Cite this article as follows: Karine Arnout, Bertrand Meyer, *Finding Implicit Contracts in .NET components*, to appear in *Formal Methods and Components* (FMCO), eds. Frank de Boer et al., Springer-Verlag, 2003

Finding Implicit Contracts in .NET Libraries

Karine Arnout[1] Bertrand Meyer[1, 2]

[1] Chair of Software Engineering, Swiss Federal Institute of Technology (ETH)
CH-8092 Zurich, Switzerland
[2] Eiffel Software, 356 Storke Road, Santa Barbara CA 93117, USA

Karine.Arnout@inf.ethz.ch
http://se.inf.ethz.ch http://www.eiffel.com

Keywords: Design by Contract™, Library design, Reuse, Implicit contracts, .NET, Metadata, Contract Wizard, Eiffel.

Abstract. Are contracts inherent in reusable libraries, or just one design technique among others? To help answer this question, we performed an empirical study of library classes from the .NET Collections library, which doesn't use Design by Contract™, to look for unexpressed contracts. This article reports on the buried contracts we have found, and discusses improvements to the architecture — especially to the libraries' ease of learning and ease of use — that may result from making the contracts explicit. It extends previous reports [3] [4] [5] [6] with an analysis of the benefits of an a posteriori addition of contracts for the library users.

1 Introduction

Equipping libraries with contracts has become a second nature to designers working with Eiffel. Many commonly used libraries, however, don't show any contracts at all. The resulting style is very different, and, to someone used to Design by Contract [21] [23] [25] [31], deficient.

Because the benefits of contracts are so clear to those who use them, it's natural to suspect that non-Eiffel programmers omit contracts because they have no good way to express them, or haven't even been taught the concepts, but that conceptually contracts are there all the same: that inside every contract-less specification there is a contract wildly signaling to be let out.

For an Eiffel programmer this is the natural interpretation. But when you are doing something different from the rest of the world, it's good to check your own sanity. Are we wrong in seeing contracts around libraries, and the rest of the world — including the most recent general-purpose development frameworks — right in continuing to act as if contracts had never been invented?

This article is such a sanity check. The basic conjecture that it explores may be stated more precisely:

Box 1. The Closet Contract Conjecture

Eiffel libraries have contracts; most others don't. Which is the right explanation?

- The contract-rich style of Eiffel libraries is but an artefact of the support for contracts in the Eiffel method, language and tools. Remove contract mechanisms, and the contracts just go away.
- Contracts are inherent in library design; if not explicitly stated, as in C++/Java/.NET libraries, they are lurking anyway under the cover, either suppressed or replaced by comments in the program, explanations in the documentation, exceptions and other ersatz techniques.

Resolving the Closet Contract Conjecture is interesting for several reasons:

- The answer can shed light on important issues of reusable component design, one of the keys to progress in software engineering.
- An answer can help library users (application programmers) choose between competing libraries.
- On a more specific point, the answer would help ascertain the potential usefulness of a "Contract Wizard". Such a tool, of which a first version has been implemented by Eiffel Software [1], takes advantage of the reflection facilities of .NET — "Metadata" — to let its users work interactively on compiled classes, coming from any non-contracted language such as C#, C++, Visual Basic, Cobol or Java, and add contracts to them a posteriori. But this is only interesting if the second answer holds for the Closet Contract Conjecture. If not, the Wizard's user wouldn't find any interesting contracts to add.
- If that second answer indeed holds, we may use it to improve our understanding of the libraries, and even to improve the library themselves by turning the implicit contracts that we have elicited into explicit elements of the software.

To help answer the conjecture, we have started a study of non-contracted libraries to see if we could spot implicit contracts. The .NET collection library [27], a comprehensive set of data structure implementations, has been one of the first targets. We examined some commonly used .NET collection classes, sleuthing around for hidden contracts, and trying to uncover language or documentation techniques used to make up for the absence of proper contract mechanisms such as precondition clauses, postcondition clauses and class invariants.

Where we spotted closet contracts, we proceeded to out them, by producing class variants that retain the original APIs but make the contracts explicit. We compared the result both with the originals and with some of the classes' closest counterparts in the EiffelBase library.

The rest of this presentation describes the analysis and its outcomes:

- Section 2 provides more details on why we have engaged in this study and explains the method of analysis.
- Section 3 recalls the principles of Design by Contract and their application to library design.
- Section 4 summarizes the contributions of .NET and its Collections library.
- Section 5 presents the results of analyzing an important .NET collection class: `ArrayList` [28].
- Section 6 introduces a variant of `ArrayList` where the implicit contracts detected in the official version have been made explicit, and gives measurements of properties of the contracted class.
- Section 7 extends the analysis to some other classes and interfaces to assess the consistency of the results across the library.
- Section 8 compares the effect of the two design styles, with and without contracts, on the development of applications using a library: ease of use, ease of learning, bug avoidance.
- Section 9 presents related work about contract extraction and evaluates the possibility of automating the extraction of hidden preconditions by analyzing the CIL code.
- Section 10 concludes with an assessment of the lessons learned.

The focus of this work is on *design and programming methodology*, in particular design methodology for the construction of libraries of reusable components; we are looking for techniques that help component *producers* turn out better components, help component *consumers* learn to use the components, and reduce the potential for errors arising from incorrect, incomplete or misunderstood specifications. We were surprised, when presenting early versions, that some listeners were mostly interested in possibilities of *extracting* the contracts automatically from non-contracted components. Although section 9 indeed describes possibilities in this direction, building on earlier work on contract extraction, we must warn the reader not to expect any miracles; no computer tool can divine the intent behind a programmer's work without some help from the programmer. The potential for automatic contract inference should not obscure the concrete and immediate benefits that good design methodology can bring to both the producers and consumers of reusable software.

2 The Context

2.1 A Distinctive Design Style

Applying to reusable libraries the ideas of Design by Contract [21] [23] [25] [31] means equipping each library class with precise specifications, or "contracts", governing its interaction with the library's clients. Contracts include the class invariant, stating general consistency conditions to be maintained by every exported routine of the class, and, for every routine, preconditions stating the clients' obligations, and postconditions stating guarantees to the clients.

Systematic application of these principles leads to a distinctive design style, immediately visible in Eiffel frameworks such as EiffelBase [10] [22] covering fundamental data structures and algorithms, EiffelVision for portable graphics, and others. These Eiffel libraries have been in wide use for many years and seemingly appreciated by their users as easy to learn, convenient to use, and beneficial to the reliability of applications built with them. A recent report by the Software Engineering Institute [37] confirms that for components in general — not just classes — the use of contracts appears to be a key condition of any effort to improve "composability" and scale up the application of component-based technology.

Design by Contract as it has been applied to libraries so far, mostly in Eiffel, is not an a posteriori addition to the design of a library; it is an integral part of the design process. The resulting contract-rich library APIs are markedly different from more traditional, contract-less designs. One might argue that criticism of current libraries [38] becomes partly unjustified when libraries are built according to this style. The difference is clear, for example, in a comparison of two libraries that cover some of the same ground: EiffelBase [10] [22], which is based on Design by Contract, and the .NET Collections library [27], which is not. Most non-Eiffel libraries, such as the .NET framework's libraries, have indeed been built without explicit consideration to the notion of contract. Three possible explanations come to mind:

- The library authors do not know about Design by Contract.
- They know about the concepts, but don't find them particularly useful.
- They know about the concepts and find them interesting but too cumbersome to apply without built-in Eiffel-style support in the method, language and supporting tools.

Regardless of the reason, the difference in styles is so stark that we must ask what happened, in these contract-less libraries, to the properties that the Eiffel designer would have expressed in preconditions, postconditions and class invariants. It's this question that leads to the Closet Contract Conjecture: are the contracts of Eiffel libraries a figment of the Eiffel programmer's obsession with this mechanism? Or are they present anyway, hidden, in non-Eiffel libraries as well?

The only way to find out is to search contract-less libraries for closet contracts. In performing this search we have been rummaging through interface specifications, source code when available, documentation, even — since any detective knows not to overlook the household's final output — generated code, which in .NET and Java still retains significant high-level information.

2.2 .NET Libraries and the Contract Wizard

A property of the .NET libraries that makes them particularly interesting for such a study is the flexibility of the .NET component model, which has enabled the development of a "Contract Wizard" [1], a tool that enables a user to examine a compiled module ("assembly" in .NET), typically coming from a contract-less language such as C#, Visual Basic, C++, Cobol etc., and interactively add contracts to its classes and routines, producing a proxy assembly that is contracted as if it had been written in Eiffel, but calls the original.

The Contract Wizard relies on the reflection capabilities provided in .NET by the *metadata* that every assembly includes, providing interface information such as the signature of each routine, retained from the source code in the compiling process.

By nature, however, the Contract Wizard is only interesting if the Closet Contract Conjecture holds. This observation provides one of the incentives for the present study: as we consider further developments of the Contract Wizard, we must first gather empirical evidence confirming or denying its usefulness. If we found that .NET and other non-contracted libraries do very well without contracts, thank you very much, and that there are no useful closet contracts to be added, it would be a waste of time to continue working on the Contract Wizard.

2.3 Method of Work

Our library analyses have so far not relied on any automatic tools. Because we are looking for something that officially isn't there, we have to exercise our own interpretation to claim and authenticate our finds. It's incumbent on us to state why we think a particular class characteristic, such as an exception, is representative of an underlying contract.

Having to rely on a manual extraction process puts a natural limit on future extensions of this article's analysis to other libraries. Beyond facilitating the analysis, automated extraction tools could help users of the Contract Wizard by suggesting possible contract additions. The results of this article indeed suggest certain patterns, in code or documentation, that point to possible contracts, as certain geological patterns point to possible oil deposits. However, the final process of contract elicitation, starting from non-contracted libraries, requires subjective decisions.

3 Building Libraries with Design by Contract

The ideas of Design by Contract are inspired by commercial relationships and business contracts, which formally express the rights and obligations binding a client and a supplier. Likewise, software contracts are a way to specify the roles and constraints applying to a class as a whole (class invariants) or to the routines of the class (preconditions and postconditions).

3.1 Why Use Contracts?

Many programmers who have heard of contracts think they are 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. Such a use — similar to the "assert" of C — is only a small part of the application of contracts, and wouldn't by itself justify special language constructs. Contracts address a wider range of issues in the software process, 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 once they are there. Designing with contracts encourages the designer to think about the abstract properties of each software element, and build the observance of these properties into the software.
– *Documentation*: From contracted software, automatic tools can extract documentation that is both abstract and precise. Because the information comes from the software text, this approach saves the effort of writing documentation as a separate product, and lowers the risk of divergence between software and documentation. It underlies the basic form of Eiffel documentation for Eiffel software: the *contract form*, produced by tools of the Eiffel environment and retaining interface information only.
– *Debugging* and *testing*: Run-time monitoring of contracts permits a coherent, focused form of quality assurance based on verifying that the run-time state of the software satisfies the properties expected by the designers.
– *Inheritance control*: Design by Contract principles provide a coherent approach to inheritance, limiting the extent to which new routine definitions may affect the original semantics (preconditions may only be weakened, postconditions strengthened).
– *Management*: Contracts allow project managers and decision makers to understand the global purpose of a program without going into the depth of the code.

The principles are particularly relevant to library design. Eiffel libraries are thoroughly equipped with contracts stating their abstract properties, as relevant to clients.

3.2 Kinds of Contract Elements

Contracts express the semantic specifications of classes and routines. They are made of *assertions*: boolean expressions stating individual semantic properties, such as the property, in a class representing lists stored in a container of bounded capacity, that the number `count` of elements in a list must not exceed the maximum permitted, `capacity`.

Uses of contracts include:

- *Preconditions*: Requirements under which a routine will function properly. A precondition is binding on clients (callers); the supplier (the routine) can turn it to its advantage to simplify its algorithm by assuming the precondition.
- *Postconditions*: Properties guaranteed by the supplier to the client on routine exit.
- *Class invariants*: Semantic constraints characterizing the integrity of instances of a class; they must be ensured by each constructor (creation procedure) and maintained by every exported routine.
- *Check instructions*: "Assert"-like construct, often used on the client side, before a call, 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 properties of library interfaces and will not be considered further here.

Although preconditions and postconditions are the best known forms of library contracts, class invariants are particularly important in an object-oriented context since they express fundamental properties of the abstract data type (ADT) underlying a class, and the correctness of the ADT implementation chosen for the class (representation invariant [9]). We must make sure that our contract elicitation process doesn't overlook them.

3.3 Contracts in Libraries

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

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

This specification tells us that the function takes a `REAL` argument and returns a `REAL` result. That is already a form of contract, specifying the type signature of the function. We can call it a *signature contract*. (Regrettably, some of the Java and .NET reference documentation uses the term "contract", without qualification, for such signature contracts, creating confusion with the well established use of the term as used in the rest of this article.) A more complete contract — *semantic contract* if we need to distinguish it from mere signature contracts — should also specify properties of the argument and result that can't just be captured by type information, but is just as important to the library client. The most obvious example is what happens for a negative argument, with at least four possible answers:

- The function might silently return a default value, such as zero. (Not very good!)
- It might return a default value, and set a special flag that it is the caller's responsibility to examine after a call.
- It might trigger an exception, which it is the caller's responsibility to handle (otherwise execution will probably terminate abnormally).
- It might produce aberrant behavior, such as entering an infinite loop or crashing the execution. (Not good in the absence of contracts.)

A fifth would be to return a *COMPLEX* result, but that is not permitted by statically typed languages if the specification, as above, declares the type of the result as *REAL*.

A contract — here a precondition and a postcondition — will express which of these specifications 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 here ...

  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 there for documentation purposes and will also appear in error messages if the contracts are checked at run time during debugging and testing.

The *contract form* of the enclosing class, as produced by the environment tools, will show the above without the **do**... part and the **is** keyword (but with the header comment). This is the basic documentation that any application programmer wishing to use an Eiffel class will receive. That documentation is, essentially, the set of contracts associated with the class.

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, but the contract is inherent to the routine, regardless of its language of implementation. If a library is to provide a usable square root routine it must be based on such a contract. Unless you know under what conditions a square root function will operate and what properties you may expect of its result, you couldn't use it properly.

The general questions are:

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

The rest of this study provides material for answering these questions.

4 Why .NET Libraries?

.NET libraries suggested themselves for our study not only because they are one of the most recent and widely publicized collections of general-purpose reusable components, but also because the innovative .NET concept of *metadata* equips them with specification information that appears directly useful to contract elicitation.

4.1 The Scope of .NET

The .NET libraries are part of the .NET framework [24], a major Microsoft endeavor. Overall, .NET addresses the needs of companies and the general public through advances in Web services infrastructure, new tools for business-to-business and business-to-consumer interactions (Universal Description, Discovery and Integration, MyServices, Biztalk), advanced Web mechanisms (through the ASP.NET framework), new security mechanisms and other innovations.

In support of these goals, .NET includes new techniques and products of direct interest to developers. The most significant advance here is extensive support for multi-language programming with full interoperability between languages. Beyond Microsoft's own languages, C# and Visual Basic .NET, implementations exist for third-party languages, in particular Eiffel. (The Eiffel implementation on .NET includes the full language as on other platforms; in particular it supports Design by Contract, genericity and multiple inheritance without restrictions [2] [36].) All such language implementations benefit from common mechanisms including a versioning scheme, an extensive security framework, a component model considerably simpler to use than Microsoft's previous COM technology, facilities for memory management, debugging and exception handling, a multi-language development environment (Visual Studio .NET), all supported by a Common Language Runtime. They also benefit from a set of libraries covering many areas of applications, which the interoperability infrastructure makes available to programs written in all languages supported on .NET.

4.2 The Role of Metadata

The basic compilation unit in .NET, covering for example a library or a section of a library, is the assembly. The key to the framework's support for component-based development is the presence, in every assembly, of documentary information known as *metadata*, making the assembly self-describing in accordance with the *self-documentation principle* [23].

The metadata of an assembly — accessible to programs through the Reflection library, to users through various tools, and to the world at large in XML — provides information on the assembly's contents, including:

– A "manifest" describing the assembly name, version, culture and public key (if the assembly is signed).
– The list of dependencies on other .NET assemblies.
– For each class, the list of its parents (interfaces, and at most one class) and of its features (routines, attributes, properties and events).
– For each class member, the signature (including arguments and return type).

In addition to these predefined categories, developers can define, assuming proper source language support, their own specific kinds of metadata in the form of *custom attributes*.

Metadata of both kinds — predefined and custom — opens attractive new possibilities. The Contract Wizard is one of them: by relying on the metadata, it enables users to examine a class and its features interactively, and add any appropriate contracts — all without having access to the source code.

5 Analysis of a Collection Class

Our first search for closet contract will target the class `ArrayList` [28], part of the core .NET library (*mscorlib.dll*). This choice of class is almost arbitrary; in particular it was not based on any a priori guess that the class would suggest more (or fewer) contracts than any other. Rather, the informal criteria were that the class, describing lists implemented through arrays:

– Is of obvious practical use.
– Seems typical, in its style, of the Collections library.
– Has a direct counterpart, *`ARRAYED_LIST`*, in the EiffelBase library, opening the possibility of comparisons once we've completed the contract elicitation process.

5.1 Implicit Class Invariants

Documentation comments first reveal properties of `ArrayList` that fall into the category of class invariants.

We find our first leads in the *specification of class constructors*, which states that

"*The default initial capacity for an ArrayList is 16*"

This comment implies that the capacity of the created object is greater than zero. Taking up this lead, we notice that all three constructors of `ArrayList` set the initial list's capacity to a positive value. This suggests an invariant, since it is part of the Design by Contract rules that an invariant property must be guaranteed by all the creation procedures of a class.

To test our intuition, we examine the other key property of an invariant: that it must be preserved by every exported routine of the class. Examining all such routines confirms this and suggests that we indeed have the germ of an invariant, which in Eiffel would be expressed by the clause

```
invariant

  positive_capacity: capacity >= 0
```

Continuing our exploration of the documentation, we note that two of the three constructors of `ArrayList`

"*initialize a new instance of the* `ArrayList` *class that is empty*".

The `count` of elements of an array list created in such a way must then be zero. The third constructor, which takes a collection `c` as parameter

"*initializes a new instance of the ArrayList class that contains elements copied from the specified collection*"

So the number of elements of the new object equals the number of elements in the collection received as parameter, expressed by the assertion `count = c.count` (which in Eiffel would normally appear in a postcondition). Can then `c.count` be negative? Most likely not. Checking the documentation further reveals that the argument `c` passed to the constructor may denote any non-void collection, represented through one of the many classes inheriting from the `ICollection` interface [29]: arrayed list, sorted list, queue etc. Without performing an exhaustive examination, we note a hint in `ArrayList` itself, in the specification of routine `Remove`:

"*The average execution time is proportional to Count. That is, this method is an O(n) operation, where n is Count*"

which implies that `count` must always be non-negative. This evidence is enough to let us add a clause to the above invariant:

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

These first two properties are simple but already useful. For our next insights we examine the *specification of class members*. Documentation on the `Count` property reveals interesting information:

"`Count` *is always less than or equal to* `Capacity`".

The self-assurance of this statement indicates that this property of the class always holds, suggesting that it is a class invariant. Hence a third invariant property for class `ArrayList` yielding the accumulated clause

```
invariant

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

5.2 Implicit Routine Preconditions

Aside from implicit class invariants, the documentation also suggests preconditions. To get our clues we may look at documented exception cases.

The specification of the routine `Add` of class `ArrayList` states that `Add` throws 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 (does not have a fixed size) before actually adding elements to the list.

Such a requirement for having the method do what it is expected to is the definition of a routine precondition in terms of Design by Contract. An Eiffel specification of `Add` would then include the following two preconditions:

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

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 to be found in the reference documentation of the .NET Framework — suggests a scheme for extracting preconditions, applicable systematically, with some possibility of tool support:

- Read the exception condition; e.g. the array list is read-only.
- Take the opposite; for `ArrayList` the condition would be

  ```
  not is_read_only
  ```

- Infer the underlying routine precondition; here:

  ```
  writable: not is_read_only
  ```

5.3 Implicit Routine Postconditions

Does the .NET documentation also reveal closet postconditions? For an answer we consider the example of the query `IndexOf`. More precisely, since it is an overloaded method, we choose 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 may 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 list indexes 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.

Such a property is a guarantee on routine exit, a condition incumbent on the supplier on completion of the task — the definition of a postcondition. In Eiffel we would add the corresponding clause to the routine:

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

5.4 Contracts in Interfaces

Class `ArrayList` implements three interfaces (completely abstract specification modules) of the .NET Collections library: `IList`, `ICollection`, and `IEnumerable`. It is interesting to subject such interfaces to the same analysis as we have applied to the class.

This analysis (see sections 6 and 7 for general statistics about the contract rates of .NET collection classes and interfaces) indicates that `IList`, `ICollection`, `IEnumerable`, and `IEnumerator` (of which `IEnumerable` is a client), do have routine preconditions and postconditions similar to those of `ArrayList`.

We have not, however, found class invariants in these interfaces. This is probably because of the more limited scope of interfaces in .NET (coming from Java) as compared to "deferred classes", their closest counterpart in the object-oriented model embodied by Eiffel. Deferred classes may have a mix of abstract and concrete features; in particular, they may include attributes. Interfaces, for their part, are purely abstract and may not contain attributes. The Eiffel policy provides a continuous spectrum from totally deferred classes, the equivalent of .NET and Java interfaces, to fully implemented classes, supporting the aims of object-oriented development with 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 [22] often express consistency properties binding various attributes together.

One can imagine, however, finding properties that hold for all the classes implementing the interface and that would be relevant candidates for "interface invariants". But our non-exhaustive analysis of the .NET Collections library did not reveal such a case.

6 Adding Contracts a Posteriori

The discovery of closet contracts in the .NET arrayed list class suggests that we should build a "contracted variant" of this class, `ARRAY_LIST`, that has the same interface as the original `ArrayList` plus the elicited contracts. We now present a sketch of this class and compare it with its EiffelBase counterpart: `ARRAYED_LIST`.

Rather than modifying the original class we may produce the contracted variant — here in Eiffel — as a new class whose routines call those of the original. This is the only solution anyway when one doesn't have access to the source code. The Contract Wizard is intended to support such a process, although for this discussion we have produced the result manually.

6.1 A Contracted Form of the .NET Arrayed List Class

The original class, `ArrayList`, has 57 features (*members* in the .NET terminology). In presenting the contracted version we limit ourselves to 12 features, discussed in the preceding analysis of the class.

The notation

```
require -- from MY_CLASS
```

or

```
ensure -- from MY_CLASS
```

shows that the assertion clauses that follow (respectively preconditions or postconditions) are inherited from the parent class `MY_CLASS`. This is the convention applied by EiffelStudio documentation tools when displaying the assertions of a class, for the parts inherited from ancestors. The difference is that in Eiffel assertions clauses are automatically inherited from parents; in .NET there is no such convention, so the .NET documentation has to repeat the same comments and exception conditions for an ancestor class or interface and all its descendants.

The **`feature`** keyword introduces a "feature clause", which groups a set of features (*members*) that have a common purpose. For example, the feature clause **`feature`** `-- Initialization` lists all the creation procedures of class `ARRAY_LIST`: `make`, `make_from_capacity` and `make_from_collection`.

```
indexing
  description: "[
                Implementation of a list using an
                array, whose size is dynamically
                increased as required.
                ]"
class interface ARRAY_LIST
create
    -- Note for non-Eiffelists: This is the list of
    -- creation procedures (constructors) for the
    -- class; the procedures’ definitions appear below.
  make,
  make_from_capacity,
  make_from_collection
feature -- Initialization
  make
      -- Create empty list with capacity
      -- Default_capacity.
    ensure
      empty: count = 0
      default_capacity_set: capacity = Default_capacity
      writable: not is_read_only
      extendible: not is_fixed_size
```

```
make_from_capacity (a_capacity: INTEGER)
    -- Create empty list with capacity a_capacity.
  require
    positive_capacity: a_capacity >= 0
  ensure
    empty: count = 0
    positive_capacity_implies_capacity_set:
      a_capacity > 0 implies capacity = a_capacity
    capacity_is_zero_implies_default_capacity_set:
      a_capacity =0 implies capacity =Default_capacity
    writable: not is_read_only
    extendible: not is_fixed_size

make_from_collection (c: ICOLLECTION)
    -- Create list containing elements copied from c
    -- and the corresponding capacity.
  require
    collection_not_void: c /= Void
  ensure
    capacity_set: capacity = c.count
    count_set: count = c.count
    writable: not is_read_only
    extendible: not is_fixed_size
```

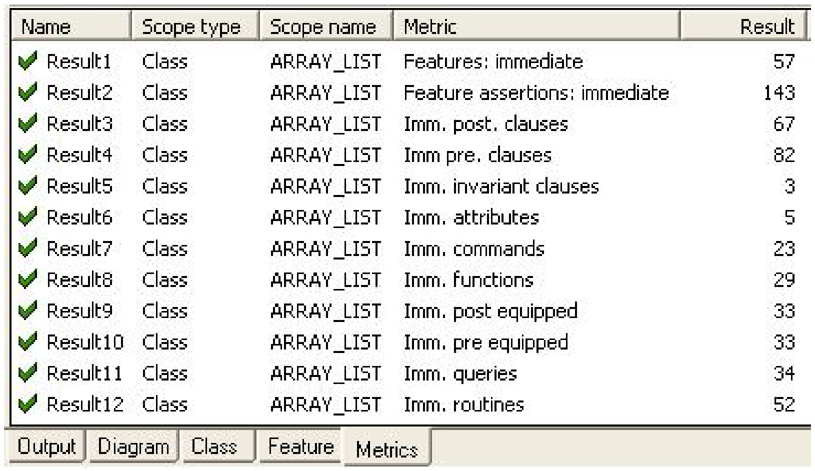
```
feature -- Access
  capacity: INTEGER
      -- Number of elements the list can store
  count: INTEGER
      -- Number of elements in the list
  Default_capacity: INTEGER is 16
      -- Default list capacity

  index_of (value: ANY): INTEGER
      -- Zero-based index of the first occurrence of
      -- value
    ensure -- from ILIST
      not_found_implies_minus_one:
        not contains (value) implies Result = - 1
      found_implies_valid_index:
        contains (value) implies
          Result >= 0 and Result < count
      found_implies_correct_index:
        contains (value) implies item (Result) = value

  item (index: INTEGER): ANY
      -- Entry at index
    require -- from ILIST
      valid_index: index >= 0 and index < count
```

```
feature -- Status report

  contains (an_item: ANY): BOOLEAN
      -- Does list contain an_item?

  is_fixed_size: BOOLEAN
      -- Has list a fixed size?

  is_read_only: BOOLEAN
      -- Is list read-only?


feature -- Status setting

  set_capacity (value: like capacity)

      -- Set list capacity to value.

    require

      valid_capacity: value >= count

    ensure

      capacity_set: value > 0 implies capacity = value

      default_capacity_set:

        value = 0 implies capacity = Default_capacity


feature -- Element change

  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

        valid_index_returned: Result = count - 1

      ensure then

        capacity_doubled: (old count = old capacity)

          implies (capacity = 2 * (old capacity))

  invariant

    positive_capacity: capacity >= 0

    positive_count: count >= 0

    valid_count: count <= capacity

  end
```

6.2 Metrics

Fig. 1 shows measurements of properties of the contracted class *ARRAY_LIST*, produced with by the Metrics Tool of EiffelStudio. The measurements apply to the full class, with all 57 features, not to the abbreviated form shown above. A feature is "immediate" if it is new in the class, as opposed to a feature inherited from a parent (and possibly redefined in the class).



Name	Scope type	Scope name	Metric	Result
✓ Result1	Class	ARRAY_LIST	Features: immediate	57
✓ Result2	Class	ARRAY_LIST	Feature assertions: immediate	143
✓ Result3	Class	ARRAY_LIST	Imm. post. clauses	67
✓ Result4	Class	ARRAY_LIST	Imm pre. clauses	82
✓ Result5	Class	ARRAY_LIST	Imm. invariant clauses	3
✓ Result6	Class	ARRAY_LIST	Imm. attributes	5
✓ Result7	Class	ARRAY_LIST	Imm. commands	23
✓ Result8	Class	ARRAY_LIST	Imm. functions	29
✓ Result9	Class	ARRAY_LIST	Imm. post equipped	33
✓ Result10	Class	ARRAY_LIST	Imm. pre equipped	33
✓ Result11	Class	ARRAY_LIST	Imm. queries	34
✓ Result12	Class	ARRAY_LIST	Imm. routines	52

Fig. 1. Metrics about the contracted arrayed list class

The Eiffel metric tool's output uses the following terminology:

– *Feature* is the general name for class members. Features include attributes ("*fields*") and routines ("*methods*"). A routine is a computation (algorithm) applicable to instances of the class; an attribute is stored in memory. If a routine returns a result, it is called a *procedure*; otherwise, it is a *function*.
– Eiffel also distinguishes between *commands* and *queries*: A command returns no result; a query returns a result. If a query is computed at run time, it is a function; if it is stored in memory, it is an attribute.

We note the following conclusions from these measurements:

– *62% of the routines now have a contract* (a precondition or a postcondition, usually both): 33 out of 52.
– The 33 routines with preconditions tend to have *more than one precondition clause: 2.5 on the average* (82 total).
– The 33 routines with postconditions tend to have *more than one postcondition clause: 2 on average* (67 total).

7 Extending to Other Classes and Interfaces

Equipped with our first results on `ArrayList` and its contracted Eiffel counterpart *`ARRAY_LIST`*, we now perform similar transformations and measurements on a few other classes and interfaces, to probe how uniform the results appear to be across the .NET Collections library. Since the assumptions and techniques are the same, we won't repeat the details but go directly to results and interpretations.

7.1 Interfaces

First, consider Eiffel deferred classes obtained by contracting the .NET interfaces from which `ArrayList` inherits:

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

Table 1 shows some resulting measurements.

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
Precondition rate	64 %	25 %	0 %	50 %
Postcondition rate	64 %	25 %	100 %	50 %
Class invariants	0	0	0	0

The statistics highlight three trends:

– Absence of class invariant in these .NET interfaces, as already noted.
– Presence of routine contracts: both preconditions and postconditions. The figures about *`IENUMERABLE`* and *`ICOLLECTION`* involve too few routines to bring valuable information — only one for class *`IENUMERABLE`* and four for *`ICOLLECTION`*. The figures about *`ILIST`* and *`IENUMERATOR`* are more significant in that respect — class *`ILIST`* has eleven routines and *`IENUMERATOR`* has six. Both classes (*`ILIST`* and *`IENUMERATOR`*) have at least one half of their routines with contracts.
– Presence of multiple routine contracts: most routines have several preconditions and postconditions.

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

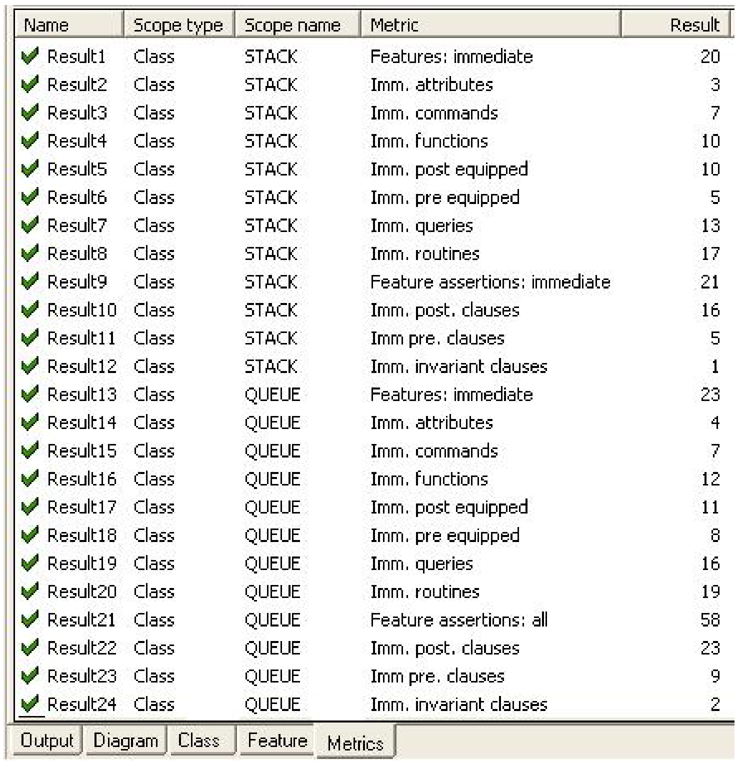
7.2 Other Classes: `Stack` and `Queue`

To test the generality of our first results on `ArrayList`, we consider two other classes of the Collections library. We choose `Stack` and `Queue` because they:

– Are concrete collection classes.
– Have no relation to `ArrayList`, except that all three implement the .NET interfaces `ICollection` and `IEnumerable`.
– Have a direct counterpart in the EiffelBase library.

Three classes is still only a small sample of the library. Any absolute conclusion would require exhaustive analysis, and hence a larger effort than the present study since the analysis is manual. So we have to be careful with any generalization of the results. We may note, however, that none of the three choices has been influenced by any a priori information or guess about the classes' likelihood of including contracts.

The same approach was applied to these classes as to `ArrayList` and its parents. Fig. 2 shows some of the resulting measurements for classes `Stack` and `Queue`.



Name	Scope type	Scope name	Metric	Result
✓ Result1	Class	STACK	Features: immediate	20
✓ Result2	Class	STACK	Imm. attributes	3
✓ Result3	Class	STACK	Imm. commands	7
✓ Result4	Class	STACK	Imm. functions	10
✓ Result5	Class	STACK	Imm. post equipped	10
✓ Result6	Class	STACK	Imm. pre equipped	5
✓ Result7	Class	STACK	Imm. queries	13
✓ Result8	Class	STACK	Imm. routines	17
✓ Result9	Class	STACK	Feature assertions: immediate	21
✓ Result10	Class	STACK	Imm. post. clauses	16
✓ Result11	Class	STACK	Imm pre. clauses	5
✓ Result12	Class	STACK	Imm. invariant clauses	1
✓ Result13	Class	QUEUE	Features: immediate	23
✓ Result14	Class	QUEUE	Imm. attributes	4
✓ Result15	Class	QUEUE	Imm. commands	7
✓ Result16	Class	QUEUE	Imm. functions	12
✓ Result17	Class	QUEUE	Imm. post equipped	11
✓ Result18	Class	QUEUE	Imm. pre equipped	8
✓ Result19	Class	QUEUE	Imm. queries	16
✓ Result20	Class	QUEUE	Imm. routines	19
✓ Result21	Class	QUEUE	Feature assertions: all	58
✓ Result22	Class	QUEUE	Imm. post. clauses	23
✓ Result23	Class	QUEUE	Imm pre. clauses	9
✓ Result24	Class	QUEUE	Imm. invariant clauses	2

Fig. 2. Metrics about the contracted stack and queue classes

The figures confirm the trends previously identified:

- Preconditions and postconditions are present. Class *STACK* has a 29% precondition rate (17 routines, of which 5 have preconditions) and a 59% postcondition rate (10 postcondition-equipped out of 17); class *QUEUE* has 42% and 58%.
- Preconditions and postconditions usually include several assertions. For example, *STACK* has 16 postcondition assertions for 10 contract-equipped routines.
- 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*.

This case-by-case analysis of 3 concrete classes and 4 interfaces of the .NET Collections library (out of 13 concrete classes and 8 interfaces) supports the second answer of the “Closet Contract Conjecture” — that contracts are inherent. We will now explore the benefits and limitations of such an a posteriori addition of contracts.

8 Effect on Library Users

To appreciate the value of the results of the preceding analysis, we should assess their effect on the only constituency that matters in the end: library users — application developers who take advantage of library classes to build their own systems. This issue is at the core of the Closet Contract Conjecture, since it determines whether we are doing any good at all by uncovering implicit contracts in contract-less libraries. By producing new versions of the library that make the contracts explicit, are we actually helping the users?

To answer this question, we may examine the effect of the different styles on the library user (in terms of ease of learning and ease of use) and on the likely quality of the applications they develop. We take arrayed lists as an example and consider three variants:

– The original, non-contracted class `ArrayList` from the .NET Collections library.
– The contracted version *`ARRAY_LIST`* discussed above.
– Finally, the corresponding class in the EiffelBase library, called *`ARRAYED_LIST`*, which was built with Design by Contract right from the start, rather than contracted a posteriori, and uses some other design ideas as well.

8.1 Dealing with Abnormal Cases in a Contract-less Style

The chapter in the .NET documentation devoted to class `ArrayList` provides a typical example of dealing with arrayed lists in that framework:

```
using System;

using System.Collections;

public class SamplesArrayList {

  public static void Main() {

    // Creates and initializes a new ArrayList.

    ArrayList myAL = new ArrayList();

    myAL.Add("Hello");

    myAL.Add("World");

    myAL.Add("!");

    // Displays the properties and values of the

    // ArrayList.

    Console.WriteLine("myAL");
    Console.WriteLine("\tCount:    {0}",myAL.Count);
    Console.WriteLine("\tCapacity: {0}",myAL.Capacity);
    Console.Write ("\tValues:");
    PrintValues (myAL);
  }
  public static void PrintValues (IEnumerable myList) {
    System.Collections.IEnumerator myEnumerator =
      myList.GetEnumerator();
    while (myEnumerator.MoveNext())
      Console.Write("\t{0}", myEnumerator.Current);
    Console.WriteLine();
  }
}
```

Running this C# program produces the following output:

```
myAL
  Count:    3
  Capacity: 16
  Values:   Hello    World    !
```

One striking point of this example is the absence of any exception handling — not even one **`if`** instruction in the class text — although our analysis of class `ArrayList` (see section 5) has revealed a non-trivial number of implicit contracts.

For example, we have seen that the .NET method `Add` can only work properly if the targeted arrayed list is writable and extendible. But there is no such check in the class text above. This is likely to be on purpose since the property always holds at this point of the method execution: the .NET constructor ensures that the created list is not read-only and does not have a fixed size (see the contracted version of class `ArrayList` introduced in section 6), which allows calling the method `Add` on it.

```
ArrayList myAL = new ArrayList();

/* Implicit check:

    (!myAL.IsFixedSize) && (!myAL.IsReadOnly)

*/

myAL.Add ("Hello");

myAL.Add ("World");

myAL.Add ("!");
```

If harmless in this simple example, such code may become dangerous if part of a reusable component. As a matter of fact, a novice programmer may overlook such a subtlety and reuse this code to create and add elements to a fixed-size arrayed list, which would cause the program execution to terminate on an unhandled exception of type `NotSupportedException`.

This becomes even clearer if we encapsulate the calls to `Add` in a separate method `FillArrayList` that would look like the following:

```
public void FillArrayList( ArrayList AL ){

  AL.Add ("Hello");

  AL.Add ("World");

  AL.Add ("!");

}
```

and use `FillArrayList` in the Main routine:

```
public static void Main()  {

  ArrayList myAL = new ArrayList();

  /* Implicit check:

      (!myAL.IsFixedSize) && (!myAL.IsReadOnly)

  */

  FillArrayList (myAL);

}
```

The previous program would work; the following one would not:

```
public static void Main()  {
  ArrayList myAL = new ArrayList();
  ArrayList.FixedSize (myAL);
  // The following call would throw an exception
  // because myAL is now a fixed-size arrayed list,
  // to which no element can be added.
  FillArrayList (myAL);
}
```

Having Design by Contract support would be the right solution here (as discussed in the next sections). But because the .NET Common Language Runtime does not have native knowledge of contracts, .NET users have to rely on other techniques:

– Using a *"defensive" style of programming*: checking explicitly for the routine requirements even if it can be inferred directly from the previous method statements (relying on the motto: "better check too much than too less"), hence adding redundant checking:

  ```
  ArrayList myAL = new ArrayList();
  if ((!myAL.IsFixedSize) && (!myAL.IsReadOnly))
    FillArrayList (myAL);
  ```

  with:

  ```
  public void FillArrayList (ArrayList AL){
    if ((!myAL.IsFixedSize) && (!myAL.IsReadOnly)){
      AL.Add ("Hello");
      AL.Add ("World");
      AL.Add ("!");
    }
  }
  ```

  This style, however, leads to needless complexity by producing duplicate error-checking code. The Design by Contract method goes in the opposite direction by avoiding redundancy and needless (*Non-Redundancy principle*, [20] p 343).

– Relying on the *exception handling* mechanism of the .NET Common Language Runtime (typically, by using **try**…**catch**...**finally**… clauses):

```
public static void Main() {

  try {

    // Creates and initializes a new ArrayList.

    ArrayList myAL = new ArrayList();

    FillArrayList (myAL);

    // Prints list values.

  }

  catch (NotSupportedException e) {

    Console.WriteLine (e.Message);

  }

}
```

with:

```
public void FillArrayList (ArrayList AL)

throws NotSupportedException {

  AL.Add ("Hello");

  AL.Add ("World");

  AL.Add ("!");

}
```

– Adding *comments* in the code to make implicit checks explicit and avoid misleading the library users:

```
public static void Main() {

  ArrayList myAL = new ArrayList();

  /* Implicit check:

      (!myAL.IsFixedSize) && (!myAL.IsReadOnly)

  */
```

```
    FillArrayList (myAL);
  }
```

with:

```
  /* This method can only be called if AL does not
   * have a fixed size and is not read-only.
   */
  public void FillArrayList (ArrayList AL) {
    AL.Add ("Hello");
    AL.Add ("World");
    AL.Add ("!");
  }
```

Such an approach is efficient in the sense that there is no redundant check, thus no performance penalty, which gets closer to the ideas of Design by Contract, but it is not enforced at run time since it just relies on comments. This suggests the next approach, a posteriori contracting of classes.

8.2 Dealing with Abnormal Cases in a Contract-rich Style

A posteriori addition of contracts to a .NET component is likely to simplify the task of clients: rather than testing for a routine's successful completion, they can just rely on the contracts, yielding to a lighter programming style (no redundant checking):

```
indexing
  description: "[
                Typical use of contracted class
                ARRAY_LIST
                ]"
class ARRAY_LIST_SAMPLE
create
  make
feature -- Initialization
```

```
  make is

      -- Create an arrayed list, fill it with
      -- Hello World!, and print its content.

    local

      my_list: ARRAY_LIST

    do

      create my_list.make

      fill_array_list (my_list)

      print_values (my_list)

    end

feature -- Element change

  fill_array_list (an_array_list: ARRAY_LIST) is

      -- Fill an_array_list with Hello World!.

    require

      an_array_list_not_void: an_array_list /= Void

      is_extendible: not an_array_list.is_fixed_size

      is_writable: not an_array_list.is_read_only

    local

      index: INTEGER

    do

      index := an_array_list.add ("Hello ")

      index := an_array_list.add ("World”)

      index := an_array_list.add (“!")

    ensure

      array_list_filled: an_array_list.count = 3

    end
```

```
feature -- Output
  print_values (an_array_list: ARRAY_LIST) is
      -- Print content of an_array_list.
    require
      an_array_list_not_void: an_array_list /= Void
    local
      my_enumerator: IENUMERATOR
    do
      from
        my_enumerator := an_array_list.enumerator
      until
        not my_enumerator.move_next
      loop
        print (my_enumerator.current_element)
      end
    end
  end
```

Since we know from the postconditions `is_extendible` and `is_writable` of creation procedure *make* of *ARRAY_LIST* that the preconditions of *fill_array_list* will be satisfied at this point of the routine execution, we do not need to add tests before calling the procedure.

For readability or to facilitate debugging — when executing the software with assertion monitoring on — we might want to use an additional **check** instruction:

```
create my_list.make
check
  non_void: my_list /= Void
  is_extendible: not my_list.is_fixed_size
  is_writable: not my_list.is_read_only
end
fill_array_list (my_list)
print_values (my_list)
```

although this is not required.

If the creation routine `make` of class *ARRAY_LIST* had no such postconditions as `is_extendible` and `is_writable`, an explicit **if** control would have been needed in the client class *ARRAY_LIST_SAMPLE* to guarantee that the requirements of feature `fill_array_list` actually hold:

```
create my_list.make
if not my_list.is_fixed_size
  and not my_list.is_read_only then
    fill_array_list (my_list)
end
```

But this is different from the "defensive" programming style used in a contract-less environment, since the test only affects the client side, not both client and supplier; the latter simply has preconditions.

We call such a use of routine preconditions the *a priori scheme*: the client must act beforehand — before calling the routine — and ensure that the contracts are satisfied (either by testing them directly with an **if** control, or by relying on the postconditions of a previously called routine or on the class invariants). With this approach, any remaining run-time failure signals a design error.

Such a design may not always be applicable in practice for either of three reasons:

– *Performance*: Testing for a precondition before a routine call may be similar to the task of the routine itself, resulting in an unacceptable performance penalty.
– *Lack of expressiveness of the assertion languages*: The notation for assertions might not be powerful enough.
– *Dependency on external events*: It is impossible to test for requirements if a routine involves interaction with the outside world, for example with a human user: there is no other choice than attempting to execute it, hence no way to predict abnormal cases.

To address these limitations of the a priori scheme, it is possible to apply an *a posteriori scheme* — try the operation first and find out how it went — if a failed attempt has no irrecoverable consequences.

Performance overhead — the first case above — is not a problem when the test being repeated is checking that a number is positive or a reference is not void. But the inefficiency might be more significant. An example from numerical computation [23] is a matrix equation solver: an equation of the form $AX = B$, where A is a matrix, and X (the unknown) and B are vectors, has a unique solution of the form $X = A^{-1} B$ only if matrix A is not singular. (A matrix is singular if one of the rows is a linear combination of others.) Applying the a priori scheme would lead the client to write code looking like the following:

```
if a.is_singular then

  -- Report error.

else

  x := a.inverse (b)

end
```

using a function *inverse* with precondition non_singular_matrix:

```
inverse (b: VECTOR): VECTOR

    -- Solve equation of the form ax = b.

  require

    non_singular_matrix: not is_singular
```

This code does the job but is inefficient since determining whether a matrix is singular is essentially the same operation as solving the associated linear equation. Hence the idea of applying the a posteriori scheme; the client code would be of the form:

```
a.invert (b)

if a.inverted then

  x := a.inverse

else

  -- Process erroneous case.

end
```

Procedure `invert` replaces the previous function `inverse`. A call to this procedure (for which a more accurate name might be `attempt_to_invert`) sets a boolean attribute `inverted` to **`True`** or **`False`** to indicate whether inverting the matrix was possible, and if it was, makes the result available through attribute `inverse`. (A class invariant may state that `inverted = (inverse /= Void)`.)

This technique, which splits any function that may produce errors into a procedure that attempts to perform an operation and two attributes, one reporting whether the operation was successful and the other giving access to the result of the operation if any, is compliant with the *Command-Query Separation principle* ([23], p 751).

This example highlights one basic engineering principle for dealing with abnormal cases: whenever available, a method for preventing failures to occur is usually preferable to methods for recovering from failures.

The techniques seen so far do not, however, provide a solution in three cases:

– When abnormal events — such as a numerical failure or memory exhaustion — can cause the hardware or the operating system to interrupt the program execution abruptly (which is intolerable for systems with continuous availability requirements).
– When abnormal situations, although not detectable through preconditions, must be diagnosed at the earliest possible time to avoid disastrous consequences — such as destroying the integrity of a database or even endangering human lives, as in an airplane control system. (One must keep in mind that such situations can appear in a contract-rich environment as well, since the support for assertions may not be rich enough to express complex properties.)
– When there is a requirement for *software fault tolerance*, protecting against the most dramatic consequences of any remaining errors in the software.

8.3 "A Posteriori Contracting" vs. "Contracting from the Start"

We have seen that clients of a .NET library are likely to benefit from an a posteriori addition of contracts: instead of having to test whether a routine successfully went to completion (with the risk of forgetting to check and getting an exception at run time), they could just rely on the contracts.

What about contracting "from the start" now? Is the EiffelBase class *`ARRAYED_LIST`* more convenient to use than the contracted class *`ARRAY_LIST`*?

To help answer this question, let's consider a variant of the previous class *`ARRAY_LIST_SAMPLE`*, representing a typical client use of arrayed lists, this time using the EiffelBase *`ARRAYED_LIST`*:

```
indexing
  description: "Typical use of EiffelBase ARRAYED_LIST"

class ARRAYED_LIST_SAMPLE

create

  make

feature -- Initialization

  make is

      -- Create a list with two elements and print the
      -- list contents.

    local

      my_list: ARRAYED_LIST [STRING]

    do

      create my_list.make

      my_list.extend ("Hello ")

      my_list.extend ("World”)

      my_list.extend (“!")

      from

        my_list.start

      until

        my_list.after

      loop

        io.put_string (my_list.item)

        my_list.forth

      end

    end

end
```

This example highlights three characteristics of the EiffelBase `ARRAYED_LIST`:

– A clear separation between commands and queries: the routine `extend` returns no result (on the contrary to the .NET feature `Add`, which returns an integer, yielding a useless local variable `index` in the `ARRAY_LIST` code example).
– The usefulness of *genericity*: we know that `my_list.item` is of type `STRING`, thus we can use a more appropriate I/O feature to print it: `put_string`, rather than the general `print`.
– A user-friendly interface to traverse the list through features `start`, `after`, `item`, and `forth`, relying on an internal cursor stored in class `ARRAYED_LIST`.

Another interesting property to look at is the easiness of switching to other list implementations. As shown on Fig. 3, `ARRAYED_LIST` inherits from both `ARRAY` and `DYNAMIC_LIST`. It suffices to remove the relationship with class `ARRAY` to obtain a `LINKED_LIST`.

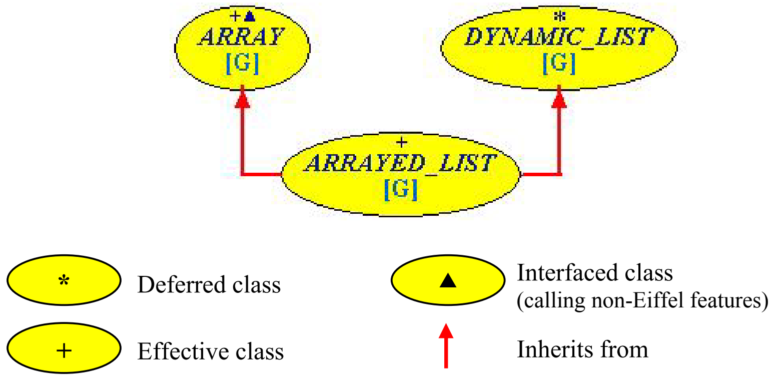


Fig. 3. Inheritance hierarchy of the EiffelBase arrayed list class

The a-posteriori-contracted class `ARRAY_LIST` (section 6) just mapped the original .NET hierarchy, which makes extensive use of interfaces to compensate the lack of multiple inheritance (Fig. 4).

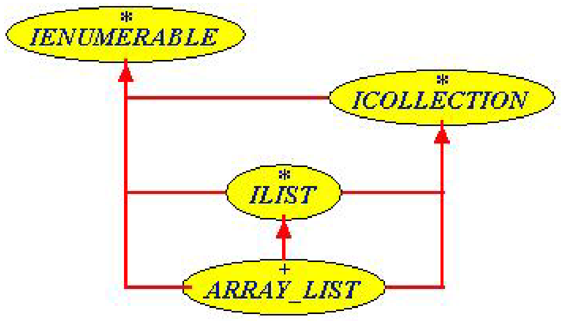


Fig. 4. Inheritance hierarchy of the .NET arrayed list class

Not surprisingly in light of how it was obtained, this class does not fully benefit from the power of Eiffel, in matter of design, reusability, and extendibility.

Another obvious difference between *ARRAY_LIST* (the contracted class) and *ARRAYED_LIST* (the EiffelBase class) is the use of genericity in the Eiffel version. Although this falls beyond the scope of the present discussion, we may note that the lack of genericity in the current .NET object model leads to code duplication as well as to run-time check (casts) that damage both the performance and the reliability of the software. Future versions of .NET are expected to provide genericity [17] [18].

The next difference is the use of enumerators in *ARRAY_LIST* whereas *ARRAYED_LIST* stores a cursor as an attribute of the class.

– In the first approach, enumerators become irrecoverably invalidated as soon as the corresponding collection changes (addition, modification, or deletion of list elements). The approach, on the other hand, allows multiple concurrent traversals of the same list through multiple cursors.

– The EiffelBase approach solves the problem of invalid cursors: addition or deletion of list elements change the cursor position, and queries *before* and *after* take care of the cursor position's validity. But the use of internal cursors requires care to avoid endless loops.

These differences of style should not obscure the fundamental difference between a design style that has contracts from the start and one that adds them to existing contract-less library components. The examples illustrate that the right time to put in the contracts is during design.

When that is not possible, for example with a library produced by someone who didn't trouble himself with contracts, it may still be worthwhile to add them a posteriori, as a way to understand the library better, improve its documentation, and make it safer to use.

9 Automatic Extraction of Closet Contracts

Our analysis of the .NET Collections library has shown interesting "patterns" about the nature and location of hidden contracts. In particular, routine preconditions tend to be buried under exception conditions. Our goal is to estimate the highest degree of automation we can achieve in extracting closet contracts from .NET libraries.

We first report on the technique of dynamic contract detection and then describe our approach of inferring preconditions from the CIL code [32] of .NET assemblies.

9.1 Dynamic Contract Inference

Dynamic contract inference, working from source code, seeks to deduce assertions from captured variable traces by executing the program with various inputs, relying on a set of possible assertions to deduce contracts from the execution output. The next step is to determine whether the detected assertions are meaningful and useful to the users, typically by computing a confidence probability.

Ernst's Daikon tool discovers class invariants, loop invariants and routine pre- and postconditions. Its first version [11] was limited to finding contracts over scalars and arrays; the next one (Daikon 2) [14] enables contract discovery over collections of data, and computes conditional assertions.

Daikon succeeds in finding the assertions of a formally-specified program, and can even find some more, revealing deficiencies in the formal specification. Daikon also succeeds in inferring contracts from a C program, which helps developers performing changes to the C program without introducing errors [12]. It appears that the large majority of reported invariants are correct, and that Daikon extracts more invariants from high-quality programs [11]. Daikon still needs improvement in terms of:

– *Performance*: "*Invariant detection time grows approximately quadratically with the number of variables over which invariants are checked*" [12]. Some experiments using incremental processing have shown promising results in improving Daikon's performance [14].
– *Relevance* of the reported invariants: Daikon still reports irrelevant — meaning useless, but not necessary incorrect — invariants. Polymorphism can help increase the number of desired invariants reported to the users [14].
– *Richness* of inferred invariants: Currently, most cover simple properties.

Ernst et al. suggest examining the techniques and algorithms used in the research fields of artificial intelligence [12] and information retrieval [13] to improve dynamic inference of invariants for applications in software evolution.

The Daikon detector is not the sole tool available to dynamically extract contracts. Some Java detectors also exist; some of them do not even require the program source code to infer contracts: they can operate directly on bytecode files (*.class).

9.2 Extracting Routine Preconditions from Exception Cases

"*Human analysis is sometimes more powerful than either, allowing deep and insightful reasoning that is beyond hope for automation*" [16]. When comparing static and dynamic techniques of program analysis, Ernst et al. admit that automatic tools fail in some cases where a human being would succeed.

Does extraction of closet contracts fall into the category of processes that cannot be fully automated? If so, can we at least automate part of the effort?

The analysis reported in this article has shown some regularity in the form and location of the closet contracts we can find in existing .NET components. In particular, preconditions tend to be buried in exception cases. Since method exception cases are not kept into the assembly metadata, we are currently exploring another approach: inferring routine preconditions from a systematic analysis of the CIL (Common Intermediate Language) code [32] of the .NET assemblies provided as input. More precisely, we are parsing the CIL code of .NET libraries — using Gobo Eiffel Lex and Gobo Eiffel Yacc [9] — to list the exceptions a method or a property may throw to infer the corresponding routine preconditions. The first results are promising.

10 Conclusion

This discussion has examined some evidence from the .NET libraries relevant to our basic conjecture: do existing libraries designed without a clear notion of contract contain some "contracts" anyway?

This analysis provides initial support for the conjecture. The contracts are there, expressed in other forms. Preconditions find their way into exceptions; postconditions and class invariants into remarks scattered across the documentation, hence more difficult to extract automatically.

The analysis reported here provides a first step in a broader research plan, which we expect to expand in the following directions:

- Applying the same approach to other .NET and non-.NET libraries, such as C++ STL (a first informal look at [26] suggests that there are contracts lurking there too).
- Investigating more closely the patterns that help discover each type of contract — class invariants, routine preconditions and postconditions — to facilitate the work of programmers interested in adding contracts a posteriori to existing libraries, with a view to providing an interactive tool that would support this process.
- Turning the Eiffel Contract Wizard into a Web service to allow any programmers to contribute contracts to .NET components.

This area of research opens up the possibility of various generalizations of this work in a broad investigation of applications of Design by Contract. (We are looking forward to seeing the evolution of the project conducted by Kevin McFarlane and aiming at providing a Design by Contract framework for use in .NET projects [19], of a current project at Microsoft Research about adding contracts into the C# language — see the "Assertions" section of [30] — and also of the new eXtensible C#© [35]; the outcome of these projects are likely to influence our research direction.)

Acknowledgements

This paper takes advantage of extremely valuable comments and insights from Éric Bezault (Axa Rosenberg), Michael D. Ernst (MIT), Tony Hoare (Microsoft) and Emmanuel Stapf (Eiffel Software). Opinions expressed are of course our own.

References [3] to [6] are previous versions of this work. Reference [7] is a summary version.

References

[1] Karine Arnout, and Raphaël Simon. "The .NET Contract Wizard: Adding Design by Contract to languages other than Eiffel". *TOOLS 39 (39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems)*. IEEE Computer Society, July 2001, p 14-23.

[2] Karine Arnout. "Eiffel for .NET: An Introduction". *Component Developer Magazine*, September-October 2002. Available from http://www.devx.com/codemag/Article/8500. Accessed October 2002.

[3] Karine Arnout, and Bertrand Meyer. "Extracting implicit contracts from .NET components". *Microsoft Research Summer Workshop 2002*, Cambridge, UK, 9-11 September 2002. Available from http://se.inf.ethz.ch/publications/arnout/workshops/microsoft_summer_research_workshop_2002/contract_extraction.pdf. Accessed September 2002.

[4] Karine Arnout. "Extracting Implicit Contracts from .NET Libraries". *4th European GCSE Young Researchers Workshop 2002*, in conjunction with *NET.OBJECT DAYS 2002*. Erfurt, Germany, 7-10 October 2002. IESE-Report No. 053.02/E, 21 October 2002, p 20-24. Available from http://www.cs.uni-essen.de/dawis/conferences/Node_YRW2002/papers/karine_arnout_gcse_final_copy.pdf. Accessed October 2002.

[5] Karine Arnout. "Extracting Implicit Contracts from .NET Libraries". *OOPSLA 2002 (17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*), Posters. Seattle USA, 4-8 November 2002. OOPSLA'02 Companion, ACM, p 104-105.

[6] Karine Arnout, and Bertrand Meyer. "Contrats cachés en .NET: Mise au jour et ajout de contrats a posteriori". LMO 2003 (*Langages et Modèles à Objets*). Vannes, France, 3-5 February 2003.

[7] Karine Arnout, and Bertrand Meyer. “Spotting hidden contracts: the .NET example”. Submitted for publication.

[8] Mike Barnett, and Wolfram Schulte. “Contracts, Components, and their Runtime Verification on the .NET Platform”. *Microsoft Research Technical Report TR 2002-38*, April 2002. Available from ftp://ftp.research.microsoft.com/pub/tr/tr-2002-38.pdf. Accessed April 2002.

[9] Éric Bezault. *Gobo Eiffel Lex and Gobo Eiffel Yacc*. Retrieved September 2002 from http://www.gobosoft.com.

[10] Eiffel Software Inc. *EiffelBase*. Retrieved October 2002 from http://docs.eiffel.com/libraries/base/index.html.

[11] Michael D. Ernst. “Dynamically Detecting Likely Program Invariants”. *Ph.D. dissertation, University of Washington*, 2000. Available from http://pag.lcs.mit.edu/~mernst/pubs/invariants-thesis.pdf. Accessed August 2002.

[12] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. “Dynamically Discovering Likely Program Invariants to Support Program Evolution”. *IEEE TSE (Transactions on Software Engineering)*, Vol.27, No.2, February 2001, p: 1-25. Available from http://pag.lcs.mit.edu/~mernst/pubs/invariants-tse.pdf. Accessed August 2002.

[13] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. “Quickly Detecting Relevant Program Invariants”. ICSE 2000 (*International Conference on Software Engineering*), Limerick, Ireland, 4-11 June 2000; Available from http://pag.lcs.mit.edu/~mernst/pubs/invariants-icse2000.pdf. Accessed August 2002.

[14] Michael D. Ernst, William G. Griswold, Yoshio Kataoka, and David Notkin. “Dynamically Discovering Program Invariants Involving Collections”, *Technical Report, University of Washington*, 2000. Available from http://pag.lcs.mit.edu/~mernst/pubs/invariants-pointers.pdf. Accessed August 2002.

[15] C.A.R. Hoare. “Proof of Correctness of Data Representations”. *Acta Infomatica*, Vol. 1, 1973, p: 271-281.

[16] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin: “Automated Support for Program Refactoring using Invariants”. *ICSM 2001 (International Conference on Software Maintenance)*, Florence, Italy, 6-10 November 2001. Available from: http://pag.lcs.mit.edu/~mernst/pubs/invariants-refactor.pdf. Accessed August 2002.

[17] Andrew Kennedy, and Don Syme. “Design and Implementation of Generics for the .NET Common Language Runtime”. *PLDI 2001 (Conference on Programming Language Design and Implementation)*. Snowbird, Utah, USA, 20-22 June 2001. Available from http://research.microsoft.com/projects/clrgen/generics.pdf. Accessed September 2002.

[18] Andrew Kennedy, and Don Syme. *Generics for C# and .NET CLR*, September 2002. Retrieved September 2002 from http://research.microsoft.com/projects/clrgen/.

[19] Kevin McFarlane. *Design by Contract Framework for .Net*. February 2002. Retrieved October 2002 from http://www.codeproject.com/csharp/designbycontract.asp and http://www.codeguru.com/net_general/designbycontract.html.

[20] Bertrand Meyer: *Object-Oriented Software Construction (1st edition)*. Prentice Hall International, 1988.

[21] Bertrand Meyer. "Applying 'Design by Contract'". *Technical Report TR-EI-12/CO, Interactive Software Engineering Inc.*, 1986. Published in *IEEE Computer*, Vol. 25, No. 10, October 1992, p 40-51. Also published as "Design by Contract" in *Advances in Object-Oriented Software Engineering*, eds. D. Mandrioli and B. Meyer, Prentice Hall, 1991, p 1-50. Available from http://www.inf.ethz.ch/personal/meyer/publications/computer/contract.pdf. Accessed April 2002.

[22] Bertrand Meyer: *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice Hall, 1994.

[23] Bertrand Meyer: *Object-Oriented Software Construction*, second edition. Prentice Hall, 1997.

[24] Bertrand Meyer, Raphaël Simon, and Emmanuel Stapf: *Instant .NET*. Prentice Hall (in preparation).

[25] Bertrand Meyer: *Design by Contract*. Prentice Hall (in preparation).

[26] Scott Meyers: *Effective STL*. Addison Wesley, July 2001.

[27] Microsoft. *.NET Collections library*. Retrieved June 2002 from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollections.asp.

[28] Microsoft. *.NET ArrayList class*. Retrieved June 2002 from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollectionsarraylistclasstopic.asp.

[29] Microsoft. *.NET ICollection interface*. Retrieved October 2002 from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollectionsicollectionclasstopic.asp.

[30] Microsoft Research. *Current research, Programming Principles and Tools*. Retrieved November 2002 from http://research.microsoft.com/research/ppt/.

[31] Richard Mitchell, and Jim McKim: *Design by Contract, by example*. Addison-Wesley, 2002.

[32] NET Experts. *ECMA TC39 TG2 and TG3 working documents*. Retrieved September 2002 from http://www.dotnetexperts.com/ecma/index.html.

[33] Jeremy W. Nimmer, and Michael D. Ernst. “Invariant Inference for Static Checking: An Empirical Evaluation”. *FSE ’02 (10th International Symposium on the Foundations of Software Engineering)*. Charleston, SC, USA. November 20-22, 2002. Proceedings of the ACM SIGSOFT. Available from http://pag.lcs.mit.edu/~mernst/pubs/esc-annotate.pdf. Accessed October 2002.

[34] Jeremy W. Nimmer, and Michael D. Ernst. “Automatic generation of program specifications”. *ISSTA 2002 (International Symposium on Software Testing and Analysis)*. Rome, Italy, 22-24 July 2002. Available from http://pag.lcs.mit.edu/~mernst/pubs/invariants-specs.pdf. Accessed October 2002.

[35] ResolveCorp. *eXtensible C#© is here!* Retrieved May 2003 from http://www.resolvecorp.com/products.htm.

[36] Raphaël Simon, Emmanuel Stapf, and Bertrand Meyer. “Full Eiffel on .NET”. *MSDN*, July 2002. Available from http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/pdc_eiffel.asp. Accessed October 2002.

[37] Software Engineering Institute. “Volume II: Technical Concepts of Component-Based Software Engineering”. *CMU/SEI-2000-TR-008*, 2000. Available from http://www.sei.cmu.edu/publications/documents/00.reports/00tr008.html. Accessed June 2002.

[38] Dave Thomas. “The Deplorable State of Class Libraries”. *Journal of Object Technology (JOT)*, Vol.1, No.1, May-June 2002. Available from http://www.jot.fm/issues/issue_2002_05/column2. Accessed June 2002.