Spotting hidden contracts: the .NET example

Karine Arnout
Karine.Arnout@inf.ethz.ch

Bertrand Meyer*
Bertrand.Meyer@inf.ethz.ch

Chair of Software Engineering
Swiss Federal Institute of Technology (ETH)
CH-8092 Zurich, Switzerland
*Also Eiffel Software, Santa Barbara, and Monash University, Melbourne
http://se.inf.ethz.ch

Abstract Can libraries written without explicit support for Design by Contract in the language or the method benefit from a posteriori addition of contracts? To explore this question of design and programming methodology, we performed an empirical study of library classes from the .NET Collections library, which doesn't use Design by Contract, to search for unexpressed contracts. This article reports on the buried contracts we have found, assesses improvements that might result from making the contracts explicit, and discusses the feasibility of automating the contract elicitation process.

Keywords: Design by Contract, Library design, Reuse, .NET, Metadata, Eiffel, Design Methods, Documentation, Contract Wizard

Article summary: Contracts are element of software specification - preconditions, postconditions, class invariants - included in the software itself. Some libraries, notably in Eiffel, make extensive use of contracts, but most commercial libraries don't. To understand whether this represents an intrinsic difference, we analyzed the.NET Collections library - which doesn't contain any explicit contracts - to see if contracts were there anyway, expressed in other ways. This analysis indeed revealed many implicit contracts. We present these results and compare the two styles of library design, with and without contracts. We discuss the idea of adding contracts a posteriori to existing library components and its consequences on documentation and testing. The article also identifies warning signs of possible implicit contracts, such as exception cases that may indicate postconditions and documentation comments that may indicate postconditions and invariants. It discusses methods to automate the contract elicitation process and presents the technique of dynamic invariant detection; it suggests a new automatic method to extract routine preconditions from .NET assemblies, applicable even when the source code is not available, thanks to .NET metadata and to the Common Intermediate Language (CIL) of .NET.

Biography of the authors:

Karine Arnout is a Ph.D. student at the Swiss Federal Institute of Technology (ETH), Zurich. Her research interests are in the area of "Trusted Components", in particular the correlation between contracts and tests. She worked at Eiffel Software in Santa Barbara, California, where she contributed at porting Eiffel to the .NET Framework, and built the first implementation of the Contract Wizard.

Bertrand Meyer is professor of software engineering at ETH, adjunct professor at Monash University, and founder and scientific advisor of Eiffel Software in Santa Barbara. His interest is developing quality software.

1. INTRODUCTION

"Design by Contract" denotes a distinctive style [6] [8] [11] of building software with precise descriptions - contracts - for each element, in the form of routine preconditions and postconditions and (in an object-oriented context) class invariants, together with precise rules regarding what happens to contracts in the inheritance structure and how to use them for design, implementation, documentation, debugging, testing and management.

Design by Contract is particularly appropriate for reusable libraries and is a key element of the design of some existing libraries [6], especially in Eiffel where the contract mechanisms are built-in. As long-time practitioners of these techniques we are convinced of their benefit and surprised that many recent libraries, including the official libraries for Java and .NET, or the C++ Standard Template Library, do not explicitly rely on them. So the work reported in this article started out as a sanity check on ourselves: do we (and others using these techniques) see contracts everywhere simply because our development environment makes them natural to use, or are they intrinsically present anyway, even when other designers don't express or even perceive them?

This question is important because many experts see reuse and component-based development as the key to the future of software development, and there are few comprehensive methodological frameworks for building components. Design by Contract is one, so it is essential to understand whether it's needed or not. This leads to the basic question explored by our work:

The Closet Contract Conjecture

Are contracts in libraries an artefact of some programming languages, or do good libraries rely on contracts anyway, possibly implicit or disguised?

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 [10], a comprehensive set of data structure implementations and the most recent addition to the world's collection of fundamental libraries, has been one of our 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 and postcondition clauses, class invariants.

Where we spotted closet contracts, we proceeded to out them by producing new versions of the classes, which retaining the original APIs but make the contracts explicit.

The rest of this presentation describes the analysis and its outcomes

2. THE CONTEXT

A distinctive design style

Applying to reusable libraries the ideas of Design by Contract 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, and resulting libraries **[7]** appear to be reliable and convenient to use. A recent report by the Software Engineering Institute **[13]** confirms that for components in general — not just classes — the use of contracts is 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. The difference is clear, for example, in a comparison of two libraries that cover some of the same ground: EiffelBase **[7]**, which is based on Design by Contract, and the .NET Collections library **[10]**, 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, 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.

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 helping this 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).

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 much 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.
- **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 can affect the original semantics (preconditions can only be weakened, postconditions can only be strengthened).
- **Management**: Contracts allow project managers and decision makers to understand the global purpose of a program without having to go 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.

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; must be ensured by every creation procedure and maintained by every exported routine.

- **Check instructions**: "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 properties of library interfaces and will not be considered further.

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 **[4]**). We must make sure that our contract elicitation process doesn't overlook them.

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. A contract — here a precondition and a postcondition — will express 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 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.

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 assertions, where will we find the implicit contract?
- Is there always a contract — expressed or not — as in this example?

The 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 substantial specification information that appears directly useful to the elicitation contract process.

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* [8].

In addition to predefined categories including the assembly name, version, dependencies, and so on, developers can define, assuming proper source language support, their own specific kinds of metadata in the form of **custom attributes**.

.NET libraries and the Contract Wizard

Metadata of both kinds — predefined and custom — open attractive new possibilities. The Contract Wizard is one of them: through interactive examination of a class and its features thanks to the metadata, it enables users to add any appropriate contracts without having access to the source code.

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.

5. ANALYSIS OF A COLLECTION CLASS

We will now describe the result of scouting class ArrayList, part of the core .NET library (*mscorlib.dll*), for closet contracts. The choice of this class for this presentation 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 way to possibly interesting comparisons after the contract elicitation process has been completed.

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 all the exported routines of the class. Examining all of them confirms this to be the case. This indicates 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 **[10]**: 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
```

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 two following 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; for example 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*.

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_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 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.

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 **[7]** 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 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 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.

A contracted form of the .NET arrayed list class

Let us go through the contracted version of routines Add and IndexOf we analyzed previously. (We will need to review a few other features as well to fully understand the contracts of Add and IndexOf.)

Here is the contracted variant of feature Add with the two preconditions — tagged writable and extendible — we discovered before. It also expresses a postcondition (with different clauses), about which we haven't talked — because its elicitation process is similar to the one shown for

the query IndexOf — ; the header comment of feature Add should suffice to 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)
        value_is_last_item: item (count – 1) = 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))
```

The comment *-- from ILIST* shows that the following assertion clauses (routine preconditions or postconditions) are inherited from the parent class *ILIST*. (*ILIST* is the contracted version of the .NET interface IList that class ArrayList implements.)

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

The postcondition of *add* 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 has to be a valid index for the list), *count* and *capacity* (for measurements). The contracts of query IndexOf (the example we chose to highlight buried postconditions in .NET classes) also rely on these routines:

```
index_of (value: ANY): INTEGER
        -- Zero-based index of the 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
```

Metrics

Measurements of properties of the contracted class *ARRAY_LIST* show that:

- **62% of the routines now have a contract** (a precondition or a postcondition, usually both): 33 out of 52 routines in the class.
- The 33 routines with preconditions tend to have **more than one precondition clause: 2.5 on the 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).

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.

Interfaces

First, let us 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.

	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 %**	**50 %**
Postconditions rate	**64 %**	**25 %**	**100 %**	**50 %**
Class invariants	0	0	0	0

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

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. 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*.

Other classes: Stack and Queue

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

- Are concrete collection classes.
- Have no relation with 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' propensity to include contracts.

The same approach was applied to these classes as to ArrayList and its parents. Resulting measurements for the contracted versions *STACK* and *QUEUE* **confirm the trends previously identified**:

- Preconditions and postconditions are present. Class *STACK* has a 29% preconditions rate (17 routines, including 5 equipped with preconditions) and a 59% (10 routines equipped with postconditions out of 17) postconditions rate; class *QUEUE* has 42% (8 out of 19) and 58% (11 out of 19).
- Routines usually have several preconditions and postconditions. For example, *STACK* has 16 postcondition assertions for only 10 routines equipped with contracts.
- 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*.

8. AUTOMATIC CONTRACT EXTRACTION

If manual analysis - as we have performed so far - can find some hidden contracts, the inevitable question follows:: could one synthesize these contracts automatically than manually?

Dynamic contract inference

Previous work on dynamic inference seeks to derive assertions from captured variable traces by executing a program — whose source code is available — with various inputs. (It relies 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.

An example of such a tool is Daikon **[3]** by Michael D. Ernst, which tries to find class and loop invariants as well as routine pre- and postconditions. Its first version was limited to finding contracts over scalars and arrays; the next one (Daikon 2) enables contract discovery over collections of data, and computes conditional assertions.

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).

Extracting routine preconditions from exception cases

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

The analysis reported above has identified some clear patterns in the form and location of the implicit contracts we can find in existing .NET components. In particular, preconditions tend to be buried under exception cases. Since method exception cases are not kept into the assembly metadata, we are currently exploring another approach: parsing the CIL (Common Intermediate Language) code of .NET libraries to list the exceptions a method or a property may throw to infer the corresponding routine preconditions.

The methodological perspective

The question of automatic analysis - which many people have asked when exposed to initial presentations of this work - is legitimate but should not obscure the underlying methodological issue: contracts bring their best benefits if they are used *as part of the development process* rather than as a designer's *pentimento* (late remorse). No automatic tool will derive the best contracts, those which express 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 it. They should be explicitly written by the component authors, not left to 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 limits the hopes that we can put on it. We should not encourage designers to code components without paying attention to contracts and expect that, somehow, some tool will invent the contracts for them afterwards. 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 these explicit methods, and (as permitted by tools like Daikon) to help improve the quality of existing software whose design did not consider contracts.

9. 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?

Our study 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.
- 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.

Many opportunities exist to extend and generalize this work to a broad investigation of Design by Contract applications; an example is the project conducted by Kevin McFarlane to provide a contract framework for .NET projects **[5]**. We hope that such developments will help improve reusable components through explicit and implicit contracts.

ACKNOWLEDGEMENTS

We have benefited from 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 our own. Reference [2] is a longer version of this article.

BIBLIOGRAPHY

[1] K. Arnout and R. Simon. "The .NET Contract Wizard: Adding Design by Contract to languages other than Eiffel". *TOOLS 39*, IEEE Computer Society, July 2001, pp. 14-23.

[2] K.Arnout and B. Meyer, "Elicitation of closet contracts from .NET components: Benefits for the library users", to appear in *Formal Methods and Components (FMCO)*, eds. Frank S. deBoer et al., Lecture Notes in Computer Science, Springer-Verlag, 2003.

[3] M. D. Ernst. "Dynamically Detecting Likely Program Invariants". *Ph.D. dissertation, University of Washington*, 2000; http://pag.lcs.mit.edu/~mernst/pubs/invariants-thesis.pdf.

[4] C.A.R. Hoare. "Proof of Correctness of Data Representations". *Acta Infomatica*, Vol. 1, 1973, pp. 271-281.

[5] K. McFarlane. *Design by Contract Framework for .Net*; http://www.codeproject.com/csharp/designbycontract.asp; http://www.codeguru.com/net_general/designbycontract.html.

[6] B. 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, pp. 40-51.

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

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

[9] B. Meyer, R. Simon and E. Stapf: *Instant .NET*. Prentice Hall (in preparation).

[10] Microsoft. *.NET Collections library*; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollections.asp.

[11] R. Mitchell and J. McKim: Design by Contract, by example. Addison-Wesley, 2002.

[12] R. Simon, E. Stapf and B. Meyer. "Full Eiffel on .NET". *MSDN*, July 2002; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/pdc_eiffel.asp.

[13] Software Engineering Institute. "Volume II: Technical Concepts of Component-Based Software Engineering". *CMU/SEI-2000-TR-008*, 2000; http://www.sei.cmu.edu/publications/documents/00.reports/00tr008.html.