Extracting implicit contracts from .NET libraries

Karine Arnout

Chair of Software Engineering – Bertrand Meyer
Swiss Federal Institute of Technology, Zürich
Karine.Arnout@inf.ethz.ch
http://se.inf.ethz.ch/people/arnout

Keywords: Design by Contract™, implicit contracts, .NET, metadata, Eiffel, Contract Wizard, reuse, library design

Classification: 2 months' work in 1st year of Ph.D. studies

Abstract. Do libraries written without explicit support for Design by Contract in the language or the design methodology contain implicit contracts? To help answer this question, we have performed an empirical study of some existing non-contracted library classes from the .NET Collections library. This paper reports on what we have found, and discusses whether the results could be used to improve the design of the classes and make them easier to learn and use.

1 Introduction

Applying to reusable libraries the ideas of Design by Contract [1] mean 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.

The experience of Eiffel libraries [2] shows that systematic application of these principles leads to significant improvements in library design. A recent report by the Software Engineering Institute [3] 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 so far, mostly in the Eiffel context, 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: the EiffelBase library, based on Design by Contract, and the .NET Collections library.

Because the resulting styles appear so different, it is important to ask what happens in a contract-less environment to the library properties that the Eiffel designer would have expressed in preconditions, postconditions and class invariants. Is the contract-rich style of Eiffel libraries an artifact of the presence of contract mechanisms in the method, language and tools? Or are the contracts lurking anyway under the cover in

libraries for contract-less environments, whether suppressed or replaced by such ersatz techniques as explanations in user documentation or exceptions?

The search for implicit contracts is particularly interesting in the .NET context because the flexibility of the component model has enabled the development of a "Contract Wizard" [4] — 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** contained in every assembly. By nature, however, the Contract Wizard is only interesting if the conjecture discussed here holds: that in every contract-less library there are contracts trying to make themselves heard.

This paper is an empirical study exploring this conjecture. Using some classes of the .NET Collections library as our study target, we look for implicit contracts. Since the question is too subtle to allow automatic analysis by a software tool, we have examined the .NET classes manually, searching for the various forms that contracts can take when they cannot be expressed as contracts. The benefits we expect from this examination include:

- Assessing whether some "contracts" exist in any carefully-designed library, even if it has not been written in a language supporting contracts.
- Assessing the usefulness of contracting .NET libraries a posteriori.

Section 2 analyzes class `ArrayList` of the .NET Collections library. In light of the results, section 3 endeavors to build a contracted form of class `ArrayList` and compares it with its counterpart in EiffelBase, class *`ARRAY_LIST`*.

2 Analysis of Class `ArrayList`

Let us now examine class `ArrayList`, part of the core .NET library (*mscorlib.dll*) to find out whether .NET components are equipped with some form of contracts.

2.1 Implicit Class Invariants

The comments found in the .NET documentation for `ArrayList` reveal information about the class that falls into the category of class invariants in Design by Contract.

For example, one sentence of the documentation describing a possible constructor for `ArrayList` is: "*The default initial capacity for an ArrayList is 16*", stating, almost as if in passing, that the capacity of the created object is positive. This property might at first appear to signal a postcondition. But we may notice that all three constructors of the class set the initial capacity of the list to a positive value. Since there is no other way to create an object of type `ArrayList` than to call one of these three constructors, such a global property of the class — which is satisfied after creation of the new instance and preserved by any further call to any other class routine — denotes a class invariant, which in Eiffel syntax would be expressed as:

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

Further exploration of the documentation helps deduce another class invariant for class `ArrayList`. The specification of the `Count` property mentions that "`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, in Eiffel syntax:

```
invariant
        valid_count: count <= capacity
```

2.2 Implicit Routine Preconditions

Aside from implicit class invariants, the documentation also includes valuable information about implicit routine contracts.

For example, the specification of method `Add` of class `ArrayList` mentions that `Add` may throw an exception of type `NotSupportedException`, the condition being that the array list is read-only or has a fixed size. It means that the underlying code first ensures that the list on which the method is called is writable (not read-only) and extendible (does not have a fixed size) before adding `value` at the end of the list: this is a working requirement, namely a routine precondition in terms of Design by Contract. In Eiffel, one would write two preconditions of the form:

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

This example — several similar ones can be found in the reference documentation — suggests a quite easy way to extract preconditions from .NET routines by using exception cases: simply read the exception condition (e.g. the array list is read-only) and take the opposite (required condition: `!IsReadOnly`) to obtain the underlying routine precondition: "`writable: not` *`is_read_only`*", in Eiffel syntax.

2.3 Implicit Routine Postconditions

Let us now examine whether the documentation provided with the .NET Framework also reveals unwitting routine postconditions.

For instance, part of the specification of query `IndexOf` — whose signature is: `public virtual int IndexOf (object value);` — mentions that the return value is: "*the zero-based index of the first occurrence of value within the entire ArrayList, if found; otherwise, -1*". The documentation clearly separates two cases: on the one hand, when the array list actually contains the value passed as argument to the routine, and on the other hand, when this value is not in the list. In the first case, the result is the index of the first occurrence of `value` found in the list: thus between zero (.NET lists are zero-based) and `count` (the number of elements in the list); in the second case, the result is -1. These properties we have just expressed informally result from the execution of query `IndexOf`, i.e. they fit in the category of routine postconditions described in Design by Contract. In Eiffel syntax, one would write:

```
ensure
    valid_result: contains (value) implies
                  Result >= 0 and Result < count
    not_found: not contains (value) implies Result = -1
```

This example reveals that routine postconditions do exist in .NET libraries, although not explicitly expressed because of the lack of support from the underlying

environment. But contrary to preconditions which appear quite easily extractable in an automatic process, routine postconditions are likely to require a case by case examination involving a human-being, since they are mainly hidden behind remarks in the reference documentation, thus requiring the understanding of a person.

3 A Posteriori Addition of Contracts

The preceding analysis of .NET class `ArrayList` suggests that we construct a contracted variant, *`ARRAY_LIST`*, which has the same interface as the original `ArrayList` plus the elicited contracts. We will present a sketch of this class and compare it with EiffelBase's *`ARRAYED_LIST`*.

3.1 A Contracted Form of Class `ArrayList`

Part of the interface of such a contracted class *`ARRAY_LIST`* would be similar to:

```
class interface
        ARRAY_LIST
create
        make
feature -- Initialization
        make is
            -- Create and initialize.
          ensure
            capacity_set: capacity = 16
            writable: not is_read_only
            extendible: not is_fixed_size
          end
invariant
        positive_capacity: capacity >= 0
        valid_count: count <= capacity
end -- class ARRAY_LIST
```

3.2 The Benefits of Contracts

An a posteriori addition of contracts to .NET libraries would have many beneficial consequences. In particular, it would simplify the task of clients — comparing to the original `ArrayList`. As a matter of fact, the lack of contracts is very likely to lead to a very defensive programming style (see C# program code below), which is heavy and inefficient.

```
ArrayList MyList = new ArrayList();
if( MyList.Capacity == 16 ){
        if((!MyList.IsReadOnly)&&(!MyList.IsFixedSize))
            Index = MyList.Add( "Hello World !" );
```

On the contrary, a client using the contracted version of `ArrayList`, *`ARRAY_LIST`*, does not need to check that each routine's execution has completed successfully: it simply relies on the contracts. Therefore the code required to perform the same task as before becomes lighter and thus more efficient:

```
create my_list.make
index := my_list.add ("Hello World !")
```

4 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 [5] 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 ISE 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.

References

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

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

[3] Software Engineering Institute, Volume II: Technical Concepts of Component-Based Software Engineering, CMU/SEI-2000-TR-008, 2000

[4] Karine Arnout, Raphaël Simon, The .NET Contract Wizard: Adding Design by Contract to languages other than Eiffel, TOOLS 39, IEEE Computer Society, p:14-23, July 2001.

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