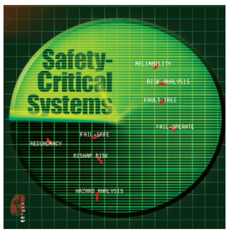
COVER FEATURE

Uncovering Hidden Contracts: The .NET Example



Can libraries written without explicit support for design by contract benefit from adding contracts? We studied classes from the .NET collections library for implicit contracts and assessed improvements that might result from making them explicit.

Karine Arnout
Bertrand Meyer
ETH (Swiss Federal Institute of Technology)

Commercial relationships and business contracts, which formally express the rights and obligations binding a client and a supplier, provide the inspiration for design by contract software development methodology. Software contracts take the form of routine preconditions (obligations on the caller), postconditions (benefits to the caller), and class invariants (consistency constraints), written into the program itself. The design by contract methodology uses such contracts for building each software element.[1-3] This approach is particularly appropriate for developing safety-critical software and for reusable libraries. The Ariane 5 accident makes a textbook case for the precision that design by contract offers in specifying reusable components.[4]

The methodology is a key design element of some existing libraries,[1] especially the Eiffel software development environment, which incorporates contract mechanisms in the programming language itself. As long-time practitioners of design by contract techniques, we see their benefits clearly and are surprised that recent languages and libraries, including the official libraries for Java and .NET and the C++ Standard Template Library, have not adopted them. Because we see the contract metaphor as inherent to quality software development, we undertook the work reported here as a sanity check: Do we see contracts everywhere simply because our development environment makes using them natural? Or are they intrinsically present, even when other designers don't express or even perceive them?

To answer these questions, we examined noncontracted libraries for *hidden contracts*—that is, for language or documentation techniques that suggest contract mechanisms such as precondition and postcondition clauses. Our first target is the Collections classes of the .NET Framework Class Library,[5] the most recent addition to the world's collection of fundamental data structure and algorithm libraries.

LOOKING FOR WHAT ISN'T OFFICIALLY THERE

This search has sent us rummaging through interface specifications, documentation, even generated code—which in .NET, as in Java, still retains significant high-level information.

Because we are looking for something that officially isn't there, we must exercise our judgment to claim and authenticate our findings, stating explicitly why we think a particular class characteristic, such as an exception, represents an underlying contract. We have therefore performed the work manually, not relying so far on any automatic tools.

A manual extraction process puts a natural limit on future extensions of this analysis to other libraries. Automated extraction tools could help, and our results suggest certain patterns in code and documentation that point to possible contracts—just as certain geological patterns point to possible oil deposits. The final contract-elicitation process, starting from noncontracted libraries, will always, however, require subjective decisions.

WHY USE CONTRACTS?

Some programmers who do know about con-

Published by the IEEE Computer Society 0018-9162/03/$17.00 © 2003 IEEE

tracts see them as just a way to help test and debug programs through conditionally compiled instructions of the form

```
if not "Some condition I expect to
    hold here" then
  "Scream"
end
```

where "Scream" might involve triggering an exception or stopping execution altogether. However, such a use—similar to the C language's "assert"—is only a small part of the methodology and would not by itself justify special language constructs.

Contracts address a much wider range of software process issues for general application development as well as library design:

- *Correctness.* Contracts help build software right in the first place by avoiding bugs rather than correcting them after they appear. Designers are encouraged to think about each software element's abstract properties and to build observance of them into the software.
- *Documentation.* Development environments such as Eiffel that support software provide automatic tools to extract documentation that is both abstract and precise. The information comes from the software text, saving the effort of writing documentation as a separate product and eliminating the risk of divergence between software and documentation.
- *Debugging and testing.* Monitoring contracts at runtime supports an effective form of quality assurance: Since the code now describes not only what the program does but what it is supposed to do, runtime checking uncovers the discrepancies between intent and realization.
- *Inheritance control.* A coherent approach to inheritance limits the extent to which routine redefinitions can affect the original semantics. The language rule only permits weakening preconditions and strengthening postconditions in descendants of the original class.
- *Management.* Project managers and decision makers can understand a program's global purpose without having to go into the code in depth.

The principles are particularly relevant to library design. Eiffel libraries are equipped with contracts stating their abstract properties, thereby helping users write robust software as well as debug and document it.

Contract elements

Contracts express the semantic specifications of classes and routines. They are made of assertions—Boolean expressions stating individual semantic properties. For example, a class representing lists stored in a container of bounded capacity might require the number `count` of list elements not to exceed the maximum permitted capacity. Contract elements include

Eiffel libraries help users write robust software as well as debug and document it.

- *preconditions*—requirements under which a routine will function properly. A precondition is binding on clients (callers); the supplier (the routine) can turn a precondition to its advantage by simplifying its algorithm to assume the precondition.
- *postconditions*—properties the supplier guarantees to the client on routine exit.
- *class invariants*—semantic constraints characterizing the integrity of instances of a class. Each constructor (creation procedure) must ensure these constraints, and every exported routine must maintain them.
- *check instructions*—an assert-like construct, often used on the client side to check that a precondition is satisfied as expected.
- *loop variants and invariants*—correctness conditions for a loop.

Check instructions, loop variants, and loop invariants address implementation correctness rather than library interface properties, so we will not consider them further here.

Although preconditions and postconditions are the best-known forms of library contracts, class invariants are particularly important in an object-oriented context. Class invariants express fundamental properties of the abstract data type underlying a class and the correctness of the representation invariant—the abstract data type implementation chosen for the class.[6]

Inherent contracts in libraries

Even a simple example shows the usefulness of contracts in library design. Consider a square root function specified as

```
sqrt (x: REAL): REAL
```

In this specification, the function takes a *REAL* argument and returns a *REAL* result.

Although some Java and .NET documentation just calls it a "contract," this type signature's spec-

The metadata concept gives .NET substantial specification information that appears directly useful to the contract-elicitation process.

ification defines the function's *signature contract*. The more broadly accepted term for "contract," which we use here, is *semantic contract*, specifying important properties of the argument and result that type information alone can't capture. An example is what happens for a negative argument.

A contract—in the form of a precondition and a postcondition—expresses which specification the function implements. In Eiffel the function would appear as

```
sqrt (x: REAL): REAL is
    -- Mathematical square root
    -- of x, within epsilon
  require
    non_negative: x >= 0
  do
    ... Square root algorithm
  ensure
    good_approximation:
      abs(Result ^2 - x)
      <= 2 * x *epsilon
  end
```

where `epsilon` is some appropriate value expressing the requested precision, `abs` gives the absolute value, and ^ is the power operator. The assertion tags "non_negative" and "good_approximation" are for documentation. They also provide a more precise error message if the execution violates an assertion, assuming the programmer has enabled runtime contract monitoring for testing or debugging.

Here we find direct support for the contract clauses—`require`, `ensure`, and the yet-to-be-encountered `invariant`—in the language and the associated documentation standard. The contract is, more generally, a property of the library's design and its interface to application programmers: To use a square-root function properly, programmers must know the conditions under which it will operate and what properties they can expect of its result.

This example raises two general questions:

- If there is no explicit contract discipline, comparable to the Eiffel practice of documenting all libraries through assertions, where will we find the implicit contract?
- Will the contract always exist, as in this example, whether it is expressed or not?

Our study of .NET libraries provides material for answering these questions.

WHY .NET LIBRARIES?

We used .NET libraries for our study not only because they are recent and widely publicized collections of general-purpose reusable components but also because the metadata concept gives them substantial specification information that appears directly useful to the contract-elicitation process.

Metadata

The basic compilation unit for a .NET library is the *assembly*. The key to the framework's support for component-based development is that every assembly includes metadata providing documentary information that makes it self-describing, in accordance with the design by contract self-documentation principle.[2]

In addition to predefined categories—assembly name, version, dependencies, and so on—with proper source-language support, developers can use custom attributes to define their own specific kinds of metadata.

.NET Contract Wizard

Both predefined and custom metadata open attractive new possibilities for analyzing and developing libraries. We built a prototype .NET Contract Wizard[7] that uses metadata to examine a class and its features so that users can add appropriate contracts interactively through a graphical interface—without having access to the source code.

By nature, however, the Contract Wizard is useful only if contracts are intrinsically present in good software libraries. This observation is one of the incentives for our study: As we consider further development of the Contract Wizard, we must first gather empirical evidence confirming or denying its usefulness. If contracts add nothing useful and noncontracted .NET libraries do very well without them, continuing our work on the Contract Wizard would be a waste of time.

COLLECTION CLASS ANALYSIS

We first scouted the class `ArrayList`, part of the core .NET library (mscorlib.dll), for hidden contracts. Our choice was almost arbitrary. In particular, we did not guess beforehand that the class would suggest either more or fewer contracts than any other.

As an informal criterion, `ArrayList` seemed typical of the .NET Collections library style, and it has obvious practical use since it describes lists implemented through arrays. It also has a counterpart in EiffelBase, *ARRAYED_LIST*, opening the way to comparisons after we completed the contract-elicitation process.

The code extracts below represent the different Eiffel and .NET conventions. For example, Eiffel uses *count*, and .NET uses Count. Likewise, the Eiffel classes *ARRAY_LIST* and *ARRAYED_LIST* match .NET's ArrayList.

Implicit class invariants

The documentation comments revealed ArrayList properties in the class invariants category.

For example, the class constructors specification states, "The default initial capacity for an ArrayList is 16." This comment implies that the created object's capacity is greater than zero. Taking this lead, we noticed that all three ArrayList constructors set the initial list's capacity to a positive value. According to design by contract rules, which require that all of a class's creation procedures must guarantee an invariant property, this consistency suggests a class invariant.

The other key invariant property is that all of the class's exported routines must preserve it. Examining the documentation of all routines showed the property to be true for ArrayList, indicating that this property had the characteristics of an invariant, which Eiffel would express by the clause

```
invariant
  positive_capacity: capacity >= 0
```

The documentation further revealed that two of the three ArrayList constructors "...initialize a new instance of the ArrayList class that is empty." The number of elements Count for an arrayed list created this way must then be zero.

The third constructor, which takes a collection c as a parameter, "initializes a new instance of the ArrayList class that contains elements copied from the specified collection." So the new object's number of elements equals the number of elements in the collection passed as argument; we can express this through the assertion Count = c.Count.

It is unlikely that c.Count could be negative. Checking the documentation further revealed that the argument passed to the constructor can denote any nonvoid collection, represented by any of the many classes inheriting from the ICollection interface.[5]

According to the specification of the routine Remove in ArrayList, "The average execution time is proportional to Count. That is, this method is an $O(n)$ operation, where n is Count." This specification implies that Count must always be nonnegative. The evidence is strong enough to suggest adding a clause to the above invariant:

```
positive_count: count >= 0
```

These first two properties are simple but already useful.

Next, we examined the class members' specification. Documentation on the Count property revealed interesting information: "Count is always less than or equal to Capacity." This statement indicates that this class property always holds, suggesting a third invariant property for the ArrayList class and the accumulated clause

```
invariant
  positive_capacity: capacity >= 0
  positive_count: count >= 0
  valid_count: count <= capacity
```

Design by contract rules require that all of a class's creation procedures must guarantee an invariant property.

Implicit routine preconditions

In addition to implicit class invariants, the documentation also suggested preconditions. For example, the documented exception cases for class ArrayList require the routine Add to throw an exception of type NotSupportedException if the arrayed list on which it is called is read-only or has a fixed size. This suggests that the underlying implementation of Add first checks that the call target is writable (not read-only) and extendible (not fixed-size) before actually adding elements to the list.

Imposing such a requirement on a method is the way design by contract defines a routine precondition. An Eiffel Add specification would then include the following precondition:

```
require
  writable: not is_read_only
  extendible: not is_fixed_size
```

where *is_read_only* and *is_fixed_size* are the Eiffel counterparts of the .NET properties IsReadOnly and IsFixedSize of class ArrayList.

This example, one of many in the .NET framework reference documentation,[5] suggests a systematic scheme for extracting preconditions:

- Read the exception conditions—for example, the array list is read-only.
- Take the opposite—for ArrayList, the condition would be **not** *is_read_only*.

Eiffel policy supports object-oriented development, providing a seamless process from analysis to design and implementation.

- Infer the underlying routine precondition—in this case, `writable: not is_read_only`.

A systematic scheme, in turn, opens the door to the possibility of automated tool support.

Implicit routine postconditions

Does the .NET documentation also reveal hidden postconditions?

To explore this question, we considered the query `IndexOf`. More precisely, since `IndexOf` is an overloaded method, we chose a specific version identified by its signature:

```
public virtual int IndexOf
(Object value);
```

The documentation explains that the return value is "... the zero-based index of the first occurrence of value within the entire `ArrayList` if found; otherwise, –1."

We can rephrase this specification more explicitly:

- If `value` appears in the list, the result is the index of the first occurrence, hence greater than or equal to zero (.NET lists are indexed starting at zero) and less than `Count`, the number of elements in the list.
- If `value` is not found, the result is –1.

This property is guaranteed on routine exit—that is, it is incumbent on the supplier as a guarantee of correct completion of its task. As such, it represents a postcondition. In Eiffel, we would add a corresponding clause to the routine:

```
ensure
  valid_result_if_found:
    contains (value) implies
      Result > = 0 and Result
        < count
  correct_index_if_found:
    contains (value) implies
      item (Result) = value
  minus_one_if_not_found:
    not contains (value) implies
      Result = -1
```

This analysis suggests that routine postconditions do exist in .NET libraries, although they are not explicitly expressed because they lack support from the underlying environment. Unlike with preconditions—for which it may be possible to devise supporting tools—extracting postconditions is likely to require case-by-case human examination because evidence is scattered across the reference documentation.

Interface contracts

Class `ArrayList` implements three "interfaces"—completely abstract specification modules—of the .NET collections library: `IList`, `ICollection`, and `IEnumerable`.

We subjected these interfaces to the same analysis as the class. We found that `IList`, `ICollection`, `IEnumerable`, and `IEnumerator`—of which `IEnumerable` is a client—have routine preconditions and postconditions similar to those of `ArrayList`. We did not, however, find class invariants in these interfaces. This is probably because .NET interfaces have a limited scope compared with "deferred classes," their closest counterpart in the object-oriented model that Eiffel embodies.

Deferred classes can have a mixture of abstract and concrete features. In particular, they can include attributes. Interfaces, on the other hand, are purely abstract and cannot contain attributes. Eiffel policy provides a continuous spectrum of classes, from totally deferred—the equivalent of .NET and Java interfaces—to fully implemented. This policy supports the aims of object-oriented development, providing a seamless process from analysis, which typically uses deferred classes, to design and implementation, which make the classes progressively more concrete. Class invariants in the Eiffel libraries often express consistency properties binding various attributes together.[8]

The classes implementing a given interface can share a set of properties that would be candidates for interface invariants. Our analysis of the .NET collections library did not, however, reveal any such properties.

ADDING CONTRACTS

The discovery of hidden contracts in the .NET arrayed list class suggested building a "contracted variant" of this class, *ARRAY_LIST*, that has the same interface as the original `ArrayList` plus the elicited contracts.

Rather than modifying the original class, we can produce the contracted variant in Eiffel as a new class whose routines call those of the original class. This was the only solution, anyway, since we did not have access to the source code. The Contract Wizard supports such a process, but for this study we produced the result manually.

Contracted .NET arrayed list class

Reviewing a few more features helps devise contracts for the Add and IndexOf routines.

The contracted variant of feature Add with the two preconditions, tagged "writable" and "extendible," also expresses a postcondition (with different clauses) because its elicitation process is similar to the process for the IndexOf query. The Add header comment helps understand the contracts:

```
add (value: ANY): INTEGER
    -- Add value to the end of the
    -- list (double list capacity
    -- if the list is full)
    -- and return the index at
    -- which value has been added.
  require -- from ILIST
    writable: not is_read_only
    extendible: not is_fixed_size
  ensure -- from ILIST
    value_added: contains (value)
    updated_count:
      count = old count + 1
    value_is_last_item:
      item (count - 1) = value
    valid_index_returned:
      Result = count - 1
  ensure then
    capacity_doubled:
      (old count = old capacity)
        implies (capacity = 2 *
        (old capacity))
```

The comment `-- from ILIST` shows that the precondition and postcondition clauses are inherited from the parent class *ILIST*. This class is the contracted version of the .NET IList interface that class ArrayList implements.

The precondition of feature *add* uses two queries of class *ARRAY_LIST*: *is_read_only* and *is_fixed_size*. These queries correspond to the properties IsReadOnly and IsFixedSize from the .NET collections class ArrayList.

The *add* postcondition relies on other queries: *contains* (to check whether an item is in the list), *item* (to have access to any list item given its position, which must be a valid index for the list), *count*, and *capacity* (for measurements). The *index_of* contracts also rely on these routines:

```
index_of (value: ANY): INTEGER
    -- Zero-based index of first
    -- occurrence of value
  ensure -- from ILIST
    valid_result_if_found:
      contains (value)
        implies Result >= 0
        and Result < count
    correct_index_if_found:
      contains (value) implies
        item (Result) = value
    minus_one_if_not_found:
      not contains (value)
        implies Result = -1
```

Writing reusable components requires contracts, and many contracts are there even if not formally expressed.

Metrics

Measurements of properties of the contracted class *ARRAY_LIST* show that 62 percent of the routines (33 of 52) now have a contract (a precondition or a postcondition, usually both). The 33 routines with preconditions tend to have more than one precondition clause—2.5 on average (82 precondition clauses total); the 33 routines with postconditions tend to have more than one postcondition clause—2 on average (67 postcondition clauses total). These figures seem to support the view that writing good reusable components requires contracts and that many contracts are there even if not formally expressed.

EXTENDING TO OTHER CLASSES AND INTERFACES

Equipped with our first results on ArrayList and its contracted Eiffel counterpart, ARRAY_LIST, we performed similar transformations and measurements on a few other classes and interfaces of the .NET collections library.

Interfaces

First, we considered Eiffel deferred classes obtained by contracting the .NET interfaces that ArrayList uses:

- *ILIST*, *ICOLLECTION*, *IENUMERABLE*, from which *ARRAY_LIST* inherits.
- *IENUMERATOR*, of which *IENUMERABLE* is a client.

Table 1 shows the results obtained using the Eiffel environment's Metrics tool to measure the contract rate for some .NET collection interfaces. These statistics highlight three trends:

- Absence of class invariant in the .NET interfaces, as already noted.
- Presence of routine contracts, both preconditions and postconditions. The numbers for

Table 1. Contract rate of some .NET collection interfaces.

	ILIST	*ICOLLECTION*	*IENUMERABLE*	*IENUMERATOR*
Routines	11	4	1	6
Routines with preconditions	7	1	0	3
Routines with postconditions	7	1	1	3
Number of preconditions	14	7	0	4
Number of postconditions	11	1	2	3
Preconditions rate (%)	64	25	0	0
Postconditions rate (%)	64	25	100	50
Class invariants	0	0	0	0

IENUMERABLE and *ICOLLECTION* involve too few routines to offer valuable information. The numbers for *ILIST* and *IENUMERATOR* are more significant, with at least one-half of the routines for both classes having contracts.

- Presence of multiple routine contracts, with most routines having several preconditions and postconditions.

The last two points are consistent with the properties observed for class *ARRAY_LIST*.

Other classes: Stack and Queue

To test the generality of our first results on `ArrayList`, we considered two other classes of the Collections library, `Stack` and `Queue`. We selected these two classes because they are concrete collection classes with no relation to `ArrayList`, except that all three implement the .NET `ICollection` and `IEnumerable` interfaces. In addition, `Stack` and `Queue` also have direct counterparts in the EiffelBase library.

We must be careful with any generalization of the results of our analysis because three classes is still only a small sample of the library. Any absolute conclusion would require an exhaustive automated analysis. With these qualifications, applying the same manual approach to `Stack` and `Queue` as for `ArrayList` and its parents confirmed the previously identified trends. Specifically, both classes include preconditions and postconditions. Class `Stack` has a 29 percent preconditions rate (5 of 17 routines had preconditions) and a 59 percent postconditions rate (10 of 17 routines had postconditions). Class `Queue` has respective rates of 42 percent (8 out of 19) and 58 percent (11 out of 19).

Further, we also found that routines usually have several precondition and postcondition clauses. For example, `Stack` has 16 postcondition assertions for only 10 routines equipped with contracts.

Finally, concrete classes have class invariants. For example, all three classes—`ArrayList`, `Stack`, and `Queue`—have an invariant clause

```
positive_count: count >= 0
```

involving one attribute: *count*.

AUTOMATIC CONTRACT EXTRACTION

Is it possible to synthesize contracts automatically rather than manually?

Dynamic contract inference

Dynamic inference seeks to derive assertions from captured variable traces by executing a program with various inputs. It relies on a set of possible assertions to deduce contracts from the execution output, and it requires the source code to be available. Dynamic inference determines whether the detected assertions are meaningful and useful to the users, typically by computing a confidence probability.

Michael D. Ernst's Daikon[9] is an example of such a tool. Daikon tries to find class and loop invariants as well as routine preconditions and postconditions. Its first version was limited to finding contracts over scalars and arrays; the next version, Daikon 2, enables contract discovery over data collections and computes conditional assertions.

There are also some Java detectors that operate directly on bytecode files (*.class), and some of them do not require the program source code to infer contracts.

Extracting routine preconditions from exception cases

Because we do not have the source code of the .NET libraries, we cannot rely on a dynamic contract inference tool such as Daikon. We need to find another way to automate the contract-extraction process.

Our analysis identified some clear patterns in the form and location of implicit contracts in existing .NET components. In particular, preconditions tend to be buried under exception cases. Since method exception cases are not kept in the assembly metadata, we are currently exploring another approach: inferring routine preconditions by parsing the Common Intermediate Language code of .NET libraries to list the exceptions that a method or a property may throw.

Methodological perspective

The question of automatic analysis is legitimate but should not obscure the underlying methodological issue: Contracts offer the most benefits if, as in Eiffel, they are used as part of the development process rather than after the fact. No automatic tool will derive the best contracts—those expressing the abstract intent behind a concrete implementation.

For reusable components, contracts serve as the specification: the set of properties that describe to potential component users what a component can do for them, independently of how it does that. The component authors should make these properties explicit—not leave them for an automatic tool to figure out.

This observation doesn't invalidate the potential benefits of automatic contract extraction, an area that we and others are continuing to investigate. But it does limit the expectations that we can place on it. There is no magic. We should encourage designers to think of contracts as a great help in the design process, not as a burden. Automatic contract extraction remains interesting as a complement to explicit methods and as an aid to improving the quality of existing software that did not specify contracts in its design.

Our analysis provides a first step in a broader research plan, which we expect to expand in several directions. First, we will apply the same approach to other libraries, such as C++ STL. Second, to facilitate the work of programmers interested in adding contracts a posteriori to existing libraries, we will investigate the patterns that help discover each contract type more closely. To accomplish this, we will continue our work on developing an interactive tool to support this process. Finally, we plan to turn the Eiffel Contract Wizard into a Web service to allow any programmer to contribute contracts to .NET components.

Many opportunities exist for extending this work to a broad investigation of design by contract applications. For example, Kevin McFarlane has conducted a project to provide a .NET contract framework.[10] We anticipate that such developments will help improve reusable components through explicit and implicit contracts. ■

Acknowledgments

We thank Éric Bezault (Axa Rosenberg), Michael D. Ernst (MIT), Tony Hoare (Microsoft), and Emmanuel Stapf (Eiffel Software) for their extremely valuable comments and insights.

References

1. B. Meyer, "Applying Design by Contract," *Computer*, Oct. 1992, pp. 40-51.
2. B. Meyer, *Object-Oriented Software Construction*, 2nd ed., Prentice Hall, 1997.
3. R. Mitchell and J. McKim, *Design by Contract, by Example*, Addison-Wesley, 2002.
4. J.M. Jezequel and B. Meyer, "Design by Contract: The Lessons of Ariane," *Computer*, Jan. 1997, pp. 129-130.
5. Microsoft, ".NET Systems.Collections Library," 2003; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollections.asp.
6. C.A.R. Hoare, "Proof of Correctness of Data Representations," *Acta Informatica*, vol. 1, 1973, pp. 271-281.
7. K. Arnout and R. Simon, "The .NET Contract Wizard: Adding Design by Contract to Languages Other than Eiffel," *Proc. TOOLS 39*, IEEE CS Press, 2001, pp. 14-23.
8. B. Meyer, *Reusable Software: The Base Object-Oriented Component Libraries*, Prentice Hall, 1994.
9. M.D. Ernst, *Dynamically Detecting Likely Program Invariants*, doctorial dissertation, Univ. of Washington, 2000; http://buffy.eecs.berkeley.edu/Seminars/2000/03.Mar/000302.ernst.html.
10. K. McFarlane, "Design by Contract Framework for .NET," 2002; www.codeproject.com/csharp/designbycontract.asp; www.codeguru.com/net_general/designbycontract.html.

***Karine Arnout** is a PhD student at ETH (Swiss Federal Institute of Technology), Zurich. While working at Eiffel Software in Santa Barbara, Calif., she contributed to porting Eiffel to the .NET framework and built the first implementation of the Eiffel contract wizard for .NET. Her research interests are in the area of trusted components, in particular the transformation of design patterns into reusable components and the correlation between contracts and tests. Contact her at Karine.Arnout@inf.ethz.ch.*

***Bertrand Meyer** is professor of software engineering at ETH, an adjunct professor at Monash University, and founder and chief architect of Eiffel Software in Santa Barbara, Calif. His interest is quality software development. Contact him at Bertrand.Meyer@inf.ethz.ch.*