Chair of
Software Engineering

Model-based Contracts for .NET Collections

Software Engineering Laboratory: Open Source Eiffel Studio

By: Tobias Kiefer

Supervised by: Nadia Polikarpova

Prof. Dr. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Abstract

After the *Mathematical Model Library* had been ported from Eiffel to the .NET platform, a huge amount of software suddenly became available to practically test the usefulness of the *Model Based Contracts* software verification approach on. As a continuation of the first attempts in that direction, this project was aimed at extending the automatic testing experiments of *MBC*-enhanced software to a more complex and probably more professionally developed piece of software: the generic collection classes of the Microsoft .NET framework. As opposed to the previous efforts on the Eiffel and .NET platforms, a special challenge inherent to this project was that the target code is closed-source.

Contents

1. Introduction

After the *Mathematical Model Library (MML)* had been ported from Eiffel to the .NET platform during a previous project[1], a huge amount of .NET software suddenly became available to practically test the usefulness of the *Model Based Contracts (MBC)*[2] software verification technique on. As a first experiment, as part of the aforementioned porting project, the open source *Data Structures and Algorithms (DSA)*[3] library had been tested using the *MML* and the *Pex*[4] testing tool. These experiments yielded promising results: several bugs as well as minor design faults were automatically detected.
As a continuation of these efforts this project was aimed at extending these kinds of automatic testing experiments of *MBC*-enhanced software to a more complex and probably more professionally developed piece of software: the generic collection classes of the Microsoft .NET framework. Probably the most notable difference in contrast to the previous, DSA related project is that the .NET framework is closed-source. How this additional challenge was handled is (amongst other things) decribed in Section 2, which deals with the process of contract development, while the rest of this section will briefly introduce the relevant parts of the .NET framework that were targeted during this project. Section 3 describes the actual testing experiments and their results, followed by Section 4 which concludes this report.

Although the relevant classes of the collection framework are spread across multiple libraries, they are all contained in the `System.Collections.Generic` namespace. Figure 1.1 provides an overview of all major collection classes in this namespace, omitting smaller classes that were not considered relevant for contract enhancement. Effective classes[1](blue) were grouped beneath the interfaces (green) they implement. It is observable that the hierarchy is less flat than in the DSA library. With the exception of `Stack<T>` and `Queue<T>`, all classes implement the `ICollection<T>` interface, and therefore the hierarchy levels below (and including) this interface were serving as a basis for the structure of "contract-class" wrappers that will be described in the next section.

[1]Non-abstract, non-interface classes

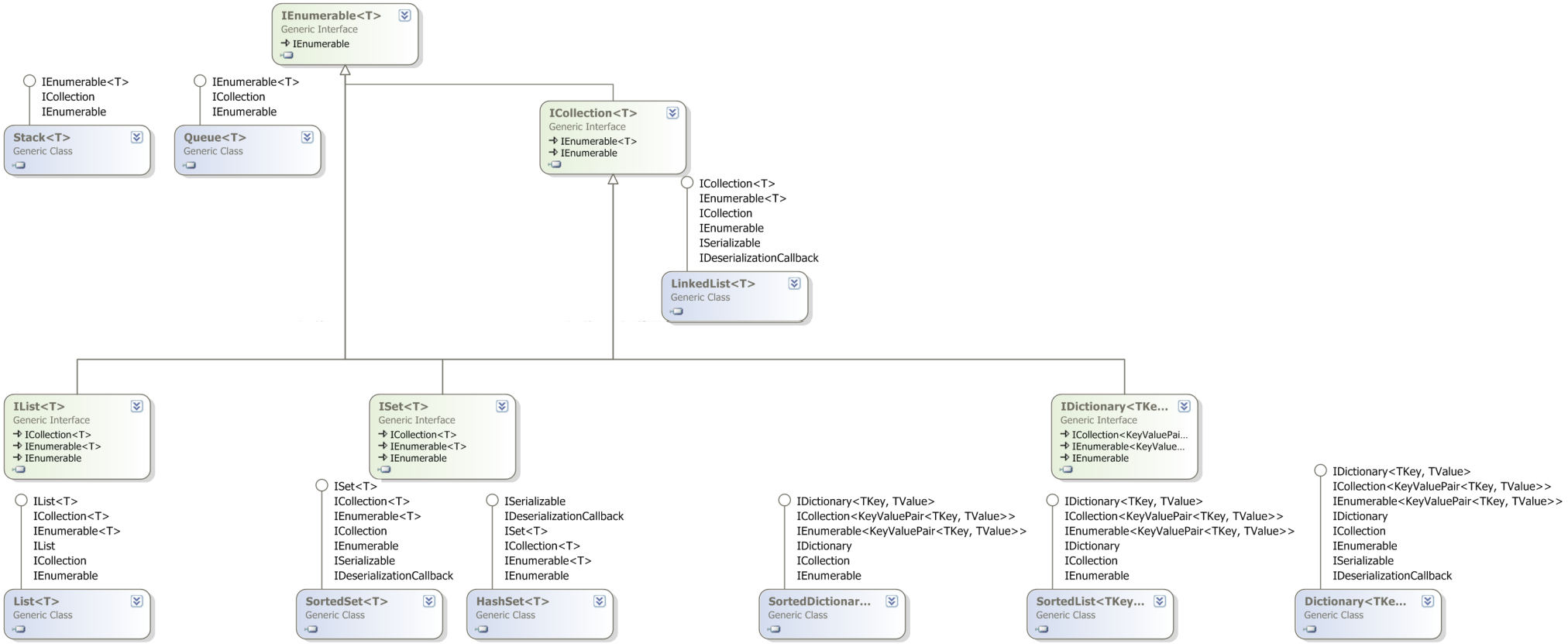

Figure 1.1.: A class diagram of the `System.Collections.Generic` namespace

2. Model-Based Contracts for .NET Generic Collection Classes

2.1. Contract Development

Since the .NET framework is closed-source, there was no immediate way to write the contracts directly into the original code. Although the *.Net Reflector*[5] decompilation software was used to take a look at the implementation when necessary, the main approach was to write *wrapper classes* for the original .NET collection classes. These wrapper classes contain all functions of their "target classes", except for functions that were not considered relevant for enhancing them with contracts. Additionally, they contain a reference to an instance of the target class, so that in every "wrapped" function the original function can be called with the appropriate parameters.
Because the original function is the only code that is executed within a wrapper function, contracts added to the wrapper functions have the same effect as if they were directly added to the original function. Listing 2.1 illustrates this approach with a simple example. The target object in this case is `m_ListTarget`, an object of type `System.Collections.Generic.List`. Target objects are created in the constructors of the wrapper objects, so one can be sure that there is always an instance available. Also, the target object references are not changed throughout the lifetime of a wrapper object.
Another important aspect is that wrapper classes derived from another wrapper class have multiple target object references. It was decided that a wrapper class should only inherit another wrapper class if there is a similar relation between their target classes. For example, `ListContracts` inherits `ICollectionContracts` because `List` implements `ICollection` (see Figures 2.1, 1.1). So in addition to the `m_ListTarget` field that was already mentioned, `ListContracts` also inherited the field `m_IColTarget`, which is a reference of type `ICollection` to the same object. Listing 2.2 shows an example of this concept: The first constructor, which in its signature exactly resembles the

respective original `List` constructor, creates a new object of type `List`, then passes it to the private constructor. The private constructor stores a reference of type `List` and calls a similar constructor in the base class, where the same object reference will be stored as a reference of type of the base type of `List` (`ICollection` in this case).
This approach can naturally be extended to deeper hierarchies, and the costs of having one additional field per hierarchy level are easily justified by the practical usefulness of inheriting large amounts of contracts from base classes using virtual functions. Figure 2.1 shows the inheritance structure of the wrapper classes. The approach described above was used extensively throughout all related classes, i.e., contracts for virtual functions were put as high as possible in the inheritance tree, and were mostly only refined in derived classes. For example, `SortedDictionaryContracts` uses the contracts already defined in `IDictionaryContracts`, and adds an additional model query of type `MML.Sequence` along with some contracts to check for the "sortedness" of the data. A complete List of contract-enhanced classes along with their models[1]is shown in Table 2.1.

Listing 2.1: A wrapper function for `List.Exists` in the `ListContracts` class

```
public bool Exists(Predicate<T> match)
{
    Contract.Ensures(Contract.Result<bool>()
                     == m_Sequence.Range.Exists(match));
    Contract.Ensures(Contract.OldValue(m_Sequence)
                                          == m_Sequence);
    return m_ListTarget.Exists(match);
}
```

Listing 2.2: Some constructors of `ListContracts`

```
public ListContracts(int capacity) : this(new List<T>(capacity))
{
    Contract.Ensures(m_Sequence.IsEmpty());
}
private ListContracts(List<T> list) : base(list)
{
    m_ListTarget = list;
}
```

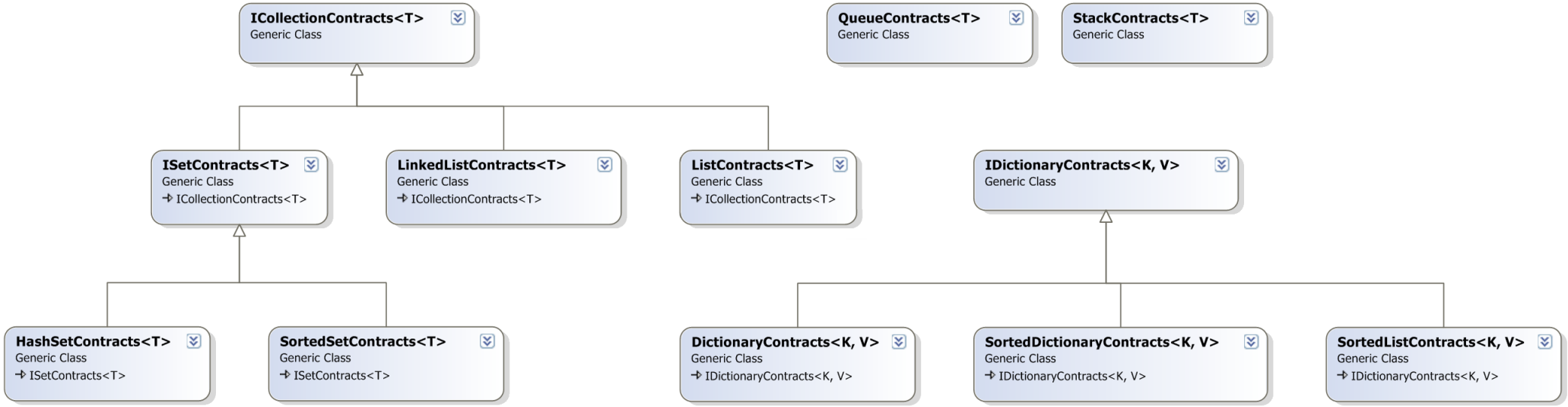

Figure 2.1.: A class diagram of the wrapper classes containing the contracts

Table 2.1.: Contract-enhanced collection classes and their models

Class	Model
Dictionary	public MML.Map<K, V> m_Map
HashSet	public MML.Set<T> m_Set
ICollection	public MML.Bag<T> m_Bag
IDictionary	public MML.Map<K, V> m_Map
ISet	public MML.Set<T> m_Set
LinkedList	public MML.Sequence<T> m_Sequence
List	public MML.Sequence<T> m_Sequence
Queue	public MML.Sequence<T> m_Sequence
SortedDictionary	public MML.Sequence<K> m_Sequence public MML.Map<K, V> m_Map
SortedList	public MML.Sequence<K> m_Sequence public MML.Map<K, V> m_Map
SortedSet	public MML.Sequence<T> m_Sequence public MML.Set<T> m_Set
Stack	public MML.Sequence<T> m_Sequence

2.2. Incomplete Contracts

A contract is incomplete if it does not fully capture the effect of a method in regard to the chosen model and the return value of a function[2]. When, in cases like this, the software has not been developed with model-based contracts in mind, it often can't be avoided that a certain amount of contracts remains incomplete. Below is a list of all methods with incomplete contracts[2]:

- In class `LinkedList`:
  - `public LinkedListNode<T> First`
  - `public LinkedListNode<T> Last`
  - `public void AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode)`
  - `public LinkedListNode<T> AddAfter(LinkedListNode<T> node, T value)`

[1]Note that in this table and throughout the remaining document for readability reasons the original class names are used, although in a strict sense the contract-enhancements are only present in the wrapper classes (suffixed with "Contracts"). In a similar manner, generic parameters will often be omitted.

  - `public void AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode)`
  - `public LinkedListNode<T> AddBefore(LinkedListNode<T> node, T value)`
  - `public void AddFirst(LinkedListNode<T> node)`
  - `public void AddLast(LinkedListNode<T> node)`

- The `TrimExcess()` functions as well as all constructors taking an `int capacity` parameter in the following classes:
  - `Dictionary`
  - `HashSet`
  - `List`
  - `Queue`
  - `SortedList`
  - `Stack`

Similar to the DSA linked list classes, the interface functions of the `LinkedList` collection class expose the internal node structure of the list by accepting and returning objects of type `LinkedListNode<T>`. These implementation-specific details are not captured by the `MML.Sequence` model that was chosen for this class, therefore the functions listed above remain incomplete. Another implementation related detail that was chosen not to be modeled is the `capacity` property that some classes use. This causes some constructors and functions to be incomplete as seen above.

Because the `System.Collections.Generic` namespace is part of a complex software framework, there were a lot of methods contained in the classes that were not considered relevant for their collection-specific functionality. Thus there is a relatively long list of methods that were not enhanced by contracts provided in Appendix A.

[2]Unless not noted otherwise, during this section a method listed as having incomplete contracts in a base class (see Figure 2.1) implies that the corresponding method in a derived class also has incomplete contracts. This applies analogously to methods being listed as having no contracts.

2.3. Specification Overhead

The specification overhead measured in working-time and the amount of additional code is a very important factor for reasoning about the usefulness of a specific approach. The time needed for developing the models and writing the specifications (contracts) for the collection classes mentioned in the earlier sections was about 60 person-hours for a person who previously had not been familiar with that code.
A detailed summary of the specification overhead in terms of source code is given in Table 2.2. The first column describes the size of the original code, i.e. without any contracts, measured in *lines of code (LOC)*. These values were obtained using the *.NET Reflector* decompilation software. In the second column the amount of code added during the contract-writing process is given. This includes code for pre- and postconditions, invariants, model queries, and additional helper functions, but not the additional code that was needed to create the wrapper classes for the contracts. Therefore these values can be seen as the overhead that would occur if the contracts were written directly into the implementation code. It should be noted that for obtaining the LOC values, empty lines as well as comments and lines containing only brackets were not counted.
The third column gives the amount of additional helper routines that were added during the contract development phase, and the remaining columns describe in detail the numbers of contracts that were added. It can be seen that there is a large amount of postconditions, which constitute the main component of the model-based contracts technique. Preconditions were not newly added during this project, but only retained from an earlier approach to write contracts for the collection classes. Because the `List` class is the only implementation of the `IList` interface, contracts for both classes were merged in the wrapper class `ListContracts`. Thus the overhead values of `List` cover both the original interface and implementation classes.

Table 2.2.: Specification overhead for the .NET collection classes

Class	**LOC source**	**LOC contracts**	**Routines added**	**Preconditions added**	**Postconditions added**	**Invariants added**
`Dictionary`	509	8	0	0	8	0
`HashSet`	595	7	0	0	7	0
`ICollection`	16	35	3	1	5	2
`IDictionary`	16	41	2	0	20	1
`ISet`	24	37	0	11	18	3
`LinkedList`	399	46	1	0	29	2
`List`	559	83	1	0	70	2
`Queue`	235	23	1	0	14	1
`SortedDictionary`	299	20	1	0	6	3
`SortedList`	367	29	1	0	15	3
`SortedSet`	903	28	1	0	15	2
`Stack`	200	23	1	0	14	1
`Total`	4122	380	12	12	221	20

3. Automatic Testing of the MBC-Enhanced Collection Classes

3.1. The Testing Project

Just like during the previous, DSA related, testing experiments the Pex[4] testing tool was used. The first version of the testing project was auto-generated by Pex and then modified manually. For example, test classes generated for wrappers of interface classes (e.g. `ISetContracts`) were removed, and the testing functions were copied directly into the test classes of all relevant effective classes. Therefore there are only nine test classes as opposed to 12 contract classes. In addition to these changes, factory classes had to be added to the test project in order to help Pex in instantiating most of the types. Also, like in the DSA project, the tests were manually restricted to generic parameters of type `int` for the sake of simplicity.

3.2. The Testing Procedure and Results

The approach used in testing the `System.Collections.Generic` classes was similar to the previous experiments involving Pex and model-based contracts: The testing tool was run until either it found an error or reached one of the limiting parameter values. In the latter case, the limit was increased and the Pex exploration process was started again. In addition to `MaxBranches`, which has been the main limiting parameter during the DSA test runs, several other parameters had to be significantly increased as well during the experiments. The `MaxConstraintSolverTime` parameter had to be increased from 2 to 24 in most classes, and `MaxRuns` was raised from a default value of 100 to 1500. In addition to that, the

Table 3.1.: Contract-enhanced collection classes and their testing paramters

Class	Max Branches	Constr. Solver	Max Runs	Testing Time
`Dictionary`	40000	24	1500	0:19:20
`HashSet`	40000	24	1500	3:20:15
`LinkedList`	40000	24	1500	2:53:06
`List`	40000	24	1500	7:55:18
`Queue`	20000	12	100	1:10:36
`SortedDictionary`	40000	24	1500	0:54:19
`SortedList`	40000	24	1500	1:29:50
`SortedSet`	40000	24	1500	3:32:56
`Stack`	40000	24	1500	1:58:29
`Total`	40000	24	1500	23:39:09

`timeout` property was set to 800 (default: 120). Table 3.1 shows some of these parameters including the length of the test runs for every test class. It is notable that the total testing time was about twice as long as the one from the DSA testing experiments [1].

During the tests a lot of warnings and exceptions showed up, but unfortunately none of these revealed a real bug in the target code. Some of the functions were especially problematic to test since they were taking a delegate function as an argument, and the delegate functions that were generated by Pex proved to be unsuitable for verification with model based contracts. That was because these functions basically return different values in two subsequent calls, regardless of the input. So when they had been called the first time by the actual function-under-test, their return value differed from the second time they were invoked by the `MML` code, and thus the contract assertion could never evaluate to true. Although the scope of this project did not permit for finding a workaround or solution for this problem, the discovery of this previously unknown correlation between Pex delegates and the MBC approach can be seen as a result of the experiments.
In addition to that, the testing phase revealed bugs regarding the `Sequence.ButLast` and the `Map.Removed` functions in the C# version of the *Mathematical Model Library* which have been fixed during this project. Therefore the improvement of the C# MML code could be seen at least as a positive side effect of the testing experiments.

[1] 12:07:11, see [1]

4. Conclusion

Unlike the previous C# experiments, this verification project was targeted at very professionally developed software, so it was almost to be expected not to find any issues in the code. On the other hand, testing can never reveal the absence of bugs, and besides that, Pex works in a probabilistic way. So there is, at least theoretically, a chance to still find some bugs in these classes using the MBC approach and probably even larger amounts of contracts and testing time. If this is worth the effort is questionable at least though. Like with most experiments there were brought up some additional unexpected things: some MML related bugs were fixed, and issues regarding model-based contracts and Pex delegates were discovered. In the end, that there haven't been any bugs revealed in one of the most widely used and approved piece of software infrastructure should not blur the fact that during this project the C# version of the *Mathematical Model Library* and the MBC approach in general have proven to be practically applicable even to larger, more complex .NET software structures and APIs.

A. List of Missing Contracts

The following list provides an overview of functions from classes in the `System.Collections.Generic` namespace that were not considered relevant for enhancement with model-based contracts. If a function was already listed in a parent class, it is not listed again in derived classes (See Figure 1.1). If a class is not listed, it means that it contains no additional functions with missing contracts.

- In class `Dictionary`
  - All `CopyTo()` methods
  - `protected Dictionary(SerializationInfo info, StreamingContext context)`
  - `public IEqualityComparer<TKey> Comparer { get; }`
  - `public virtual bool Equals( Object obj)`
  - `protected virtual void Finalize()`
  - `public virtual int GetHashCode()`
  - `public virtual void GetObjectData(SerializationInfo info, StreamingConte context)`
  - `public Type GetType()`
  - `protected Object MemberwiseClone()`
  - `public virtual void OnDeserialization(Object sender)`
  - `public virtual string ToString()`
- In class `HashSet`
  - All `CopyTo()` methods
  - `protected HashSet(SerializationInfo info, StreamingContext context)`
  - `public IEqualityComparer<T> Comparer { get; }`
  - `public virtual bool Equals( Object obj)`
  - `protected virtual void Finalize()`
  - `public virtual int GetHashCode()`
  - `public virtual void GetObjectData(SerializationInfo info, StreamingConte context)`
  - `public Type GetType()`
  - `protected Object MemberwiseClone()`
  - `public virtual void OnDeserialization(Object sender)`
  - `public virtual string ToString()`
- In class `ICollection`:
  - `bool IsReadOnly { get; }`
  - `void CopyTo(T[] array, int arrayIndex)`

  - `IEnumerator GetEnumerator()`

- In class `ISet`:
  - All `CopyTo()` methods

- In class `LinkedList`:
  - All `CopyTo()` methods
  - `protected LinkedList(SerializationInfo info, StreamingContext context)`
  - `public virtual bool Equals( Object obj)`
  - `protected virtual void Finalize()`
  - `public virtual int GetHashCode()`
  - `public virtual void GetObjectData(SerializationInfo info, StreamingConte context)`
  - `public Type GetType()`
  - `protected Object MemberwiseClone()`
  - `public virtual void OnDeserialization(Object sender)`
  - `public virtual string ToString()`

- In class `List`:
  - All `CopyTo()` methods
  - `public int Capacity { get; set; }`
  - `public ReadOnlyCollection<T> AsReadOnly()`
  - `public virtual bool Equals( Object obj)`
  - `protected virtual void Finalize()`
  - `public virtual int GetHashCode()`
  - `public List<T> GetRange(int index, int count)`
  - `public Type GetType()`
  - `protected Object MemberwiseClone()`
  - `public virtual string ToString()`

- In classes `Queue` and `Stack`
  - All `CopyTo()` methods

  - public virtual bool Equals(Object obj)
  - protected virtual void Finalize()
  - public virtual int GetHashCode()
  - public Type GetType()
  - protected Object MemberwiseClone()
  - public virtual string ToString()
- In class SortedDictionary
  - All CopyTo() methods
  - public IComparer<TKey> Comparer { get; }
  - public virtual bool Equals(Object obj)
  - protected virtual void Finalize()
  - public virtual int GetHashCode()
  - public Type GetType()
  - protected Object MemberwiseClone()
  - public virtual string ToString()
- In class SortedList
  - All CopyTo() methods
  - public int Capacity { get; set; }
  - public IComparer<TKey> Comparer { get; }
  - public virtual bool Equals(Object obj)
  - protected virtual void Finalize()
  - public virtual int GetHashCode()
  - public Type GetType()
  - protected Object MemberwiseClone()
  - public virtual string ToString()
- In class SortedSet
  - All CopyTo() methods
  - protected SortedSet(SerializationInfo info, StreamingContext context)
  - public IEqualityComparer<T> Comparer { get; }

- public virtual bool Equals(Object obj)
- protected virtual void Finalize()
- public virtual int GetHashCode()
- public virtual void GetObjectData(SerializationInfo info, StreamingConte
  context)
- public Type GetType()
- protected Object MemberwiseClone()
- public virtual void OnDeserialization(Object sender)
- public virtual string ToString()

Bibliography

[1] `https://code.vis.ethz.ch/svn/mbc/Documents/Report/`. [Online; accessed 10-June-2013].

[2] N. Polikarpova, C. A. Furia, and B. Meyer, "Specifying Reusable Components," in *Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'10)*, G. T. Leavens, P. O'Hearn, and S. Rajamani, eds., vol. 6217 of *Lecture Notes in Computer Science*, pp. 127–141. Springer, August, 2010.

[3] `http://dsa.codeplex.com`. [Online; accessed 10-June-2013].

[4] `http://research.microsoft.com/en-us/projects/pex/`. [Online; accessed 10-June-2013].

[5] `http://www.red-gate.com/products/dotnet-development/reflector/`. [Online; accessed 10-June-2013].