to its caller than the supertype method does. The calling code depends on the supertype method's postcondition only if the call also satisfies its precondition; hence, the postcondition rule's antecedent is restricted to legal inputs of the supertype method.

In the context of component retrieval, *plug-in compatibility* supports safe reuse [Fischer97]: Components found under this match function may be considered as black boxes and can be reused "as is" without any further modification.

Some other aspects concerning the matches (4), (5), (6), (9) and (10) are mentioned in [Penix99]: *plug-in, weak plug-in*, and *satisfies* all require the precondition of the class (query) to be stronger then that of the subclass (component). They differ in the set of inputs for which a valid output is assured by the postcondidtion: *plug-in* checks the whole domain, *weak plug-in* restricts the check to the legal inputs of the component, and *satisfies* further restricts to the legal inputs of the problem. The *plug-in post* and *weak post* matches differ from the above three by not requiring all legal problem inputs to be legal component inputs; thus there could be problem inputs that cause unspecified behavior of the component. However, for any legal problem input that is also a legal component input, a valid output is computed. Therefore, components that match in one of these two ways provide a partial solution to the problem.

Inheritance and behavioral subtyping

As already mentioned in section 1, the object-oriented approach is very powerful for developing large software systems. Much of this power is due to the key concept of inheritance. However, the statically checks enforced by e.g. C++ or Java compilers upon derived classes test for such syntactic and typing restrictions only that guarantee the lack of runtime type errors. This is the contracting and specification level that has been used for too many years in the past by most software developers. Obviously, this is not enough to prevent surpising and often disastrous behavior of programs.

That means, that checks done by compilers are only part of what is needed to reason about the behavior (i.e. the semantics) of software, especially for object-oriented systems when new subtypes are added. Behavioral subtyping is a technique for preventing unexpected behavior in a modular way: it ensures that any reasoning that has been done about the behavior of a piece of client code that uses objects of a base class C continues to hold if the code is instead applied to objects of a subclass SC of C, i.e. it remains valid if calls to a method *m* are dispatched to m_{SC} instead of to m_C. This is the case if methods redefined in subclasses satisfy their base class specification, i.e. if they fulfill a reuse ensuring specification match. Thus, objects of new subtypes (instances of subclasses) "act like" objects of their supertypes, when used as if they were supertype objects. This is what the *Liskov Substitution Principle* (LSP) for object-oriented design states [Liskov88]:

- *In class hierarchies, it should be possible to treat a specialized object as if it were a base class object.*
- In other words: *Object-oriented functions that use pointers or references to a base class must be able to use objects of a derived class without knowing it.*

The basic idea here is as simple as it is important: Inheriting classes should not perform any actions that will invalidate the assumptions made by (the client of) a parent class. Put differently, any object of a subtype must be substitutable for an object of a supertype in the hierarchy without any effect on the program's observable behavior. If the Liskov Substitution Principle is followed, code using a base class pointer will never break after another class has been added to the inheritance tree.

Modular reasoning

An important concern in both object-oriented and component-based software development is how to reason about the extension of programs. The key requirement to be satisfied by any disciplined use of implementation inheritance (syntactic reuse) is the preservation of *modular reasoning*: It must be possible to establish properties of code using the static types of its expressions (especially of pointers to objects) without need to inspect any of the subclasses involved. That is, if the static type is *T*, then the dynamic type of the expression's value must be a subtype of *T*. As argued above in connection with the LSP, this is not enough since the semantics is not taken into account: In order to preserve modular reasoning it is necessary that each subtype used in the program is a behavioral subtype of each of its supertypes.

Modular reasoning is of paramount importance for extensible software systems, in which the set of subclasses of a given class is open. The advantage of modular reasoning is that unchanged methods of client code do not have to be respecified and reverified when new behavioral subtypes are added to class libraries.

3 ASSERTIONS IN SOFTWARE ENGINEERING PRACTICE

In this section we will look how some of the concepts introduced above can be transferred to software systems in practice. Bertrand Meyer has coined the phrase "Design by Contract" (DbC) to denote a software development style which (1) emphasises the importance of formal specifications, and (2) interleaves them with actual code. DbC is a systematic method of assertion usage and interpretation introduced as a standard feature of the Eiffel language [Meyer92a]. Without it, no trial would have ever been made to provide a similar mechanism in other languages and, by no means, would we have discussion papers like this and the ones mentioned in the references.

What is Design by Contract™ ?

Software contracts have been invented to capture mutual obligations and benefits among classes, as they are e.g. needed in design patterns, where each of the involved classes is expected to exhibit a "proper" behavior. A software contract is the specification of the behavior of a class and its associated methods. The contract outlines the responsibilities of both the caller and the method being called. Failure to meet any of the responsibilities stated in the contract results in a breach of the contract, and indicates the existence of a bug somewhere in the design or implementation of the software or - one must not forget this possibility in earlier project phases - in the assertions themselves. Software contracts can be completely specified through the use of preconditions, postconditions, and class invariants in object-oriented software.

DbC views software construction as based on contracts between clients (callers) and suppliers (routines). Each party expects some benefits from the contract, and accepts some obligations in return. As in human affairs, the contract document spells out these mutual benefits and obligations and protects both he client, by specifying how much should be done, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope.

The DbC paradigm is as follow: The client's obligation is to call a method only in a program state where both the class invariant and the method's precondition hold. The method, in return, guarantees that the work specified in the postcondition has been done, and the class invariant is still respected. A precondition violation thus points out an error of the client, and a postcondition failure a bug in the implementation of the routine, which did not fulfill its promise. (**Note:** The phrase "An assertion fails" in real life means just the opposite: the assertion did its job well, because it has found a bug.)

DbC is, in a way, the opposite of *defensive programming*, a method which recommends to protect every software module by as many checks as possible. This may result in redundancy and makes it also difficult to precisely assign responsibilities among modules.

Class correctness

Software contracts are a necessary prerequisite for being able to introduce a notion of correctness: If you do not state what your program should do, you are lacking the norm to which to compare what your program does in reality. In defining class correctness we follow [Meyer97], p. 370 (remember Definition 1,(b)):

Definition 2. A class C is *correct* with respect to its specification if

1. For any set of valid arguments x_p to a creation procedure p:

   $\{Default_C \wedge pre_p(x_p)\}\ \ p\ \ \{post_p(x_p) \wedge INV_C\}$
2. For every public method m and any set of valid arguments x_m:

$$\{pre_m(x_m) \ \wedge \ INV_C\} \ \ m \ \ \{ \ post_m(x_m) \ \wedge INV_C\}$$

where $Default_C$ denotes the assertion expressing that the attributes of C have the default values of their type.

Contracts and inheritance.

The basic rule governing the relationship between inheritance and assertions is that in a descendant class all the ancestors' contracting assertions (i.e. routine pre- and postconditions, and the class invariants) still apply. (Remember: data assertions have only local impact.)

Inheritance of assertions guarantees that the behavior of a class is compatible with that of its ancestors: The assertions specify a range of acceptable behaviors for the routine and its eventual redefinitions which may specialize this range, but not violate it. In other words, as an honest subcontractor, as soon as you accept a contract, you must be willing to do the job originally requested, or better than that, but not less. For use in the subsequent considerations let us introduce some terminology in

Definition 3. (a) An assertion A is said to be *stronger than* another assertion B, if it logically implies it, i.e. A $\rightarrow$ B. If A is stronger than B, then B is said to be *weaker than* A.

(b) What is actually checked at runtime for a derived class is usually called the *effective* or *accumulated assertion*.

We will use the following terminology: Let C denote a class; we take $C := S^0C$, and for $k \geq 1$ $S^kC := SS^{k-1}C$. Then we assume that SC is a subclass of C, SSC a subclass of SC, and generally that for $k \geq 1$ S^kC is a subclass of $S^{k-1}C$.

Failure and success conformance

A contracting assertion (i.e. a precondition, a postcondition, or a class invariant) is said to be in a *contractor position* if it is used in a non-OO procedural context or in a base class, and it is in a *subcontractor position*, if it is used in a derived class.

Definition 4. An (effective) assertion is *failure (success) conformant* if it is in a contractor position (the special and trivial case), or – if in a subcontractor position – if it fails (succeeds) exactly if all assertions of the same kind of all its parent classes also fail (succeed). In other words: if failure (success) can always be seen from the base class specification.

More formally, using the generic term "effective constraint", we have

- success conformance: if $effCon_{S^nC}$, then also $effCon_{S^{n-1}C}$.
- failure conformance: if $\neg effCon_{S^nC}$, then also $\neg effCon_{S^{n-1}C}$.

An important question is: What do missing failure or success conformance mean in practice ?

	effective preconditions	*effective postconditions*
success conformance	*Some cases, where subclass methods could be used, are invisible from base class specification.* *⇒ Clients miss some opportunities for valid use of subclass methods.*	*Some cases, where subclass methods would do correct computation, are invisible from base class specification.* *⇒ Clients miss some cases of desired results from subclass methods.*
failure conformance	*Some cases, where subclass methods can not be used, are invisible from base class specification.* *⇒ Clients get an error where they do not expect one (since a subclass pre-condition fails).*	*Some cases, where subclass methods would do incorrect computation, are invisible from base class specification.* *⇒ Clients may not get delivered a result where they expect one (since a subclass postcondition fails).*

The straightforward conclusion is that failure conformance poses far more serious problems: Whereas missing success conformance does not result in bad situations but only in non-optimal organization of the work to be done, this is obviuosly not true when failure conformance is not available.

Some laws of propositional logic

(T1) $A \wedge B \rightarrow A$
(T2) $A \rightarrow A \vee B$
(T3) **True** $\wedge A \equiv A$
(T4) **False** $\vee A \equiv A$
(T5) $A \rightarrow$ **True** $\equiv$ **True**
(T6) $A \rightarrow$ **False** $\equiv \neg A$
(T7) **True** $\rightarrow A \equiv A$
(T8) **False** $\rightarrow A \equiv$ **True**
(T9) $A \rightarrow B \equiv \neg A \vee B$
(T10) $A \wedge (A \rightarrow B) \equiv A \wedge B$
(T11) $A \wedge (B \rightarrow A) \equiv A$
(T12) $A \wedge B \rightarrow C \equiv A \rightarrow (B \rightarrow C)$
(T13) $A \rightarrow (B \rightarrow C) \equiv B \rightarrow (A \rightarrow C)$
(T14) $(A \rightarrow C) \wedge (B \rightarrow C) \equiv (A \vee B) \rightarrow C$
(T15) $(A \rightarrow B) \wedge (A \rightarrow C) \equiv A \rightarrow (B \wedge C)$

Table 1: Logical laws

The common percolation pattern and its properties

For Java and C++ I do not know of any tool or macro or class API that does not follow the Eiffel model. So there is need to give a few remarks on it. Eiffel builds the assertions of derived classes by OR-ing (for preconditions) and AND-ing (for postconditions and class invariants), respectively, along the class hierarchy as shown below. This means that – instead of really checking the hierarchy properties of contracts on method level specifications – Eiffel constructs a correct hierarchy on the level of effective contracts. This kind of traversing the class hierarchy has been called the *percolation pattern* in [Binder99, Binder00].

Let $effPre_C$:= C_{pre}, and for $k \geq 1$ $effPre_{S^kC} := S^kC_{pre} \vee effPre_{S^{k-1}C}$; thus $effPre_{S^{k-1}C} \rightarrow effPre_{S^kC}$, i.e. an effective superclass precondition implies that of its subclass by (T2). In an analogous way, using $\wedge$ instead of $\vee$ the effective postconditions and class invariants are constructed for derived classes, e.g. with $effPost_C := C_{post}$, and for $k \geq 1$ $effPost_{S^kC} := S^kC_{post} \wedge effPost_{S^{k-1}C}$; thus $effPost_{S^kC} \rightarrow effPost_{S^{k-1}C}$ i.e., an effective subclass postcondition implies that of its superclass, and likewise for effective class invariants, by (T1).

Let us consider a simple example with the three classes *C*, *SC*, and *SSC*; thus, by our convention *SSC* is a subclass of *SC*, and *SC* a subclass of *C*. We further focus our attention on a call to a method m_{SSC} redefined in *SSC* and, therefore, take a closer look at some of the possible configurations for m_{SSC} with the help of *Table 2*.

	C_{pre}	SC_{pre}	$effPre_{SC}$	SSC_{pre}	$effPre_{SSC}$
1	**false**	**false**	**false**	**false**	**false**
2	**false**	**false**	**false**	**true**	**true**
3	**false**	**true**	**true** $\longrightarrow$	**false**	**true**
4	**true** $\longrightarrow$	**false**	**true**	**false**	**true**
5	**true**	**true**	**true**	**true**	**true**

Table 2 Percolation pattern example for preconditions

(Remember: The effective precondition of e.g. m_{SSC} is given as $effPre_{SSC} := C_{pre} \vee SC_{pre} \vee SSC_{pre}$, and $effPre_C := C_{pre}$. In lines 3 and 4, denotes a $\longrightarrow$ precondition hierarchy violation)

Analysing *Table 2* for preconditions we see that the first two lines do not cause any problems: Line 1 reflects the failure conformance of effective preconditions in the percolation pattern, i.e. if $\neg effPre_{S^nC}$, then also $\neg effPre_{S^{n-1}C}$; and line 2 represents the case where m_{SSC} accepts the call on basis of its own precondition. Also the situation shown in line 5 is straightforward.

But line 2 also shows a serious deficiency of the percolation pattern: it does not provide success conformance for preconditions, i.e. if $effPre_{S^nC}$, then also $effPre_{S^{n-1}C}$. Thus, modular reasoning for clients of C is impossible on the basis of its specification. So, even in the absence of any proper hierarchy violation, the percolation pattern can cause considerable irritation of C's clients.

We also have some problems connected to lines 3 and 4: Consider line 3 which has a precondition hierarchy violation from SC_{pre} to SSC_{pre}, since $SC_{pre} \rightarrow SSC_{pre} \equiv$ **false**. m_{SSC} will be executed although SSC_{pre} itself evaluates to **false**, because due to OR-percolation $effPre_{SSC}$ becomes **true**. (Note: Even if we had $SSC_{pre} \equiv$ **true**, clients of C would get cheated, since from C_{pre} they learn that performing m_C or an overriding method like e.g. m_{SSC} would not be executed, whereas m_{SSC} in fact will be performed.) As an example imagine that $SC_{pre}(p) := p < 0$ and $SSC_{pre}(p) := p \geq 0$, and take $p = -5$. This, I suspect, could give rise to some surprises. (This deficiency has first been mentioned in [Karaorman99, section 4.1].) Now for line 4: It gives an example where, although there is a hierarchy violation C_{pre} to SC_{pre}, a client of C would not get cheated what concerns the execution of m_C or any of its overriding methods. But (s)he should be prepared for some surprises if – due to polymorphism - m_{SC} or m_{SSC} would be called: for both of them the effective precondition evalutes to **true**, whereas the method specific one to **false**.

Of course, we can easily find analogous situations for postconditions as can be seen from *Table 3*:

	C_{post}	SC_{post}	$effPost_{SC}$	SSC_{post}	$effPost_{SSC}$
1	**false**	**false**	**false**	**false**	**false**
2	**false**	**false** $\longleftarrow$	**false**	**true**	**false**
3	**true**	**false**	**false**	**false**	**false**
4	**true**	**true**	**true**	**false**	**false**
5	**true**	**true**	**true**	**true**	**true**

Table 3: Percolation pattern example for postconditions

(Remember: The effective postcondition of e.g. m_{SSC} is given as $effPost_{SSC} := C_{post} \wedge SC_{post} \wedge SSC_{post}$; and $effPost_C := C_{post}$. In line 2, denotes a $\longrightarrow$ postcondition hierarchy violation)

Line 1 is straightforward, and line 6 reflects the success conformance of effective post-conditions in the percolation pattern, i.e. if $effPost_{S^nC}$, then also $effPost_{S^{n-1}C}$. In line 2, although $SSC_{post} \equiv$ **true**, the execution of m_{SSC} would be reported to have produced a wrong result, since $effPost_{SSC} \equiv$ **false**. This is obviously due to the postcondition hierarchy violation $SSC_{post} \rightarrow SC_{post} \equiv$ **false**. Only a hierarchy error can cause this kind of undesired effect.

Lines 3 and 4 are examples of the missing failure conformance of postconditions (i.e. if $\neg effPost_{S^nC}$, then also $\neg effPost_{S^{n-1}C}$), again cheating a client who reasons on basis of *C*'s specification. Such a situation occurs in case $S^nC_{post} \equiv$ **false** and $S^{n-1}C_{post} \equiv$ **true**, whence $S^nC_{post} \rightarrow S^{n-1}C_{post}$ gives a valid logical implication. This demonstrates that implication validity is not a suitable criterion for hierarchy correctness. (Be aware, that the specification matches shown in *Figure 1*are focused on reusability.)

The Eiffel approach fulfills the plug-in specification match; effective preconditions are failure but not success conformant, and effective postconditions and class invariants are success but not failure conformant. Thus, the widely used percolation pattern is neither pre-LSP nor post-LSP, and therefore it does not support modular reasoning. In other words, a reuse ensuring match is, in general, not sufficient for enabling modular reasoning.

Besides these problems, the task of assigning blame for malformed class (and - in case of Java - interface) hierarchies is a difficult one that is not performed correctly by existing tools; for an example see [Findler01].

Alternative ways for hierarchy checks

Due to the above mentioned disadvantages of the commonly used percolation pattern there is the need to search for alternatives. For that purpose we will take a closer look at three of the specification matches presented in *Figure 1*: plug-in, weak plug-in, and relaxed plug-in (satisfies). We will focus our discussion on a fixed (and therefore almost never explicitly mentioned) example method m_C defined in base class *C*, with redefinitions m_{SC} and m_{SSC} in subclasses *SC* and *SSC*, respectively.

(1) Let us first turn to the preconditions' part, which is the same for all three specification matches we will examine. Throughout our discussion we assume the class hierarchy to be correct, i.e. the superclass precondition implies that of its subclass. For a method we will check its specific precondition as well as the correctness of the class hierarchy. Thus, we get e.g. the following effective preconditions:

$$C \;:\; C_{pre}$$
$$SC \;:\; SC_{pre} \;\wedge\; (C_{pre} \rightarrow SC_{pre})$$
$$SSC: SSC_{pre} \wedge\; (SC_{pre} \rightarrow SSC_{pre}) \;\wedge (C_{pre} \rightarrow SC_{pre})$$

Generalizing this gives

$$effPre_{S^nC} := S^nC_{pre} \;\wedge\; preHierCheck_n, \qquad \text{(pre)}$$

where: $preHierCheck_0 := \textbf{true},$

$$preHierCheck_n := (S^{n-1}C_{pre} \rightarrow S^nC_{pre}) \;\wedge preHierCheck_{n-1}$$

$$= \bigwedge_{i=0}^{n-1} \left(S^i C_{pre} \rightarrow S^{i+1} C_{pre} \right) \qquad \text{(preh)}$$

Since we assume the hierarchy to be correct, i.e. *preHierCheck*$_n$ evaluates to **true** (or *preHierCheck*$_n$ for short), this means that $S^k C_{pre} =$ **true** implies $\forall j \geq k$: $S^j C_{pre} =$ **true** (use (T8) and (T10) from *Table 1*). We thus have

$$effPre_{S^n C} \equiv \bigwedge_{i=k}^{n} S^i C_{pre} \,. \qquad \text{(epre)}$$

(2) Things become a little bit more complicated for postconditions. In order to ensure a reasonable basis for the comparison between the different checking strategies, we assume all the involved assertions and hierarchy checks to be fulfilled, as well as the method to be implemented correctly.

(2a) Plug-in match: A correct class hierarchy for plug-in match means that the subclass postcondition implies that of its superclass. As in the precondition case, for a method we will check its specific postcondition as well as the correctness of the class hierarchy. Thus, we get e.g. the following effective postconditions:

C : C_{post}

SC : $SC_{post} \wedge (SC_{post} \rightarrow C_{post})$

SSC: $SSC_{post} \wedge (SSC_{post} \rightarrow SC_{post}) \wedge (SC_{post} \rightarrow C_{post})$

Generalizing this gives

$$effPost_{S^n C} := S^n C_{post} \wedge postHierCheck_n, \qquad \text{(p)}$$

where: $postHierCheck_0 :=$ **true**,

$$postHierCheck_n := (S^n C_{post} \rightarrow S^{n-1} C_{post}) \wedge postHierCheck_{n-1}$$

$$= \bigwedge_{i=1}^{n} \left(S^i C_{post} \rightarrow S^{i-1} C_{post} \right) \qquad \text{(ph)}$$

Since we assume the hierarchy to be correct, i.e. *postHierCheck*$_n$, this means that $S^k C_{post} =$ **true** implies $\forall j \leq k$: $S^j C_{post} =$ **true** (use (T8) and (T10) from *Table 1*). We thus have

$$effPost_{S^n C} \equiv \bigwedge_{i=0}^{n} S^i C_{post} \,. \qquad \text{(ep)}$$

(2b) Weak plug-in match: Under the same assumptions as before we get e.g. the following effective postconditions:

C : C_{post}

SC : $SC_{post} \wedge (SC_{pre} \wedge SC_{post} \rightarrow C_{post})$

SSC: $SSC_{post} \wedge (SSC_{pre} \wedge SSC_{post} \rightarrow SC_{post}) \wedge (SC_{pre} \wedge SC_{post} \rightarrow C_{post})$

Generalizing this gives

$$w\text{-}effPost_{S^n C} := S^n C_{post} \wedge w\text{-}postHierCheck_n, \qquad \text{(w)}$$

where: $w\text{-}postHierCheck_0$:= **true**,

$$w\text{-}postHierCheck_n := (S^nC_{pre} \wedge S^nC_{post} \rightarrow S^{n-1}C_{post}) \wedge w\text{-}postHierCheck_{n-1}$$

$$\equiv \bigwedge_{i=1}^{n} \left(S^iC_{pre} \wedge S^iC_{post} \rightarrow S^{i-1}C_{post}\right)$$

$$\equiv \bigwedge_{i=1}^{n} \left(S^iC_{pre} \rightarrow \left(S^iC_{post} \rightarrow S^{i-1}C_{post}\right)\right) \qquad \text{(wh)}$$

Assuming S^nC_{pre} and a correctly implemented method we also have S^nC_{post}, whence we get $S^{n-1}C_{post}$ from (wh). Turning to the general case that $S^kC_{pre}, k \leq n$ then we know that $\Lambda_{i=k}^{n} S^iC_{pre}$; thus we have $\Lambda_{i=k}^{n}\left(S^iC_{post} \rightarrow S^{i-1}C_{post}\right)$ from (wh), and – since we also assume the hierarchy to be correct, i.e. $w\text{-}postHierCheck_n$ – using logical law (T10) from *Table 1* we finally get

$$\Lambda_{i=k-1}^{n} S^iC_{post} \qquad \text{(ew)}$$

(2c) Relaxed plug-in match (satisfies): As for weak plug-in above, we assume all parts involved to be correct and valid, repectively. Thus, we get e.g. the following effective postconditions:

C : C_{post}

SC : $SC_{post} \wedge (C_{pre} \wedge SC_{post} \rightarrow C_{post})$

SSC: $SSC_{post} \wedge (SC_{pre} \wedge SSC_{post} \rightarrow SC_{post}) \wedge (C_{pre} \wedge SC_{post} \rightarrow C_{post})$

Generalizing this gives

$$r\text{-}effPost_{S^nC} := S^nC_{post} \wedge r\text{-}postHierCheck_n, \qquad \text{(r)}$$

where: $r\text{-}postHierCheck_0$:= **true**,

$$r\text{-}postHierCheck_n := (S^{n-1}C_{pre} \wedge S^nC_{post} \rightarrow S^{n-1}C_{post}) \wedge r\text{-}postHierCheck_{n-1}$$

$$\equiv \bigwedge_{i=1}^{n} \left(S^{i-1}C_{pre} \wedge S^iC_{post} \rightarrow S^{i-1}C_{post}\right)$$

$$\equiv \bigwedge_{i=1}^{n} \left(S^{i-1}C_{pre} \rightarrow \left(S^iC_{post} \rightarrow S^{i-1}C_{post}\right)\right) \qquad \text{(rh)}$$

Assuming S^nC_{pre} and a correctly implemented method we have S^nC_{post}, but not more in this case. Turning to the general case that $S^kC_{pre}, k \leq n$ then we know that $\Lambda_{i=k}^{n} S^iC_{pre}$; thus we have $\Lambda_{i=k}^{n-1}\left(S^{i+1}C_{post} \rightarrow S^iC_{post}\right)$ from (rh), and – since we also assume the hierarchy to be correct, i.e. $postHierCheck_n$ - using logical law (T10) from *Table 1* we finally get

$$\Lambda_{i=k}^{n} S^{i} C_{post} \qquad \text{(er)}$$

Comparing the three alternatives, we see that (ep) induces stronger postcondition constraints than (ew), which is stronger than (er): (ep) → (ew) → (er) as was to be expected from the relationships between the underlying specification matches. Is there any gain compared to the percolation pattern ? Definitely yes:

- Hierarchy errors are detected in all three versions for both pre- and post-conditions.
- Never will a routine with its own precondition evaluating to **false** be executed (remember *Table 2*, lines 3 and 4).

However, concerning failure and success conformance, there is practically no improvement as the following tables show. Only the effective postconditions for plug in match can easily be seen to be success conformant. (The '**?**' signs indicate that the associated precondition is not relevant in this case.)

	C_{pre}	SC_{pre}	$effPre_{SC}$	SSC_{pre}	$effPre_{SSC}$
1	**false**	**false**	**false**	**true**	**true**
2	**true**	**true**	**true**	**false**	**false**

(epre)

	C_{post}	SC_{post}	$effPost_{SC}$	SSC_{post}	$effPost_{SSC}$
1	**true**	**true**	**true**	**true**	**true**
2	**true**	**true**	**true**	**false**	**false**

(ep)

	C_{pre} / C_{post}	SC_{pre} / SC_{post}	$effPost_{SC}$	SSC_{pre} / SC_{post}	$effPost_{SSC}$
1	**? / true**	**false / true**	**false**	**true / true**	**true**
2	**? / true**	**false / true**	**true**	**true / false**	**false**

(ew)

	C_{pre} / C_{post}	SC_{pre} / SC_{post}	$effPost_{SC}$	SSC_{pre} / SC_{post}	$effPost_{SSC}$
1	**false / true**	**false / false**	**false**	**? / true**	**true**
2	**false / true**	**false / true**	**true**	**? / false**	**false**

(er)

How to make a choice among the three alternatives analysed above? Some arguments we propose are the following:

- Failure of success conformance for postconditions is probably the least harmful deficiencey.
- Bookkeeping of a method's own precondition evaluation seems to be easier to implement than that of its superclass method – at least at a first glance.
- The difference between (ew) and (er) is that the former does an additional check of $S^{n-1}C_{post}$ only; thus the strengthening from (er) to (ew) is not very high, whence the loss of reuse possibilities should be modest.

In summary, based on these considerations it seems reasonable to choose (ew) as the favourite checking strategy.

4 SUMMARY AND CONCLUSIONS

The main purpose of this paper was to present a thorough analysis of the commonly used Eiffel mechanism (also called 'percoaltion pattern') for dealing with contracts in class hierarchies. It has been shown that there are some inherent deficiencies in this strategy. As alternatives we have proposed three variants of so-called reuse ensuring specification matches, and it has been shown that they also do not avoid all of the problems we have found for the percolation pattern, but should nevertheless be preferred due to the fact that they really check the class hiearchy for behavior conformance, and make it impossible that a subclass method gets activated if its own precondition evaluates to false.

During our analysis of *Table 3* we have also seen that the use of implication validity as a criterion for hierarchy correctness may perhaps be an unsuitable choice. But this point needs further consideration.

In order to fulfil my promise from Note 2 in section 1 concerning the self-containedness of this paper, let me give a list with the most important aspect and benefits of DbC.

- Writing contracts from the very beginning enforces developers to state what they are trying to do already during design. This definitely helps doing it right!
- Assertion based programming in general, and DbC as a systematic and well defined variant of it, should be regarded as built in self tests and as a permanent online review that both help in early error detection and give the software developers a feeling about the correctness status of their programs.
- Even a systematic use of contracts is, of course, not an a-priori static proof that some properties of the software are valid for all legal data sets; instead, it is a dynamic mechanism that checks the code for each single data set applied. At best you get something like an “inductive proof” that increases the confidence in the correctness of programs checked in this way.
- Contracts can, and usually should, also serve as a basis for (technical) documentation, especially for an up-to-date description of interfaces.

- Contracts enable the top developers to express the intent behind their designs and hence to leave behind a clear statement of the original design, reducing the risk that further contributors or maintainers will destroy the software's consistency and quality.
- Contracts are a remarkable testing and debugging tool: (i) they save debugging time due to the improved observability, where failures occur close to the bugs; (ii) they express unambiguously what an author expects and what he garuantees in turn. So one has a clear statement not only of What Is (in the implementation part) but also of What Should Be (in the assertions part). And only under such circumstances one can definitely identify a discrepancy between the two; and (iii) they help you in designing and performing your tests.
- You gain seamlessness, i.e. it is possible to use a single notation and a single set of concepts throughout the software life cycle, from analysis and design, to implementation and maintenance.
- You may also overcome the programmer's usual resistance against what they feel excessive documentation requests burdened upon them by higher management, if you advise them to write documentation in a form that has immediate operational impact on their work.
- You can - and this is the most important of all the benefits - apply a win-win strategy using DbC with obvious advantages for both your customer and you as the developer of a software system (object-oriented or not).
- "Self-standing contractually specified interfaces decouple clients and providers. The same interface may be used by a large number of different clients but also be supported by a large number of different providers." ([Szyperski98], p.72)

All these ideas are not as new as you might perhaps believe from section 1. They date back to at least the late 60's and the early 70's. There is indeed a long and tedious way in formal methods from the assigning of meanings to programs in [Floyd67] and the axiomatic basis of [Hoare69] to Abrial's B-book [Abrial96] with its emphasis on assigning programs to meaning. Today's average software engineer usually does not make use, and often does not even know, of formal methods and corresponding tools. However, I think it is high time to take advantage of some of their findings. DbC or a newer improved variant of it may eventually prove to be the bridge between current practice and theory.

REFERENCES

[Abrial96] J.-R. Abrial: *The B-Book: Assigning Programs To Meanings*, Cambridge University Press, 1996.

[Chen00] Y. Chen/B.H.C. Cheng, A Semantic Foundation for Specification Matching. In G.T. Leavens and M. Sitaraman (Eds.), *Foundations of Component-Based Systems*, Cambridge Univ. Press, 2000, 91-109.

[Binder99] R. Binder, The Percolation Pattern – Techniques for Implementing Design by Contract in C++, *C++ Report*, May 1999, 38-44.

[Binder00] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 2000.

[Cook94] S. Cook/J. Daniels, Designing Object Systems: Object-Oriented Modelling With Syntropy, Prentice Hall, 1994.

[D'Souza99] D. D'Souza/A.C. Wills, Objects, *Components, and Frameworks with UML. The Catalysis Approach*, Addison Wesley, 1999.

[Findler01] R.B. Findler/M. Latendresse/M. Felleisen, Behavioral Contracts and Behavioral Subtyping. In *Proceedings of ACM Conference Foundations of Software* Engineering (FSE 2001), 2001

[Fischer97] B. Fischer/G. Snelting, Reuse by Contract, Proc. ESEC/FSE-Workshop on Foundations of Component-Based Systems, Zürich, September 1997, 91-100.

[Floyd 67] R.W. Floyd, Assigning meaning to programs. In Proc. of Symposia in Applied Mathematics, Mathematical aspects of computer science Vol. 19, American Mathematical Society, 1967, 19-32.

[Hoare69] C.A.R. Hoare, An axiomatic basis for computer programming, *Communications of the ACM*, Oct. 1969, 576-583.

[Hoare73] C.A.R. Hoare, Hints on Programming Language Design, Stanford University Artificial Intelligence memo AIM224/STAN-CS-73-403. Reprinted in [Hoare89], 576-583.

[Hoare89] C.A.R. Hoare/C.B. Jones (Eds.), *Essays in Computing Science* (reprints of Hoare's papers), Prentice Hall, 1989.

[Karaorman99] Karaorman, M./U. Hölzle/J. Bruno, jContractor: A Reflective Java Library to Support Design By Contract, in: Proceedings of Meta-Level Architectures and Reflection, 2nd International Conference, Reflection '99. Saint-Malo, France. *Lecture Notes in Computer Science* #1616, Springer Verlag, 1999, pp. 175-196.

[Knuth91] D.E. Knuth, Theory and Practice, *Theoretical Computer Science* **90** (1991), 1-15.

[Kramer98] R. Kramer, iContract - The Java^{TM} Design by Contract^{TM} (formerly available at http://www.reliable-systems.com/tools/iContract/iContract.htm)

[Liskov88] B. Liskov, Data Abstraction and Hierarchy, *SIGPLAN Notices* **23**(5), May 1988.

[Liskov94] B. Liskov/J.M. Wing, A behavioral notion of subtyping, *ACM Trans-actions on Programming Languages* **16**, 1811-1841, November 1994.

[Liskov01] B. Liskov (with J. Guttag), *Program Development in Java. Abstraction, Specification, and Object-Oriented Design*, Addison-Wesley, 2001.

[Maley00] D. Maley, Mind the Gap – Formalising the Development of Scientific Software. Ph.D. Theses, Queen's University of Belfast, March 2000.

[McIlroy69] M.D. McIlroy, Mass produced software components. In P. Naur and B. Randell (Eds.), Software Engineering. Report on a Conference Sponsored by the NATO Science Committee, Garmisch,Germany, 7th to 11th October, 1968, 138–150. NATO Science Committee, 1969; reprinted in P. Naur, B. Randell, and J.M. Buxton (Eds.), Software Engineering: Concepts and Techniques. Petrocelli/Charter Publishers Inc., New York, NY, 1976, 88-98.

[Meyer92a]Meyer, B., *Eiffel – the Language*, Prentice Hall, 1992

[Meyer92b]Meyer, B., Applying "Design by Contract", *IEEE Computer* **25**(10), 40-51, October 1992

[Meyer97] Meyer, B., *Object oriented software construction*, 2nd Ed., Prentice Hall, 1997.

[Mitchell02] R. Mitchell/J. McKim, *Design by Contract, by Example*, Addison-Wesley, 2002.

[Penix99] J. Penix/P. Alexander, Efficient Specification-Based Component Retrieval, *Automated Software Engineering* **6**, 139-170, 1999.

[Szyperski98] C. Szyperski, *Component Software. Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[Zaremski97] A.M. Zaremski/J.M. Wing, Specification Matching of Software Components, *ACM Transactions on Software Engineering and Methodology* **6**(4), 333-369, October 1997.

About the author

Herbert Toth is a senior software engineer in the Program and System Engineering Department of SIEMENS AG Austria. He holds a diploma degree in computer science from the Technical University, and a Ph.D. degree in mathematical logic from the University, both in Vienna. During the eighties and nineties his research interests led to some publications on the foundations of fuzzy set theory. He can be reached at mailto:herbert.toth@siemens.com.