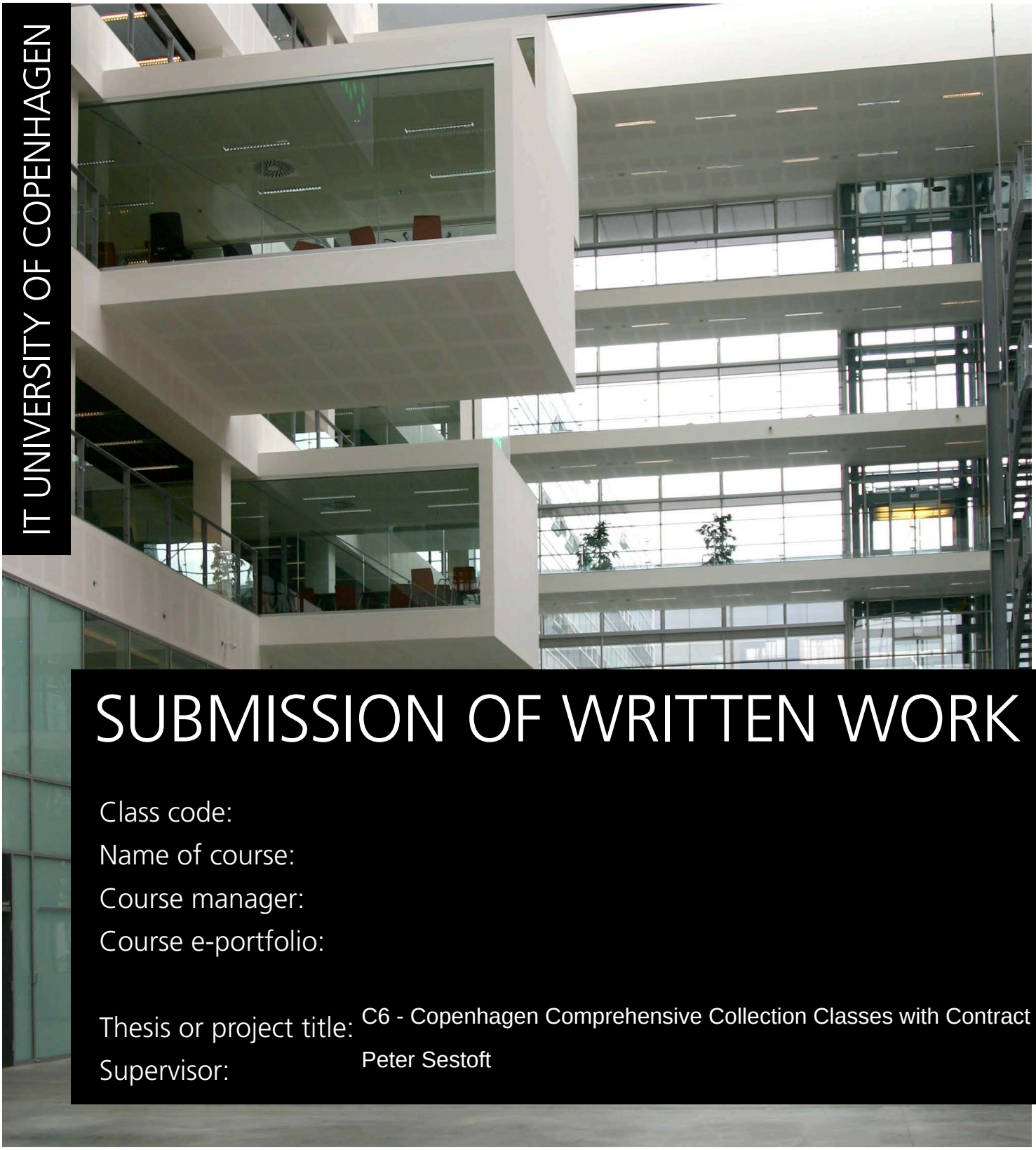
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code:

Name of course:

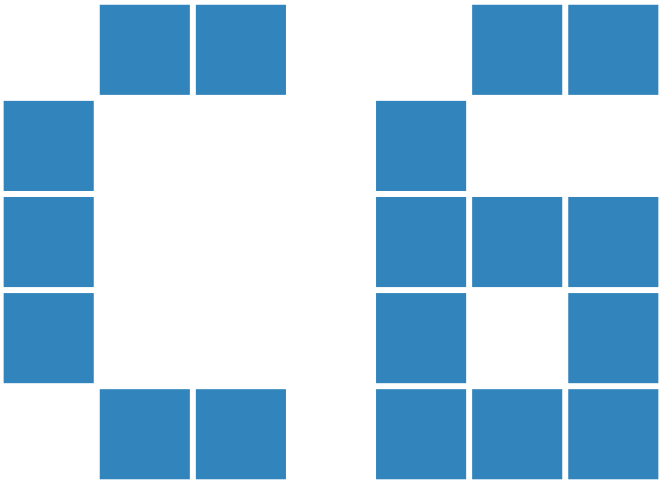
Course manager:

Course e-portfolio:

Thesis or project title: C6 - Copenhagen Comprehensive Collection Classes with Contract

Supervisor: Peter Sestoft

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Mikkel Riise Lund	06/01-1988	mirl@itu.dk
2.			@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk



Copenhagen Comprehensive Collection Classes with Contracts for C#

Mikkel Riise Lund – mirl@itu.dk
IT University of Copenhagen

Supervisor: Peter Sestoft

Spring Semester
June 1, 2016

Abstract

This thesis project examines the benefits of using Microsoft's Code Contracts framework in a large, real-life project like the C5 Collection Library.

Microsoft's Code Contracts is a modern framework that provides a language-agnostic way to express coding assumptions in .NET programs using the idea of *Design by Contract*. Using preconditions, postconditions, and object invariants, previously implicit assumptions and specifications become explicit and can be enforced using the automated tool to help increase the quality of large code projects like C5.

The C5 Collection Library aims at being a C#/CLI generic collection library whose functionality, efficiency and quality meets or exceeds what is available for similar contemporary programming platforms. As time passes, C5 itself has struggled to stay contemporary, and though it received a larger update in 2011, the library has not been actively developed for many years.

To help C5 come up to date, this project updates the library to better reflect the current state of the .NET Framework and to take advantage of the many new additions to the C# language. The library is ultimately updated to pave the way for Code Contracts, and though the overall goal is to add contracts to the whole library, this project focuses on the well-known data structure `ArrayList<T>` and the relevant parts of C5. The comprehensive upgrade shall result in a new collection library called C6 for Copenhagen Comprehensive Collection Classes *with Contracts* for C#.

The project furthermore surveys different tools that use Code Contracts, such as Code Contracts' own static checker and Visual Studio's IntelliTest, to further explore the capabilities of the contract framework and to see whether the tools can help the development of a large project like C6.

Contents

Chapter 1

Introduction

I often hear people saying that interfaces specify contracts. I believe this is a dangerous myth. Interfaces, by themselves, do not specify much beyond the syntax required to use an object. [...] Interfaces separate syntax from implementation [...] In reality, the contract is semantics, and these can actually be nicely expressed with some implementation.

Krzysztof Cwalina [11, section 4.3]

An interface describes the syntax used when communicating with a class. An interface for a collection class might have a `Count` property that returns an integer and an `Add()` method that takes a generic item. From the member names and documentation, we might be able to tell that `Count` is the number of items in the collection, and `Add()` will add the item to the collection. From this we get that an empty collection has a `Count` of zero, and that `Count` increments by one, when we add an item to the collection. Though this is what we expect, there is really nothing that enforces this behavior! The member names and documentation of the interface might clearly state the expected behaviour, but apart from unit tests, we cannot be sure that the class actually works as expected.

Code contracts allow us to express the expected behaviour from any class that implements an interface. Instead of just expressing the behavior in words, we can express it in a way that the computer understands and enforces. A contract could for instance say that if an item is added to the collection, its `Count` must increment by one; if the count is not incremented, the contract is violated. This requires no actions from the developer implementing the interface: as long as the contract is enabled, the check will be performed whenever the method is called. The code contract thereby ensures that the intended behaviour is also the actual behaviour.

The C5 Generic Collection Library (C5) [32] is a powerful, well-structured and scalable generic collection library for C#/CLI. The library provides functionality and data structures not provided by the standard .NET `System.Collections.Generic` namespace. C5 is now more than thirteen years old, and though it received a larger update in 2011 [102], it has not been actively developed for many years; meanwhile, the .NET Framework and the C# programming language have received several large updates. If the library is to remain relevant in the future, it requires an update. One way to improve and update C5 is to incorporate Code Contracts into the library's interface hierarchy and its vast number of implementation classes, and this is exactly what this project will focus on.

1.1 Reading Guide

This report is addressed to any person with an interest in Code Contracts and the C5 collection library. General programming experience as well as knowledge of the C# pro-

gramming language, basic data structures and the C5 library is assumed.

The report is divided into the following chapters:

Chapter 1 – Introduction provides an overall presentation of the project along with its goals and scope.

Chapter 2 – Microsoft's Code Contracts introduces the Code Contracts framework and discusses its advantages and disadvantages.

Chapter 3 – From C5 to C6 describes the key differences between the C5 and C6 collection libraries and considers what was added, changed or removed in C6.

Chapter 4 – Code Contracts in C6 explains how Code Contracts were added to C6. This includes an overview of the contract helper methods, the interface contracts and the many challenges encountered.

Chapter 5 – Unit Testing in C6 describes how unit testing was done in C6.

Chapter 6 – Implementing ArrayList in C6 considers how `ArrayList<T>` was implemented in C6 and what changes were made to the data structure since C5.

Chapter 7 – Bugs in C6 lists the bugs that tests and contracts helped uncover.

Chapter 8 – Tools that Work with Contracts evaluates some of the tools that benefit from Code Contracts and discusses their value in relation to C6.

Chapter 9 – Discussion reflects on the use of Code Contracts in large, real-life projects like C5 and C6.

Chapter 10 – Conclusion finally gives a short conclusion and sums up the most important points of the project.

1.2 Project Goals

The project examines the benefits of using Code Contracts in a large, real-life project like the C5 Collection Library. In this project, I intend to:

- analyze the existing C5 library to understand how it works and how it can be updated and extended with Microsoft's Code Contracts framework.
- create a new library called *C6 - Copenhagen Comprehensive Collection Classes with Contracts for C#* that:
  - updates the C5 library.
  - follows what is now considered "good .NET (library) design" in general and according to Microsoft.
  - contains a showcase data structure.
  - adds contracts to the interfaces and classes relevant for implementing the chosen data structure.

In this report, I intend to:

- survey different tools that benefit from the presence of Code Contracts and reflect on the extent to which they are helpful.
- examine whether there are ways to write contracts that better enables tools that use or rely on Code Contracts.
- evaluate whether contracts are a worthwhile addition.
- evaluate whether Code Contracts could be improved to be more valuable for projects like C6.

During the project, I wish to become better at:

- writing code contracts in a modern object-oriented language like C#.
- writing thorough and covering unit tests using a modern unit test framework (like NUnit and xUnit.net).
- writing proper documentation that assists developers using the collection library.

1.3 Scope

The project focuses on extending a single data structure and the interfaces it uses in C5 with Code Contracts to see whether it improves development of the library. C5's `ArrayList<T>` was chosen, as this is a well-known data structure that touches many of the library's interfaces and allows for a bare-bone version, which can be extended with advanced features like events and views, if time allows. Events were implemented, but adding views would require a lot more testing, because they implement (almost) all the same functionality as a list, but in a slightly different context. Furthermore, views posed a challenge in relation to Code Contracts; however, a possible solution to this problem was found and described in section 4.3.3.

Though the long-term goal would be to add contracts to the whole library, only interfaces and classes relevant to `ArrayList<T>` have been considered, and the code (including documentation and unit tests) has only been developed with this rather narrow perspective. This means that though things are intended to work with other types of data structures such as sets and multisets, it is unknown if they actually do.

C5 and C6 are written in C# and the main focus is therefore C#. Even though many languages support code contracts, only those relevant for C# have been considered.

C5 supports compiling the library for .NET 3.5, 4.0, and 4.5 as well as a portable and as a universal library. To narrow the scope, C6 has been developed as a portable library targeting .NET 4.5, Silverlight 5, Windows 8, Windows Phone 8.1 and Windows Phone Silverlight 8. It is, however, the intent that C6 should support the same versions as C5 currently does, or maybe use the new .NET Core framework [17].

The project contains no performance tests. None of the changes made in the project are expected to affect performance negatively, but there is currently no proof that this is in fact the case.

1.4 Accompanying Files

This project is accompanied by the following files:

- The C6 repository [34] with the Visual Studio solution consisting of:
  - **C6:** the library project which contains interfaces, data structures, etc.
  - **C6.Tests:** the testing project for the library project.
  - **C6.UserGuideExamples:** a small project containing a usage example of `ArrayList<T>`.
- The HTML documentation generated from the source code by Doxygen [138].

1.5 Namespace Shortening Convention

Throughout the report the abbreviation `SCG` is used for the `System.Collections.Generic` namespace (the CLI class library namespace for generic collection classes) to avoid ambiguities when class and interface names in system collection library and C5/C6 collide. Likewise, `SC` is occasionally used for the `System.Collections` namespace of non-generic collection classes. Both namespaces are standardized as part of ECMA CLI [25].

1.6 Special Thanks

Special thanks go to my project advisor Professor Peter Sestoft for the overall guidance through this extensive project, to Ben Vautier who helped come up with the initial idea, to C5 maintainer Rasmus Lystrøm for some great discussions on the C5 and C6 libraries, to Nicolai Dahl, Tomas Lieberkind, Morten Hyllekilde Andersen, Jonas Busk, Matthias Haamann, and Henrik Lund who all helped review and proofread the report, and finally to my girlfriend Mille Israelsen who has helped me with everything but the project. I am very grateful for all your help.

Chapter 2

Microsoft's Code Contracts

This chapter introduces the concept of Design by Contract by looking at Microsoft's Code Contracts framework and its predecessor Spec#. This includes a quick introduction to the framework and a discussion of its advantages and disadvantages along with some quick tips about readability and pitfalls.

2.1 Design by Contract

A typical problem in software development today is that numerous assumptions made by the developers are never verified (systematically), as they are left either unspecified, informal, as natural-language documentation, or as some standardized programming interface [3]. This makes it very easy to break these assumptions, either during maintenance or further development. We normally try to verify these assumptions using tests, but ensuring consistent checking can be difficult.

Design by Contract (DbC) is an approach to software development, where interface specifications are defined in a formal way using preconditions, postconditions and invariants, that can be verified either at runtime or at compile time using static analysis [139]. DbC allows developers to precisely express the previously hidden intents using contracts, and the contracts can furthermore be used to automatically generate documentation. The term *Design by Contract* was introduced in 1986 [53][54] by Bertrand Meyer in connection to his work on the programming language Eiffel [125] and was later trademarked by Eiffel Software [140]. Few languages currently support Design by Contract natively/directly, but some of the more well-known languages include Ada 2012 [7], Clojure [23], Eiffel [125] and Spec# [120].

Code contracts consist of three parts: preconditions, postconditions, and invariants. Preconditions specify the requirements when a member is entered. These can be requirements on the parameters or the state of the called object. Postconditions specify the requirements after the member has been called. These conditions can either be on the returned values or the called object's state. Object invariants specify the requirements for an object whenever it is visible to a client. The object invariants express what is considered a valid state for the object in-between method calls.

2.2 Spec#

C# normally has no language support for code contracts. Spec# was an early attempt by Microsoft Research to implement it. The researchers wished to provide the complete infrastructure for using contracts, including libraries, tools, design support, and integrated editing capabilities that provided both short- and long-term benefits [3, chapter 0]. The

Spec# programming language, which is a superset of C#, adds features like non-null types `T!`, *checked* and *unchecked* exceptions, and pre- and postconditions. The example in listing 2.1 (originally from [3, section 1.1]) shows pre- and postconditions written in Spec# on the non-generic class `System.Collections.ArrayList`'s `Insert()` method:

```
public virtual void Insert(int index, object value)
    requires !IsReadOnly && !IsFixedSize;
    requires 0 <= index && index <= Count otherwise ArgumentOutOfRangeException;
    ensures Count == old(Count) + 1;
    ensures value == this[index];
    ensures Forall{int i in 0 : index; old(this[i]) == this[i]};
    ensures Forall{int i in index : old(Count); old(this[i]) == this[i + 1]};
{
    // Method implementation...
}
```

Listing 2.1: Code contracts on `SC.ArrayList.Insert()` *written in Spec#.*

Notice how the contracts express what must be true; this is slightly different from normal *if-then-throw* statements which are *negated* preconditions:

```
if (IsReadOnly || IsFixedSize) {
    throw new Exception();
}
if (index < 0 || Count < index) {
    throw new ArgumentOutOfRangeException();
}
```

Listing 2.2: The preconditions from listing 2.1 as normal if-then-throw statements.

The conditions are written after the method signature and are checked at runtime. If violated, preconditions will throw a `RequiresViolationException` (explicit exception types can also be specified as in listing 2.1 with `ArgumentOutOfRangeException`) and postconditions will throw an `EnsuresViolationException`. Furthermore, Spec# provides a static program verifier called Boogie, which will attempt to verify the preconditions at compile time.

2.3 Code Contracts – The Framework

The last stable release of Spec# was released in 2011, but many of the ideas of Spec# have since made it into the newer project Code Contracts, also by Microsoft Research. Instead of being a superset of C#, Code Contracts is implemented as a framework that works for all languages on the .NET platform. Code Contracts first became available in C# 4.0 with the release of Visual Studio 2010, where the contracts classes – residing in the `System.Diagnostics.Contracts` namespace [73] – are found in `mscorlib`. For C# 3.5 and Visual Studio 2008, Code Contracts is available as a separate library `Microsoft.Contracts`, which must be referenced explicitly [24].

The project has since January 2015 been open-sourced on GitHub [67]. Though the project currently evolves rather slowly, the first community-driven (pre)release was published in January 2016 [66], about a year after the open-source community took over. This is the version of Code Contracts used in this project.

2.4 Quick Guide

This section gives an overview of how Microsoft's Code Contracts works. Readers already familiar with Code Contracts may simply skip it. For further detail see [56], [67] and [68].

Code Contracts uses the static class `System.Diagnostics.Contracts.Contract` to specify contracts. Pre- *and* postconditions are stated at the beginning of a method. Preconditions are stated using `Contract.Requires()` and are checked just before a method is entered. Preconditions most often validate parameters, and can only access what is at least as visible as the method itself. `Contract.Requires<TException>()` is used to specify the type `TException` of the thrown exception. Postconditions are checked just prior to exiting a method using `Contract.Ensures()`. They express what must hold on normal termination. Postconditions have no visibility restriction on the variables used.

Listing 2.3 contains the Code Contracts equivalent of the example in listing 2.1:

```
1  public virtual void Insert(int index, object value) {
2      Contract.Requires(!IsReadOnly && !IsFixedSize);
3      Contract.Requires<ArgumentOutOfRangeException>(0 <= index && index <= Count);
4
5      Contract.Ensures(Count == Contract.OldValue(Count) + 1);
6      Contract.Ensures(value.Equals(this[index]));
7      Contract.Ensures(Contract.ForAll(0, index,
           i => Contract.OldValue(this[i]).Equals(this[i])));
8      Contract.Ensures(Contract.ForAll(index, Contract.OldValue(Count),
           i => Contract.OldValue(this[i]).Equals(this[i + 1])));
9
10     // Method implementation...
11 }
```

Listing 2.3: The same contracts as in listing 2.1, but written in C# using Code Contracts.

Code Contracts provides special helper methods that can only be used within postconditions. The return value of method is provided by `Contract.Result<T>()`, where `T` is the return type. `Contract.OldValue<T>(e)`, where `T` is the type of the expression `e`, refers to the value of `e` in the method's pre-state. The value of `ref` and `out` parameters at method return are given by `Contract.ValueAtReturn<T>(out x)`, where `T` is the type of `x`. The generic type of both `OldValue()` and `ValueAtReturn()` can be inferred by the compiler. `Contract.EnsuresOnThrow<T>()` expresses what must hold when an exception of type `T` escapes the method:

```
1  Contract.EnsuresOnThrow<Exception>(Contract.ForAll(0, Count,
       i => Contract.OldValue(this[i]).Equals(this[i])));
```

Listing 2.4: The contract must hold, when an `Exception` escapes the method.

Object invariants are declared using `Contract.Invariant()` within a private nullary instance method that returns void, marked with the attribute `ContractInvariantMethod`:

```
1  [ContractInvariantMethod]
2  private void ObjectInvariant() {
3      Contract.Invariant(x != null);
4      Contract.Invariant(Count >= 0);
5  }
```

Listing 2.5: `ContractInvariantMethod` marks the method containing the object invariants.

The invariants are checked at the end of the outermost public method call to a class.

`Contract.Assert()` can be used within the code to state a condition that must hold true at that point. `Contract.Assume()` works similarly, but is used to help the static checker, when it cannot prove an assertion by itself.

Universal quantifications are written using the overloaded `Contract.ForAll()`, which either takes an `IEnumerable<T>` and a predicate that takes an item of type `T`:

```
1 Contract.Requires(Contract.ForAll<T>(enumerable, x => x != null));
```

Listing 2.6: The overloaded universal quantifier that takes an enumerable and a predicate.

Or an *inclusive* lower bound, an *exclusive* upper bound, and an `int`-to-`bool` predicate:

```
1 Contract.Ensures(Contract.ForAll(0, Contract.Result<int[]>().Length,
      i => Contract.Result<int[]>()[i] > 0));
```

Listing 2.7: The overloaded universal quantifier that takes two bounds and a predicate.

`Contract.ForAll()` returns `true`, only if the predicate holds for all elements. Existential quantifications use `Contract.Exists()`, which comes in the same two overloaded versions as `Contract.ForAll()`. `Contract.Exists()` returns `true`, if the predicate holds at least once. The quantifiers terminate as soon as an answer is known and can be used within all contracts and even with `Contract.OldValue()`. The quantifiers can easily be assisted by LINQ [92].

Contracts on C# interfaces and abstract methods are written using a contract class, as seen in listing 2.8. The contract class (annotated with `ContractClassFor()`) must be abstract and implement the interface or inherit from the class with the abstract method(s), which is annotated with `ContractClass()`. The contract methods can either return a default dummy value or throw an exception, e.g. a `NotImplementedException`.

```
1  [ContractClass(typeof(IFooContract))]
2  interface IFoo {
3      int Count { get; }
4  }
5
6  [ContractClassFor(typeof(IFoo))]
7  abstract class IFooContract : IFoo {
8      public int Count {
9          get {
10             Contract.Ensures(Contract.Result<int>() >= 0);
11             return default(int); // ignored dummy return value
12         }
13     }
14 }
```

Listing 2.8: `IFooContract` contains the contracts for `IFoo`, which every implementation of `IFoo` inherits.

All methods used in Code Contracts must be without any visible side effects to the caller. Code Contracts allows developers to annotate side-effect-free methods with `Pure`, and the compiler gives a warning, if non-`Pure` methods are used within contracts (it is currently not verified whether the methods are indeed pure).

All the contract methods have an overload, that accepts an additional user-defined string parameter, displayed at runtime if the contract fails. A failing contract throws

a `ContractException` declared in the `System.Diagnostics.Contracts` namespace. The exception class is private to ensure that it cannot be caught (explicitly) by client code.

It is possible to build different versions of a project with Code Contracts. The runtime checking level can be changed to best suit the situation. Normally, a debug configuration will check all contracts (checking level *Full*), whereas a release configuration might only contain preconditions that throw a specified exception (checking level *ReleaseRequires*). It is also possible to build a reference assembly that contains the code contracts, so they can be used when referencing the assembly.

2.5 For and Against Code Contracts

2.5.1 Advantages

The single biggest advantage of Code Contracts is the runtime checks. Always checking the developer's assumptions when running the code helps uncover bugs much sooner, because (well-written) contracts will notify the developer as soon as an invalid state is encountered. This also makes Code Contracts invaluable when it comes to complicated invariants that are not easily checked with unit tests, e.g. when a self-balancing tree contains a bug that causes improper balance in subtrees.

Code Contracts is a great addition to unit tests, because the contracts can give each test a much larger reach. A unit test might only test a single aspect of the system under test, but the code contracts can check any facet of the post-state to ensure that it is valid. If a test only checks the state *after* a compound operation, Code Contracts can help validate the state after each individual step, without requiring any extra effort from the developer. This is very different from the manual checks in C5, described in section 6.9. Contracts can also help find bugs in special cases that were not properly covered by unit tests, as demonstrated in section 7.1.4.

Code Contracts moves some of the responsibility away from the developer and over to the tool instead. A good example of this is parameter validation on an interface implementation: normally the developer is responsible for always checking that the implementation has the necessary parameter validation; by moving the checks to the interface contract, the developer can implement the parameter validation once for the interface and not worry about it in the implementing classes afterwards. This can also shorten the code considerably, as seen in listings 6.3 and 6.4.

Test classes normally do not have access to the private members of the system under test. This can make them harder to test without changing their visibility. Instead of testing the functionality indirectly, postconditions can ensure that the result is always as expected. This is extremely beneficial with more complicated private methods like binary searches.

Code Contracts allows us to work with tools like Visual Studio's IntelliTest, which can help us find bugs by auto-generating unit tests. Without contracts, IntelliTest is mostly useful for legacy code that needs refactoring. Section 8.2 contains a detailed look at IntelliTest used together with Code Contracts.

2.5.2 Disadvantages

There are some downsides to having Code Contracts implemented as a framework. The lack of language support means that adding contracts is not always done in the most elegant way. Try comparing the two examples in listings 2.1 and 2.3. Apart from Code Contracts being much more verbose, Code Contracts places contracts *inside* the method body alongside the implementation. This also becomes visible when working with interfaces, where a separate contract class is needed, because interfaces cannot contain the required code. Furthermore, the interfaces and contract classes must be decorated with the `ContractClassAttribute` and the `ContractClassForAttribute` respectively to ensure that they correctly reference each other. Though a contract class can easily be stored in the same file as the interface, the contracts are moved away from the interface, thus making them more difficult to find (this does actually have the benefit of separating syntax and contracts and increases the readability of both). Missing implementation bodies also become a problem for auto-implemented properties. The contracts must therefore be implemented as invariants, which slightly delays the checks.

Preconditions and postconditions are stated at the beginning of a method. Though Microsoft recommends a particular order of the contracts [56, section 5.3], the framework does not enforce it, which means that the programmers themselves are responsible for keeping method calls ordered. This also means that it requires more effort to distinguish pre- and postconditions when skimming through the code. Section 2.6 presents different ways to help overcome readability problems.

Execution of the contracts end as soon as the first contract is violated. As with multiple assertions in a single unit test (see section 5.1.3.3), this obscures part of the picture when something goes wrong, because it is unknown whether the remaining contracts are violated.

Code Contracts inevitably extends compile times for most solutions. This is expected as the rewriter has to manipulate the compiled code to add contract checks wherever necessary. Code Contracts' rewriter `ccrewriter` manipulates the compiled code after compilation, which can in some cases [16] interfere with other tools like Fody [15]. Furthermore, the performance is significantly hit especially when using quantifiers in contracts. However, this is normally only a problem in postconditions and object invariants, which should be disabled in production code anyway. One can further argue that a bug-free program will never break a precondition and that preconditions therefore also can be disabled in production to increase performance further.

It currently seems that the biggest disadvantage of Code Contracts is its somewhat fragile future. This is discussed in section 9.2.

2.6 Improving Readability and Usability

There are many small steps that can be taken to improve the readability and usability of Code Contracts. This section lists those with the biggest impact.

2.6.1 Static Import

C# 6.0 introduces statically import classes that allow static members to be called without a type name. It is useful to statically import Code Contracts' `Contract` class:

```
1  using static System.Diagnostics.Contracts.Contract; // Placed with the other imports
```

Listing 2.9: A static import of the `Contract` class in C# 6.0.

All methods on `Contract` can now remove the class name to shorten the code considerably:

```
1  // Without static import
2  Contract.Ensures(Count == Contract.OldValue(Count) + (Contract.Result<bool>() ? 1 : 0));
3
4  // With static import
5  Ensures(Count == OldValue(Count) + (Result<bool>() ? 1 : 0));
```

Listing 2.10: Comparison of contracts without and with a statically imported `Contract` class.

The rest of this report assumes that `Contract` was statically imported, which is also the case in all of C6.

2.6.2 Wrapping Contracts in Regions

Wrapping code contracts in a *Code Contracts* region can greatly improve readability in editors that support code regions [101]. Contracts have a tendency to clutter up the view when they are declared at the beginning of a method, as seen in listing 2.11, which contains a slightly simplified version of one of `C6.ArrayList<T>`'s constructors. The actual implementation is only the last four lines of code, whereas the remaining body is contracts. This can greatly lower readability of the code, depending on the number of contracts.

```
1  public ArrayList(IEnumerable<T> items, IEqualityComparer<T> equalityComparer = null,
       bool allowsNull = false)
2  {
3      // Argument must be non-null
4      Requires(items != null, ArgumentMustBeNonNull);
5
6      // All items must be non-null if collection disallows null values
7      Requires(allowsNull || ForAll(items, item => item != null), ItemsMustBeNonNull);
8
9      // Value types cannot be null
10     Requires(!typeof(T).IsValueType || !allowsNull, AllowsNullMustBeFalseForValueTypes);
11
12
13     // The specified enumerable is not equal to the array saved
14     Ensures(!ReferenceEquals(items, _items));
15
16     // The count is equal to the number of items
17     Ensures(Count == items.Count());
18
19
20     EqualityComparer = equalityComparer ?? EqualityComparer<T>.Default;
21     AllowsNull = allowsNull;
22     _items = items.ToArray();
23     Count = Capacity;
24 }
```

Listing 2.11: Code contracts on a `ArrayList<T>` constructor without a Code Contracts region.

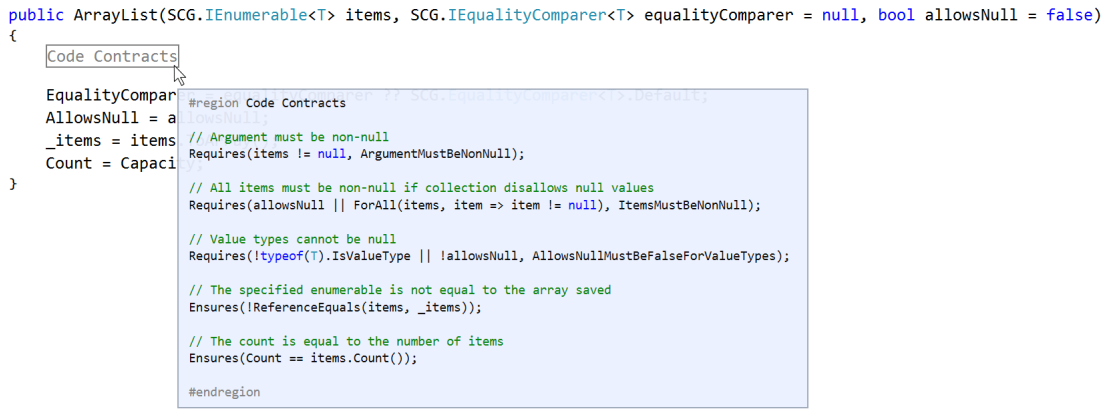


Figure 2.1: The same constructor as showed in listing 2.11. The contracts are wrapped in a region and displayed either by expanding the region, or hovering the mouse over the collapsed region's name.

Visual Studio can expand or collapse the code region, so only the implementation code becomes visible, as seen in figure 2.1. The contracts can be read either by expanding the region, or by simply hovering the mouse over the region name.

This is not a problem for inherited members as their contracts normally reside in another (contract) class. Nor is it a problem for object invariants, as they only contain contracts and require no further means of separation from the remaining code.

2.6.3 Splitting Conditions to Help Debugging

A simple index bound check can be done as a one-liner:

```
1 // Argument must be within bounds
2 Requires(0 <= index && index < Count);
```

Listing 2.12: A code contract that contains two different checks.

It does, however, help debugging to split the checks into two preconditions:

```
1 // Argument must be within bounds
2 Requires(0 <= index);
3 Requires(index < Count);
```

Listing 2.13: The same contracts as in listing 2.12, but split into two separate contracts.

If the joined preconditions fail, it is difficult to know which one failed without debugging the code. When split, only one of them will fail, revealing the problem immediately.

2.6.4 Contract Ordering and Spacing

No contract ordering is required by Code Contracts, but Microsoft recommends a certain ordering [56, section 5.3], as listed in table 2.1. For interfaces only the public contracts are relevant, but for classes all the contracts may be used.

Leaving space between the different types of contracts can help group them visually, as seen in listing 2.11.

`Requires, Require<E>`	All public preconditions
`Ensures`	All public (normal) postconditions
`EnsuresOnThrow`	All public exceptional postconditions
`Ensures`	All private/internal (normal) postconditions
`EnsuresOnThrow`	All private/internal exceptional postconditions

Table 2.1: Microsoft's recommended order for non-legacy code contracts.

2.6.5 Comments and User Messages

Code contracts are normally written as Boolean one-liners, though helper methods can be used (see section 4.1). When the statement increases in length, readability often goes down. A simple comment explaining the intent of the contract can help a lot:

```
1 // Successfully adding an item increases the count by one
2 Ensures(Count == OldValue(Count) + (Result<bool>() ? 1 : 0));
```

Listing 2.14: A comment explaining the intent of the code contract.

This also helps ensure that the original idea gets communicated to whomever reviews it. Alternatively, the comment can be supplied as the user-specified message:

```
1 Ensures(Count == OldValue(Count) + (Result<bool>() ? 1 : 0),
      "Adding an item increases the count by one");
```

Listing 2.15: A code contract with a user-specified error message.

This has the advantage of providing help to the client when a contract fails, but the message will also be included in the automatically generated documentation. Furthermore, the comments can be used with unit testing to ensure a test violates the expected precondition, as discussed in section 5.4.1.1.

A comment can also improve readability, when pre- or postconditions are missing:

```
1  public Speed CountSpeed {
2      get {
3          // No preconditions
4
5
6          // Result is a valid enum constant
7          Ensures(Enum.IsDefined(typeof(Speed), Result<Speed>()));
8
9
10         return default(Speed);
11     }
12 }
```

Listing 2.16: A comment helping readability of a member without preconditions.

Without the first comment, the reader has to check whether the first contract is a pre- or postcondition. The comment helps visually create the grouping.

2.7 Pitfalls

As with most large pieces of software, Code Contracts has some pitfalls, popularly called *gotchas*: some valid constructs that work as intended, but are likely to cause mistakes [141]. Here are some of the most typical ones, all of which the author have at some point in time fallen victim of.

2.7.1 Old Value of Reference Types

In postconditions, we use `Contract.OldValue()` to refer to the value of an expression from the pre-state, i.e. at the time the method is called. For instance:

```
1 // Count goes up by one
2 Ensures(Count == OldValue(Count) + 1);
```

Listing 2.17: A postcondition that uses the old value of simple type.

The expression in `OldValue()` is evaluated when the method is called, and can be used once the method has terminated. A common problem occurs when referring to the pre-state value of a reference types:

```
1 // The collection does not change
2 Ensures(this.SequenceEqual(OldValue(this), EqualityComparer));
```

Listing 2.18: A postcondition that uses the old but unchanged reference in `this`.

Here we check whether two enumerables are equal before and after the method call. The problem is that this always evaluates to `true` (provided that the enumerable and equality comparer are deterministic). `OldValue(this)` will refer to the old value of the reference — *not* the old value of the enumerable! We must therefore reference something that does not change during the method call. We can do this by retrieving the values from the enumerable and storing them in a list, which can then be referenced:

```
1 // The collection does not change
2 Ensures(this.SequenceEqual(OldValue(this.ToList()), EqualityComparer));
```

Listing 2.19: A postcondition that uses the old values in `this` by saving them to a list first.

2.7.2 Code Contracts Executes Code

It might seem trivial, but can easily be forgotten: Code Contracts executes your code. This can cause lazily-evaluated constructs to suddenly seem eager, or result in head-scratching bugs that only show when contracts are enabled. The latter is demonstrated in section 7.2.2, where a bug in a contract helper caused a shuffle method to always shuffle list items in a certain order.

2.7.3 Dependencies

Code Contracts adds no dependencies in C# 4.0 or later, as it is already included in `mscorlib`. This means that Visual Studio 2010 or later will be able to build projects like C6 without having Code Contracts installed. The contracts are, however, not added to the library until *Microsoft Code Contracts (devlab_ TS) for .NET* [69] is installed.

2.7.4 Contract Inheritance for Members with Multiple Roots

One of the greatest features of Code Contracts is contract inheritance. Contracts on parent classes or interfaces will automatically be inherited by any child or implementing class. The problem is that inheritance can be extremely difficult to predict, when a member has multiple roots. According to the Code Contracts User Manual [56, chapter 3], the framework does not allow inherited preconditions to be strengthened, but the manual explicitly allows postconditions to be. In reality, the framework behaves differently.

The preconditions are not necessarily inherited when an interface or class redeclares a member with the `new` keyword. If an implementing class only implements the member once, the preconditions are inherited, but the compiler issues a warning. If the inherited member is explicitly implemented, the two methods suddenly each have their own contracts and nothing is inherited. Calling one implementation from the other will furthermore throw contract exceptions that cannot be intercepted at runtime with call-site requires as used in section 5.4.1. Inheritance for members with multiple roots is further discussed in section 4.3.2 in the context of C6.

Though postconditions allegedly can be strengthen, this does not always seem to be enforced. It works if the postconditions are declared in the implementation, but not always if the postconditions are on a new interface. The example in listing A.1 [41] shows the subtle differences that can cause postconditions to be used or ignored.

To me it can still sometimes feel unintuitive when postconditions are actually checked.

2.8 Advice on Code Contracts

2.8.1 Implications

It is often necessary to write contracts with implications: $p \rightarrow q$. If p is true, then q is too. However, neither C# nor Code Contracts support writing implications directly, so a slight rewrite is necessary:

```
1 // If the collection contains the item, then the result is non-negative
2 Ensures(!Contains(item) || Result<int>() >= 0); // Contains(item) -> Result<int>() >= 0
```

Listing 2.20: A contract containing a rewritten implication.

We might be tempted to use an extension method to make contracts more readable, but as explained in appendix B, this tends to lower the readability and is therefore not used in C6.

2.8.2 Cyclic Calls

One trick with Code Contracts is to use other members to write better or more precise contracts. This does, however, make it possible to have cyclic calls, either within the contracts themselves or in an actual implementation. Consider `IIndexed<T>`'s `IndexOf()`, which has the following contract:

```
1 // Result is a valid index
2 Ensures(Contains(item)
3     ? 0 <= Result<int>() && Result<int>() < Count
4     : 0 <= ~Result<int>() && ~Result<int>() <= Count);
```

Listing 2.21: This postcondition on `IndexOf()` *uses* `Contains()` *to bound the returned index.*

If the collection contains the item, the returned index is within bounds; otherwise, the one-complement of the index is. The contract uses `ICollection<T>`'s `Contains()` to decide whether the result should be negative or not. `Contains()` is, however, implemented using `IndexOf()` in `ArrayList<T>`:

```
1 public bool Contains(T item) => IndexOf(item) >= 0;
```

Listing 2.22: The implementation of `Contains()` *in* `C6.ArrayList<T>` *uses* `IndexOf()` *internally.*

This will again trigger the contract on `IndexOf()`, which again calls `Contains()`, and so on. It would be impossible to ensure that contracts never made cyclic calls, and luckily Code Contracts handles this brilliantly by only making a fixed number of cyclic calls before bypassing the contracts and directly calling the implementation. Contracts are therefore not ignored, but only verified until a certain cyclic depth.

2.8.3 Violating Contracts

Violated contracts are not a rare sight during normal development, but there are many different reasons as to why a contract might fail. Violated contracts might be expected and even wanted, but most often they indicate an error in the program.

A violated precondition is typically caused by invalid input, where the pre-state requirements are not fulfilled by the caller. This is the kind of violation that should never occur in production, as it indicates a problem in the calling code. Yet during testing, we deliberately violate our preconditions to ensure that they correctly catch invalid input or that they are indeed enabled, when expected. Section 5.4.1.1 discusses this further.

A situation where both postconditions and invariants get violated is when an implementation does not fulfil the specification. Here a violated contract is a good thing, as it helps catch erroneous or missing behavior, which may otherwise not be easily detectable. Before implementing a member, we should always verify that the postconditions are in fact violated, since a non-violated postcondition could indicate an error in the postcondition or possibly elsewhere.

When invariants are violated, it is often because they have been too strictly specified. We might think that we can assume certain things, but in reality corner cases often require invariants to either be loosened, tweaked or completely removed, simply because they are not true. Classes that use inner classes, for instance trees or linked lists, might even have

to move invariants on the inner classes to the outer class. During a call to the outer class, the inner class' invariant might temporarily be violated, only to once again hold true when the outer call returns. This is for instance often the case during tree rotations.

Finally, a contract might fail if it asserts the wrong thing or – even worse – not fail because it asserts the wrong thing. This is normally caused by copy-pasting and is best discovered using the tests suggested above.

2.8.4 Core Library Contracts

It can be interesting to look at the code contracts written for the core library. The original core library does not contain any contracts – as they were introduced much later than the library itself – but the Code Contracts project [67] contains contract classes for the core library that can be used with the rewriter. The contract class for `SCG.ICollection<T>` [57] is seen in listing 2.23 and the contract class for `SCG.IList<T>` [58] in listing 2.24. Both interfaces are used in C6.

Surprisingly enough, many of the contracts found in the Code Contracts project are rather vague. Instead of giving a precise value of `Count` after adding an item, the post-condition on `Add()` only ensures that `Count` does not decrease. The same is true for `Remove()`, where the contracts only ensure that `Count` is not increased. Both values could be expressed exactly using the old value of `Count`.

The contracts on `IndexOf()` (starting at line 17 in listing 2.24) is also vague. They do not state that -1 is the return value used when an item is not found, and neither do they use `ICollection<T>`'s `Contains()` method as seen in listing 2.21. The preconditions even interfere with the contracts in C6, as explained in section 4.3.2.

The contract classes also seem to have a low internal consistency. The methods on the two abstract classes are implemented differently: implicitly in `ICollectionContract<T>`, but explicitly in `IListContract<T>`. Return values also differ: sometimes `default()` is used, other times a `NotImplementedException` is thrown (line 19 in listing 2.24). I am not aware if either of these choices should have any practical implications for the contracts, but I find it a shame that Code Contracts has not used a particular style throughout the contract classes. The contract classes have, nevertheless, served as inspiration for C6, and the contracts have been either directly copied or modified to more precisely state what is actually expected to be true before and after calling the interfaces.

```
[ContractClassFor(typeof(ICollection<>))]
abstract class ICollectionContract<T> : ICollection<T>
{
    public int Count {
        get {
            Ensures(Result<int>() >= 0);
            return default(int);
        }
    }

    public bool IsReadOnly { get { return default(bool); } }

    public void Add(T item) {
        Requires(!IsReadOnly);
        Ensures(Contains(item));
        Ensures(Count >= OldValue(Count));
        return;
    }

    public void Clear() {
        Requires(!IsReadOnly);
        Ensures(Count == 0);
        return;
    }

    public bool Contains(T item) {
        Ensures(!Result<bool>() || Count > 0);
        return default(bool);
    }

    public void CopyTo(T[] array, int arrayIndex) {
        Requires(array != null);
        Requires(arrayIndex >= 0);
        Requires(arrayIndex <= array.Length - Count);
        return;
    }

    public bool Remove(T item) {
        Requires(!IsReadOnly);
        Ensures(Count <= OldValue(Count));
        Ensures(!Result<bool>() || Count >= OldValue(Count - 1));
        return default(bool);
    }

    // Inherited members...
}
```

Listing 2.23: The contract class for SCG.ICollection<T> *found in the Code Contracts solution.*

```
[ContractClassFor(typeof(IList<>))]
abstract class IListContract<T> : IList<T> {
    T IList<T>.this[int index] {
        get {
            Contract.Requires(index >= 0);
            Contract.Requires(index < this.Count);
            return default(T);
        }
        set {
            Contract.Requires(index >= 0);
            Contract.Requires(index < this.Count);
        }
    }

    [Pure]
    int IList<T>.IndexOf(T item) {
        Contract.Ensures(Contract.Result<int>() >= -1);
        Contract.Ensures(Contract.Result<int>() < this.Count);
        throw new NotImplementedException();
    }

    void IList<T>.Insert(int index, T item) {
        Contract.Requires(index >= 0);
        Contract.Requires(index <= this.Count);
    }

    void IList<T>.RemoveAt(int index) {
        Contract.Requires(index >= 0);
        Contract.Requires(index < this.Count);
        Contract.Ensures(this.Count == Contract.OldValue(this.Count) - 1);
    }

    // Inherited members...
}
```

Listing 2.24: The contract class for SCG.IList<T> *found in the Code Contracts solution.*

Chapter 3

From C5 to C6

While the overall ideas of C5 have been maintained, many details have changed in order to make C6 work with Code Contracts. The following pages give an overview of what changed from C5 to C6.

It bears mentioning that C5 is more than a decade old and simply predates many of the current conventions and tools. Many of the updates in C6 are therefore not a sign of bad design in C5, but rather an indication that the surrounding environment has continued developing since C5 was written.

3.1 Design Goals of the C5 Collection Library

Though this report expects some knowledge of the C5 Collection Library, it is worth listing some of C5's design goals, before looking into the changes. The C5 Collection Library [32] aims at being a C#/CLI collection library whose functionality, efficiency and quality meets or exceeds that available for similar, contemporary programming platforms [31, section 1.2]. The library aims to be well tested and documented and to provide all the well-known data abstractions (such as lists, sets, bags, dictionaries, priority queues, (FIFO) queues and (LIFO) stacks) and the implementations thereof (array-based, linked list-based, hash-based, tree-based). The library uses existing C# patterns (like `IEnumerable<T>`, the `foreach` statement, and events), but introduces convenient but hard-to-implement features, all done in a *program-to-interface-not-implementation* style. The latter results in an extensive interface hierarchy, as seen in figure 3.1, which makes up the data abstractions. Notice that the library also contains interfaces for dictionaries not considered in this project.

3.2 C6 – The Code Project

It seems that the best *and* easiest way to upgrade C5 is to start anew. Instead of blindly adding contracts to every part of C5, the code must be handpicked, ported, and updated. This forces us to question the old design decisions before contracts are introduced.

C6 has therefore been created as a new Visual Studio solution – a new library. As with the last upgrade of C5 [102, chapter 4], C6 makes no attempt to stay backwards compatible with previous versions of C5. This is probably the best way to bring the library up to speed without compromising the quality. This also clearly separates C5's code from the the code written in this thesis project.

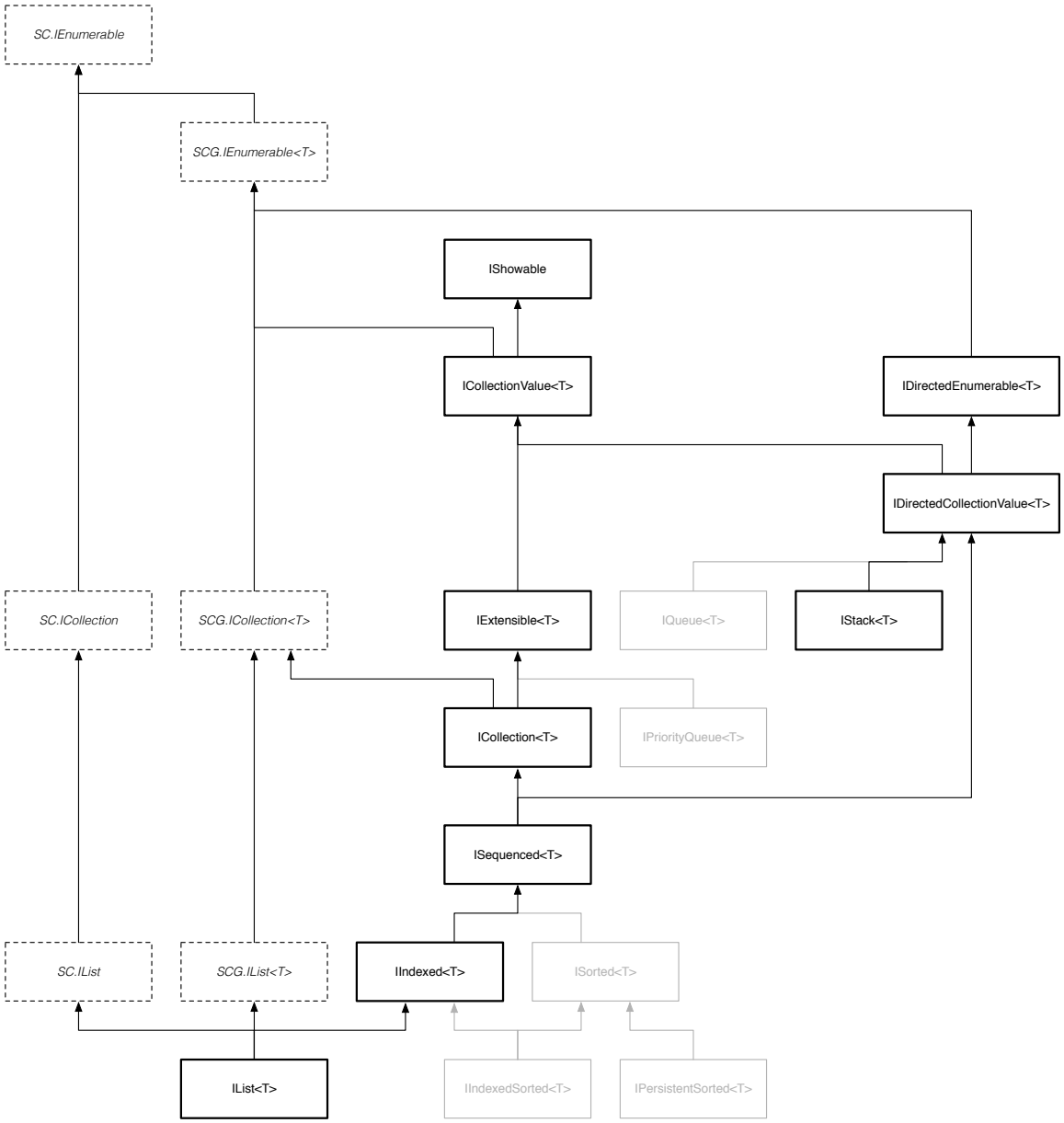


Figure 3.1: The collection interface hierarchy in C5: the thick-bordered interfaces are relevant for `ArrayList<T>`*; the dashed, italicized interfaces are from the* `System.Collections` *(SC) and* `System.Collections.Generic` *(SCG) namespaces; the faded interfaces exist in C5, but have not been relevant for this project.*

3.2.1 Namespaces

A new namespace structure was introduced in C6:

- The `C6` namespace contains the interface hierarchy along with its related classes, such as the key-value pair, enum types, and event arguments. It also contains collection extension methods that provide C5 and C6 functionality to collection types like `SCG.ICollection<T>` and `SCG.IEnumerable<T>`.
- The `C6.Contracts` namespace contains all functionality related to contracts, such as helper contract classes (section 4.1) and precondition messages (section 5.4.1.1). Notice that the interface contract classes reside in the default `C6` namespace.
- The `C6.Collections` namespace contains all the standard collection classes (currently just `ArrayList<T>`) and related helper classes, such as base classes and the `Showing` class for handling text formatting of collections (section 3.6.6).
- The `C6.Tests` namespace (used only in the test project) contains all the testing classes needed to test everything in the other namespaces.

The project folder structure matches the new namespaces, and every class resides in its own file – except for interface contract classes, which are found in the same file as the interface they describe (section 4.2).

Moving functionality into different namespaces should help only include the intended functionality. It is for instance undesirable that the contract helper methods are used for anything else than contracts. This is better prevented by requiring users to explicitly import the `C6.Contracts` namespace. The new structure also makes it possible to later include data structures that are more advanced or specialized, such as the interval data structures from C5.Intervals [52] or the collections in `System.Collections.Specialized` [86], in their own specialized namespaces.

Keeping things separated also helps, if C6 should act as a data structure development framework, as later described in section 9.3.1, where users would only need the interface hierarchy, but not necessarily the collection classes.

3.2.2 C# 6.0

C6 uses the newest version of C#, namely 6.0. Though C# 6.0 can compile to older versions of the .NET Framework, it does require that the library is compiled using Visual Studio 2015 or newer. At the time of writing, Microsoft offers a free community version of Visual Studio 2015 [70]. For a quick rundown of the new features in C# 6.0, I highly recommend AlexArchieve's *C# 6 equivalents in C# 5* [1].

3.2.3 Version Control and GitHub

Like the current version of C5, C6 uses Git [8] and GitHub [21] to manage the code base. A new GitHub organization [113] was created to host this project [34] and any future projects related to the collection library.

C6 uses the branching model presented by Vincent Driessen [13].

3.2.4 Coding Conventions and ReSharper

C6 follows the latest framework design guidelines from Microsoft [11][55]. The coding style is slightly different than recommended [11, appendix A], because the recommended style does not exactly encourage readability, neither does it consider Code Contracts nor newer additions to the language – after all, the guidelines are from 2009.

The C# coding style conventions used in C6 are enforced by ReSharper [28] using a solution-wide code formatting settings file, which is shared in the code base. This allows users with ReSharper to format whole files with a simple keyboard shortcut.

Be aware, that the code listings in this report might be formatted slightly different, in order to minimize the number of lines.

3.3 Testing and Documentation

The unit tests in C5 are as old as the library. Instead of trying to modify the existing tests to match the new coding style and behavior, a completely new test suite was developed for C6. The testing side of the project is described extensively in chapter 5.

C5 is both very well *and* very poorly documented: on one side, the C5 technical report [31] is a well-written, comprehensive description of the library and its interfaces, methods and classes; one the other side, the code (which has received several updates since the technical report was written) contains sparse and inconsistent XML documentation that at times completely contradict the actual behavior. Though this is likely to happen with big projects, that evolve over as many years as C5 has, it is an unfortunate situation, that should be addressed.

All the library documentation has therefore received an extensive update in C6. An effort was put into matching Microsoft's documentation style and language as much as possible using the standard XML documentation tags. Much has been copied from the technical report and .NET Framework Class Library documentation on Microsoft's Developer Network [91] or merged with the existing code documentation already in C5. The documentation was extended with information about the many collection events in C6, which previously only existed in the C5 technical report. Furthermore, a consistent format has been applied to the XML documentation to enhance readability and preserve it through document maintenance; its formatting is also handled by ReSharper.

Code Contracts is able to emit contracts into the XML documentation files, which can then be used to create documentation with tools like SandCastle [14]. However, the current version of Code Contracts contains a year-old bug that hinders this [10]. Instead of using SandCastle, which uses the XML documentation files, but has a high entry cost, C6 uses Doxygen [138]. Doxygen works directly with the source files to generate a browsable HTML documentation that can be hosted online along with the project. This is also the solution that C5 currently uses.

3.4 The Interface Hierarchy

Since the interface hierarchy is such a big part of C5, this is also where many of the changes were made. Most changes were on the interface members, but the hierarchy itself

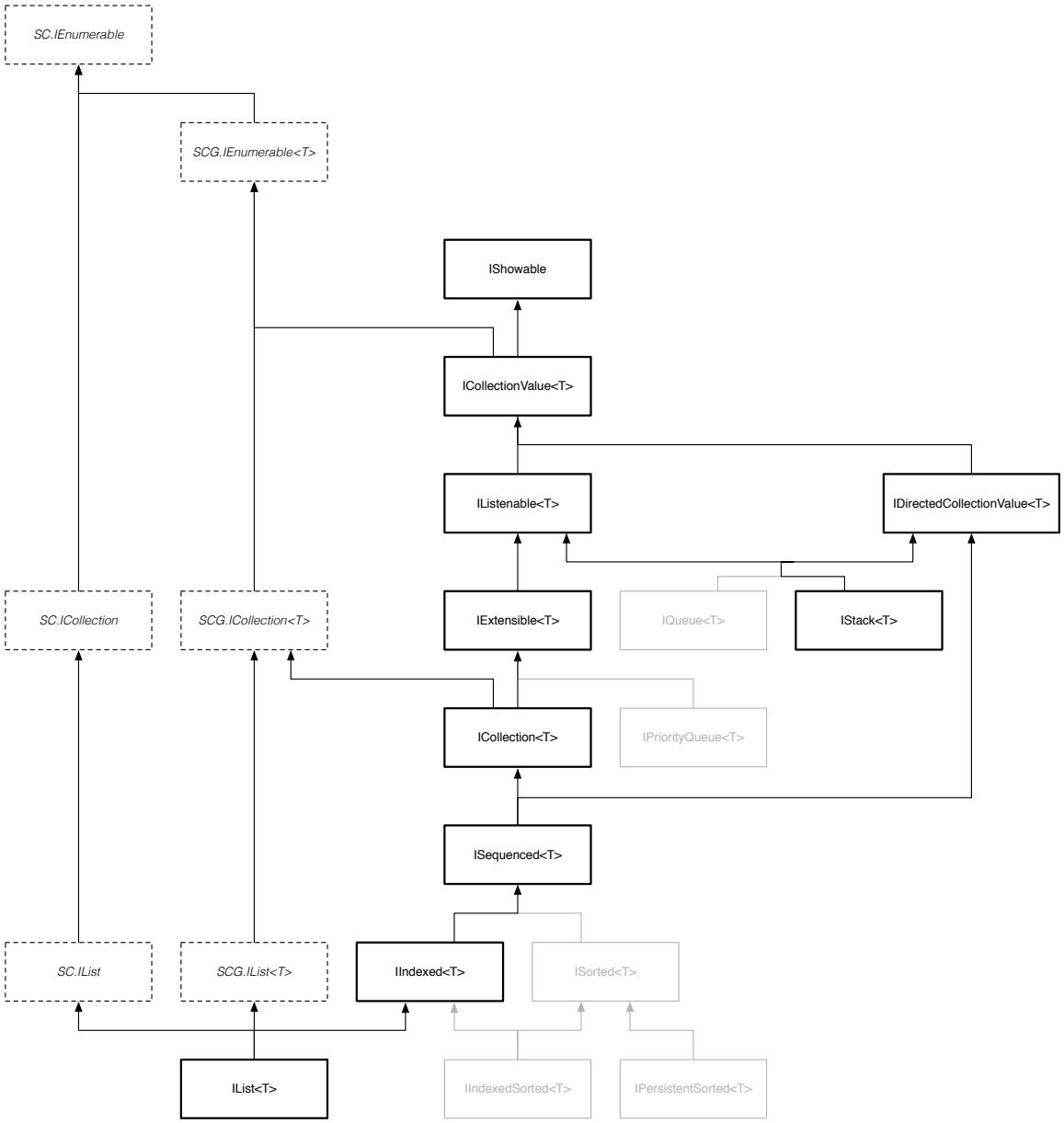


Figure 3.2: The new collection interface hierarchy in C6: the thick-bordered interfaces are relevant for `ArrayList<T>`*; the dashed, italicized interfaces are from the* `System.Collections` *(SC) and* `System.Collections.Generic` *(SCG) namespaces; and the faded interface have either not been added to C6 yet or have not been relevant for this project. Notice the addition of* `IListenable<T>` *and elimination of* `IDirectedEnumerable<T>`*.*

was also slightly changed in C6, as can be seen in figure 3.2. Further changes might be desirable or even required as more collections are introduced in the future.

The `IListenable<T>` interface was introduced in C6 to move events away from the `ICollectionValue<T>` interface (and thereby also `IDirectedCollectionValue<T>`). In C5, a collection value returned from a collection query does not have any listenable events, but still requires developers to implement them. All event-related members were therefore moved down the hierarchy, allowing just the collection classes – and not the returned collection values – to implement them.

The interface `IDirectedEnumerable<T>` was omitted in C6. The interface allows collections in C5 to return enumerables that can be enumerated in either direction. Though the interface is beautifully named and used throughout C5, its functionality is almost matched by the `IDirectedCollectionValue<T>` interface, which extends both `IDirectedEnumerable<T>` and `ICollectionValue<T>`. Most methods in C5 that return an `IDirectedEnumerable<T>` are implemented using other methods that already return an `IDirectedCollectionValue<T>`. Those that do not, can easily implement the new functionality; `ISorted<T>.RangeTo()` can for instance calculate the collection value size simply from the index of the last item. When the size cannot be calculated in constant time, this can easily be signaled by `ICollectionValue<T>`'s `CountSpeed`.

3.5 Interface Members

Several of the interface methods, especially on the `C5.ICollectionValue<T>` interface, have become obsolete due to new additions to the language and framework, e.g. the LINQ extension methods introduced in C# 3.5 [92]. Instead of keeping the methods around (in which case they need documentation, contracts, testing, and maintenance), most have been removed or replaced.

3.5.1 Removed Members

`C5.ICollectionValue<T>.Find()` has partly been replaced by the LINQ extension method `Enumerable.FirstOrDefault<T>()` [78], but to maintain functionality, the method has been replaced with the `C6.CollectionExtensions.Find<T>()` extension method, which even extends its scope to all `IEnumerable<T>`s. When the method is called on a backwards enumerated `C6.IDirectedCollectionValue<T>`, it returns the same result as the method `C5.IDirectedCollectionValue<T>.FindLast()` did, which is why `FindLast()` was left out of C6. Using the extension method instead should have no impact on performance.

Several of the methods on `C5.ICollectionValue<T>` have become obsolete: `Apply()` can be replaced by a simple `foreach` loop or a custom extension method; `Exists()` has been directly replaced by `Enumerable.Any()` [77]; `All()` has been directly replaced by `Enumerable.All()` [76]; and `Filter()` has been directly replaced by `Enumerable.Where()` [80]. Though equivalents of these methods still exist on `SCG.List<T>` [82–85] (for backwards compatibility), they have been removed from C6.

`C5.IExtensible<T>.Check()` has been directly replaced by Code Contracts as object invariant methods. This is described in more detail in section 6.9.

Both `C5.IStack<T>` and `C5.IQueue<T>` contain an `AllowsDuplicates` property, which is completely ignored when pushing or enqueuing items – or at least always expected to

be true. It could have been beneficial to move the property from C6.IExtensible<T> up the hierarchy to an ancestor shared by all three interfaces, but no meaningful solution was found to this problem yet.

The methods Map() and FindAll() on C5.IList<T>, which both returned a new list, were removed, since similar functionality can be achieved through LINQ [79][80]. They do still exist on SCG.IList<T> [81][83], so they might reappear later in C6, if deemed favourable.

C5.IList<T>.FIFO was discarded in C6. The property dictated from which end of a list items were removed, but its behavior was only partly enforced in C5 and often caused inefficient methods. Furthermore, the old behavior is still available through other interface methods, for instance, instead of using Remove(item) to remove the first instance of item (when FIFO was true), one can use RemoveAt(IndexOf(item)), which is just as (in)efficient.

3.5.2 Added Members

3.5.2.1 Collections with Null Items

C5 has a somewhat mixed attitude toward null items. On one hand, C5 allows null items in collections, since none of the methods check whether items are null. On the other hand, several of the array-based collections check for null items in their invariants, even though the collections seem to handle null just fine. C5 has no fail-fast approach to null items and offers no real support, when users wish to have a null-free collection.

In C6 the Boolean property AllowsNull was added to the ICollectionValue<T> interface to solve this problem. AllowsNull offers a way for users to decide whether a collection should allow null items and a way for code contracts to enforce that. A collection of value type values can naturally not contain null items, meaning that AllowsNull is always false for value types. For non-value type collections, the user can decide at construction time, but the default value is false, i.e. collections disallow null items. A user must therefore explicitly allow null items. This is intended to aid the user, so that collections only can contain null items, if the user wants it.

3.5.2.2 Finding Duplicates

C6.ICollection<T> was extended with a new FindDuplicates(T) methods, which returns an ICollectionValue<T> with all the items in the collection that are equal to a specified item. This can easily be achieved with LINQ's Enumerable.Where() for unstructured collections like lists, but is much more efficiently handled by sets or by bags that stores duplicates using a count.

3.5.3 Changed or Moved Signatures

3.5.3.1 All-Suffixed Methods

Methods with the suffix All that take an enumerable of items, e.g. RemoveAll() and ContainsAll(), have had the suffix replaced with Range to better match the naming conventions in the System.Collections.Generic namespace [97][99][100]. This is also to

avoid confusing these `All`-suffixed methods with the Boolean, predicate-based, universal quantifiers such as `Enumerable.All()` and `Contract.ForAll()`. Those of the methods that might modify the collection now return a Boolean value indicating whether the collection was actually modified. This provides the information directly to the user, instead of requiring the user to manually retrieve it. This also helps in writing code contracts and is closely related to when events must be raised.

3.5.3.2 Priority Queue

Though C6 contains no priority queue yet, the `IPriorityQueue<T>` interface is already part of the interface hierarchy in C6. The interface did see some changes to its methods. Many of the methods had a prefix of `Delete`. The prefix was replaced with `Remove`, because the methods do not actually delete the items, but rather remove them from the collection. This was already reflected in the documentation. `IPriorityQueue<T>.Find()` was renamed to `TryFind()`

`C5.IPriorityQueue<T>.Find(IPriorityQueueHandle<T>, out T)` checks whether a priority queue handle is associated with a collection, in which case it returns `true` and assigns the associated value to the `out` parameter. In C6, the method has changed name to `Contains()`. Furthermore, it comes in an overloaded version without the `out` parameter, which can be used with Code Contracts. Methods with `out` parameters can be difficult to use in one-liner code contracts, because the `out` variable must be declared beforehand. This will likely also benefit users of the library that do not need the `out` parameter.

3.5.3.3 Duplicates

`C5.ICollection<T>.RemoveAllCopies(T)` was renamed to `RemoveDuplicates(T)` to better match the general naming in C6. The method also returns a Boolean value indicating whether any items were removed.

Likewise, `ICollection<T>.ContainsCount(T)` was renamed to `CountDuplicates(T)`.

3.5.3.4 Index Ranges

`IIndexed<T>.this[int start, int count]` was in C6 changed to `GetIndexRange(int start, int count)`, as indexed properties with more than one parameter should be avoided [11, section 5.2.1]. To match the new member name and to avoid using the word *interval* outside of C6.Intervals (if/when that is added), `IIndexed<T>.RemoveInterval()` was likewise renamed to `RemoveIndexRange()`.

3.5.3.5 Fixed Size Collections

Both `C5.IList<T>` and `SCG.IList<T>` provide the `IsFixedSize` property, which determines whether a list can change its size. Certain inherited methods, like `Add()`, require that this property is `false`. To properly enforce this would require us to add preconditions to inherited methods in `IList<T>`, which Code Contracts does not allow [56, chapter 3].

We can either solve the problem by trying to cast objects to `IList<T>` to access `IsFixedSize` (this is a bad solution), or we can simply move the property up the inheritance hierarchy to where it is needed. The preconditions can thereby be added the first time the methods are declared. `IsFixedSize` is therefore found in `IExtensible<T>` in C6.

3.5.4 Modified Behavior

3.5.4.1 Considering a Collection Changed

A consequence of changing a collection is that its associated enumerables are invalidated. This is something we would like to avoid whenever possible. Pure methods can by definition never change a collection, but a non-pure method does not necessarily have to change the collection either. For instance, trying to remove an item that is not in a collection, will not change the collection. Neither will adding an existing item to a set.

C6 is therefore less strict than C5, when it comes to considering a collection changed. Only when a collection actually changes will the collection's internal version number (known as "stamp" in C5, but as "version" in C6 due to `SCG.IList<T>` [72]) change. We adopt the design principle that the version number changes if and only if the collection raises a `CollectionChanged` event (given a listener is assigned). We therefore do not consider a collection changed, when new event listeners are added or removed.

Be aware, that we cannot necessarily observe, that a collection has changed. For instance when updating an item with itself or reversing a list containing the characters of a palindrome, the collection changes, but the change is not observable. C6 makes no clever attempts to avoid making an update. However, reversing an empty or single-item list does not change it, nor does sorting a sorted collection.

3.5.4.2 Collection Values as Return Values

An `ICollectionValue<T>` or an `IDirectedCollectionValue<T>` returned from a collection method now has the same status as an enumerable: the returned collection value (whether directed or not) is only valid as long as the collection that returned it has not been changed. Changes to the collection will cause any method call on the collection value to throw an `InvalidOperationException` – just as an enumerable would [71]. This allows `ICollectionValue<T>` to defer execution and/or be lazily evaluated (just as LINQ's extension methods [75]) which can help improve performance. The effect of this change in regards to `ArrayList<T>` is discussed in section 6.8.

3.5.4.3 Default Value for Duplicates By Counting

`IExtensible<T>.DuplicatesByCounting` is by convention now always `false` for collections with set semantics. This makes it easier to predict behaviour without assessing the value of `IExtensible<T>.AllowsDuplicates`, for instance when writing code contracts.

3.5.4.4 Bad Enumerables

`AddRange()`, `InsertRange()`, `RemoveRange()`, and `RetainRange()` take an enumerable of items used for either adding or removing items from the collection. The enumerable could potentially throw an exception while being enumerated, in which case C5 only adds the items retrieved before the exception. This results in rather complicated code, especially in `ArrayList<T>`, where a try-finally block is used to remove any holes created from fewer items being added than initially expected.

To simplify the behavior, both for users and developers of the library, the methods are now considered pure, if the enumerable throws an exception. This is given by the code contract:

```
1 // Collection doesn't change if enumerator throws an exception
2 EnsuresOnThrow<Exception>(this.IsSameSequenceAs(OldValue(ToArray())));
```

Listing 3.1: Ensures that collection remains unchanged if exception is thrown (see sections 2.4 and 4.1.2).

This makes the outcome more predictable, since it can otherwise be difficult to know which items were actually added before the exception was thrown. This decision has been made from a viewpoint of `ArrayList<T>`, where it greatly simplifies things at the expense of extra memory usage.

3.6 Altered or Removed Classes

3.6.1 Event Handlers and Arguments

All the events found in C5's `ICollectionValue<T>` use custom event handler delegates. `C6.IListenable<T>` instead uses the generic `EventHandler<TEventArgs>` [11, section 5.4] [123]. Removing the delegates has little to no practical implications for library users.

The custom `EventArgs` remain, though their public read-only *fields* have been changed into public read-only *properties*, as the former is now discouraged [11, section 3.1.3].

3.6.2 Comparer Factory

C5 contains a generic comparer factory that helps creating comparers using delegate methods. In C6, `ComparerFactory` is now static and its methods have been made generic instead of the class itself. The classes `Comparer<T>` (previously `InternalComparer<T>`) and `EqualityComparer<T>` (previously `InternalEqualityComparer<T>`) have been nested in `ComparerFactory` as private classes. This makes it more clear that `ComparerFactory` should never be instantiated, and that the nested classes should only be constructed using the factory methods.

A method was also added to help create an equality comparer based on reference equality, and an extension method was added to turn a `Comparison<T>` into an `IComparer<T>`.

3.6.3 Event Types

`EventTypeEnum` has changed name to `EventTypes`, since enumerations should not have an `Enum` suffix, and because flags should have a plural type name [11, section 3.5.3].

3.6.4 Exception Classes

C5 uses the custom `CollectionModifiedException`, when a collection is modified during enumeration. According to the documentation of `IEnumerator.MoveNext()` [71], the expected exception type is `InvalidOperationException`, which the C5 exception does not inherit from. The correct exception class was instead used in C6.

The `NoSuchItemException` is currently not needed in C6, but could be useful when `ISorted<T>` is introduced. The rest of the custom exceptions in C5 are all exceptions thrown when input is invalid. These remaining exceptions have directly been replaced by Code Contracts in C6, as discussed in section 4.2.3. C6 therefore currently contains no custom exception classes.

3.6.5 Sorting

C5 supports introspective sorting, which guarantees efficient in-place sorting. Starting with the .NET Framework 4.5, the overloaded `System.Array.Sort()` methods also use introspective sort [74], which is why the `Sorting` class from C5 has not made it into C6 in its current version. In the future, a `C6.Sorting` class might be reintroduced to provide more sorting algorithms, e.g. the stable TimSort [142].

3.6.6 Showable Objects

The `IShowable` interface is intended to make objects (read *collections*) printable, for instance during debugging. There are, however, new alternatives for debugging in the .NET framework: the debugger display attributes [90] allow us to decide how classes (especially our collections) are displayed in the debugger view. This is much more flexible and useful than a potentially cropped string output.

`IShowable` has currently made it into C6 with the static `Showing` class, as it does seem to have some use cases, for instance when printing the collection in the (test) console. However, the debugger display attributes have been added to `ArrayList<T>` and is intended to also extend the rest of the data structures once added.

Chapter 4

Code Contracts in C6

Contracts are a major part of C6; not only do they provide the extra *C* to the library name, but they also help formalize some of the previously informal and implicit specifications of C5. This chapter examines the new contracts in C6 and considers the many challenges encountered while adding them.

4.1 Contract Helpers

Contracts must be written as one-liners when using Microsoft's Code Contracts. This is often not a problem, since the framework provides plenty of helpful methods that enable things like universal or existential quantification in-line:

```
1 // All items must be non-null if the collection disallows null items
2 Requires(AllowsNull || ForAll(items, item => item != null), ItemsMustBeNonNull);
```

Listing 4.1: A contract that uses Code Contracts' universal quantifier helper method.

When a contract cannot be expressed with a single line of code, or when a single line is too difficult to read or takes up too much space, Code Contracts allows the use of `Pure` methods instead [56, section 4.3]. C6 uses a dozen or more pure contract helpers located in the static `ContractHelperExtensions` class as extension methods, and this section introduces the different extension methods created. The contract helper methods are only intended for contract use and are not part of the standard C6 library.

4.1.1 Order-Indifferent Equality

LINQ contains the extension method `Enumerable.SequenceEqual()` [63], which determines whether two sequences are equal by comparing their elements using a default or custom equality comparer. This works great for verifying that two ordered enumerables are equal. Unfortunately, no equivalent exists for unordered enumerables.

C6 has a helper method called `UnsequenceEqual()`, which compares the elements in unordered enumerables, not unlike `C6.ICollection<T>`'s `UnsequencedEquals()` method. The method works by copying the two enumerables into arrays, which are then sorted on hash code and compared pair-wise as long as the arrays contain equal elements. If the collections contain few hash collisions, the method is expected to run in linearithmic time. This could be implemented using sets in expected linear time, but currently does not because C6 has no sets yet.

A bug was found in `UnsequenceEqual()` while testing `ArrayList<T>.Shuffle()`. The problem is described in section 7.2.2.

4.1.2 Identicality Comparer – Beyond Item Equality

When writing contracts, it is often desirable to ensure that two items or collections of items are the same after an operation has been performed. A collection's equality comparer can tell whether two items are considered equal, but it cannot tell whether the items are the same. In other words, it checks for equality, but not *identicality*. If the items were always reference types, they could simply be compared using `ReferenceEquals()`, but when working on generic types that simply will not work: value types get boxed before they get passed to the method, and since the boxing objects are different, even equal value types appear to be reference unequal [62]. So how is identicality in generic methods best assessed?

Instead of using the collection's equality comparer, we use a helper method that returns an appropriate identicality comparer based on a generic type. If the generic type is a reference type, it simply returns a reference equality comparer. If the type is a simple type, like an integral type or an `enum` type, it uses the type's default equality comparer.

Things become a bit more complicated, if the type is a `struct`: the structure's own equality comparer cannot be used, as it could be overridden by the user or use `ValueType`'s `Equals()` method [65], which does not use reference equality for reference types. We must instead use reflection on the structure to recursively compare its fields using the type-dependant identicality comparers. The helper method is shown in listing 4.2.

Several of the contract helper methods use the identicality comparer internally:

- `ContainsSame()` checks whether an enumerable contains a given item.
- `ContainsSameRange()` checks whether one enumerable contains all the same items as another enumerable.
- `CountSame()` counts how many times an enumerable contains a given item.
- `IsSameAs()` checks whether two items are the same.
- `IsSameSequenceAs()` checks whether two enumerables contain the same sequence of items. This is equal to `SequenceEqual()` using an identicality comparer. Notice that the enumerables do not have to be of the same type, only their items.
- `HasSameAs()` checks whether two enumerables contain the same items without regards to order. This is equal to `UnsequenceEqual()` using an identicality comparer. Notice that the enumerables do not have to be of the same type, only their items.

The methods and the identicality comparer are extensively used in the code contracts, and *only* in the code contracts. They are therefore not optimized for performance. The identicality comparer has no influence on user code or item equality in general.

4.1.3 Counting Duplicates

The extension method `CountDuplicates()` works just like `ICollection<T>`'s method with the same name. The extension method is partly used for classes that do not implement `ICollection<T>`, but also in the `CountSame()` extension method, which uses the identicality comparer described in section 4.1.2. With the extension method, we only need to get the comparer once, which helps performance, as seen in listing 4.3.

```
[Pure]
public static IEqualityComparer<T> GetIdenticalityComparer<T>() {
    if (!typeof(T).IsValueType) {
        return ComparerFactory.CreateReferenceEqualityComparer<T>();
    }
    if (typeof(T).IsPrimitive) {
        return EqualityComparer<T>.Default;
    }
    return CreateStructComparer<T>();
}

[Pure]
public static IEqualityComparer<T> CreateStructComparer<T>() =>
    ComparerFactory.CreateEqualityComparer<T>(StructEquals, type => type.GetHashCode());

[Pure]
private static bool StructEquals<T>(T x, T y) {
    foreach (var fieldInfo in typeof(T).GetFields(Instance | Public | NonPublic)) {
        // Get field values
        object thisObject = fieldInfo.GetValue(x), thatObject = fieldInfo.GetValue(y);
        var fieldType = fieldInfo.FieldType;

        if (fieldType.IsClass) {
            // Compare reference equality for objects
            if (!ReferenceEquals(thisObject, thatObject)) {
                return false;
            }
        }
        else if (fieldType.IsPrimitive) {
            // Compare values for simple types
            if (!thisObject.Equals(thatObject)) {
                return false;
            }
        }
        else {
            // Call method recursively for structs
            // We make generic method, because variables are non-typed objects
            var methodInfo = typeof(ContractHelperExtensions)
                .GetMethod(nameof(StructEquals), Static | NonPublic);
            var genericMethod = methodInfo.MakeGenericMethod(fieldType);
            var structEquals = (bool) genericMethod.Invoke(
                null,
                new[] { thisObject, thatObject }
            );
            if (!structEquals) {
                return false;
            }
        }
    }
    return true;
}
```

Listing 4.2: The identicality comparer factory that helps generate an equality comparer that checks whether two items are the same. `StructEquals()` *is used to compare* `struct`*s recursively to assess if they are likely the same structure.*

```
// Count identical items using LINQ's Count() (repeatedly asks for an identicality comparer)
Ensures(this.Count(x => GetIdenticalityComparer<T>().Equals(x, item)));

// Count identical items using contract helper (ask for one identicality comparer)
Ensures(this.CountSame(item) == OldValue(this.CountSame(item)) + 1);
```

Listing 4.3: Using `CountSame()` *only requires us to retrieve one identicality comparer.*

4.1.4 Skip Range or Index

`SkipRange(int startIndex, int count)` will skip a section of an enumerable, i.e. it will enumerate the first `startIndex` items, then skip the next `count` items, and then enumerate the remaining items. This is practical when consecutive items are removed from a collection and the collection needs to be compared to a reference enumerable.

`SkipIndex()` uses `SkipRange()` to skip a single index.

4.2 Interface Contracts

Most contracts in C6 are found on the interfaces. This is expected because all contracts written on an interface are inherited by its implementing classes. The contracts are inspired by the documentation, the tests, and the code. This section will give an overview of the many interface contracts in C6.

4.2.1 Contract Classes

Interface contracts are written in special abstract contract classes as described in section 2.4 and shown in listing 2.8. The contract classes for the C6 interfaces are stored in the same file as the interfaces themselves to keep interface and contract close together and make them more easily accessible. A contract class has the same name as its interface but with a suffix of `Contract`, e.g. `IListContract<T>` contains the contracts for `IList<T>`.

Contract classes cannot inherit from other contract classes and must therefore implement all members in the interfaces it implements (Code Contracts accepts abstracts methods in that case). C6 stores those methods in a `Non-Contract Methods` region [101] at the end of each contract class.

4.2.2 Pure Methods

Code Contracts uses the `Pure` attribute to mark members that are free of observable side-effects [27]. The pure members of a class may not change the class or reference type method arguments, but it may create new objects; for this reason `ToArray()` is considered pure, but `CopyTo()` is not. Only pure methods can be used in code contracts, and Code Contracts will emit a warning, if non-pure methods are used.

In C6, almost any method that does not raise events is marked pure. Exceptions to this are methods with `out` parameters and the methods used to register event handlers on the collections, as they do alter the object, but not the collection.

```
1 if (divisor == 0) {
2     throw new DivideByZeroException();
3 }
4 if (item == null) {
5     throw new ArgumentNullException();
6 }
7 if (i < 0 || array.Length <= i) {
8     throw new ArgumentOutOfRangeException();
9 }
```

Listing 4.4: Input validation using if-then-throw statements that throw "unchecked" exceptions.

4.2.3 Preconditions

C5 is filled with classic *if-then-throw* statements, like the ones in listing 4.4. These are all checks that ensure that methods are called correctly based on input and/or object state. The type of exception thrown here is known as *unchecked exceptions* in Java [112] and Spec# [3, section 1.1], but has no name in C#. They represent problems that we cannot expect (nor wish) the client code to recover from. They are due to a programming problem in the calling code; Eric Lippert rightfully calls them *boneheaded* exceptions [36]. These exceptions are very different from *checked* exceptions, where the program is momentarily in a state from which it can and should recover, e.g. when a file is missing on disk, or the connection to a resource is lost.

With Code Contracts, all "unchecked" exceptions are replaced with preconditions, which also cannot be caught (see section 5.4.1) as they too represent problems that client code should not recover from. The if-then-throw statements from C5 are rather easy to convert to Code Contracts, and all the statements have been moved from the implementations in C5 to the interface contracts in C6 as Code Contracts preconditions using `Contract.Requires()`. All the general preconditions on interface members in C6 are seen in listing 4.6. The list is rather short, as there are only so many different things that can be required from the user input.

4.2.4 Postconditions

Where preconditions tend to be general, postconditions are often more specific. Postconditions describe how an operation alters the object state, and since each method likely will alter the collection differently, few postconditions tend to be reused. Listing 4.7 contains a sample of some of the postconditions that often occur.

The general philosophy when writing postconditions (and invariants) is *the more the better*. This is well illustrated by `IQueue<T>.Enqueue()`'s postconditions seen in listing 4.5.

```
1  // The collection becomes non-empty
2  Ensures(!IsEmpty);
3
4  // Adding an item increases the count by one
5  Ensures(Count == OldValue(Count) + 1);
6
7  // The collection will contain the item added
8  Ensures(this.ContainsSame(item));
9
10 // Adding the item increases the number of equal items by one
11 Ensures(this.CountSame(item) == OldValue(this.CountSame(item)) + 1);
12
13 // The item is added to the end
14 Ensures(item.IsSameAs(this.Last()));
15
16 // The added item is at the end of the queue
17 Ensures(this.IsSameSequenceAs(OldValue(ToArray()).Append(item)));
```

Listing 4.5: The postcondition on `IQueue<T>.Enqueue()`.

The first five contracts each check different things: the first ensures that the queue is not empty; the second ensures that the count increases by one; the third ensures that the queue contains the added item; the fourth ensures that the queue only contains one more of that item; and the fifth ensures that the item is the last in the queue.

```
// Collection must be writable
Requires(!IsReadOnly, CollectionMustBeNonReadOnly);

// Collection must be non-fixed-sized
Requires(!IsFixedSize, CollectionMustBeNonFixedSize);

// Collection must be non-empty
Requires(!IsEmpty, CollectionMustBeNonEmpty);

// Argument must be non-null
Requires(arg != null, ArgumentMustBeNonNull);

// Item must be non-null if collection disallows null items
Requires(AllowsNull || item != null, ItemMustBeNonNull);

// Items must be non-null if collection disallows null items
Requires(AllowsNull || ForAll(items, item => item != null), ItemsMustBeNonNull);

// Collection must not already contain item if collection disallows duplicate values
Requires(AllowsDuplicates || !Contains(value), CollectionMustAllowDuplicates);

// Collection must not already contain the items if collection disallows duplicate values
Requires(AllowsDuplicates || ForAll(this, item => !Contains(item)),
    CollectionMustAllowDuplicates);

// Index must be within bounds (Count excluded)
Requires(0 <= index, ArgumentMustBeWithinBounds);
Requires(index < Count, ArgumentMustBeWithinBounds);

// Index must be within bounds (Count included)
Requires(0 <= index, ArgumentMustBeWithinBounds);
Requires(index <= Count, ArgumentMustBeWithinBounds);

// Range must be within bounds
Requires(0 <= startIndex, ArgumentMustBeWithinBounds);
Requires(startIndex + count <= Count, ArgumentMustBeWithinBounds);

// Argument must be non-negative
Requires(0 <= arg, ArgumentMustBeNonNegative);

// Event is listenable
Requires(ListenableEvents.HasFlag(Changed), EventMustBeListenable);

// Event is active
Requires(ActiveEvents.HasFlag(Changed), EventMustBeActive);
```

Listing 4.6: A complete list of the general interface preconditions in C6.

```
// Result is non-null
Ensures(AllowsNull || Result<T>() != null);

// Result is the same as skipping the first index items
Ensures(Result<T>().IsSameAs(this.ElementAt(index)));

// The collection becomes non-empty
Ensures(!IsEmpty);

// The collection will contain the same item
Ensures(this.ContainsSame(item));

// Adding an item increases the count by one
Ensures(Count == OldValue(Count) + 1);

// Item is inserted at index
Ensures(item.IsSameAs(this[index]));

// Collection doesn't change if enumerator throws an exception
EnsuresOnThrow<Exception>(this.IsSameSequenceAs(OldValue(ToArray())));
```

Listing 4.7: A sample of the general interface postconditions in C6.

The last postcondition on `Enqueue()` essentially says the same thing as the others: when an item is enqueued, the new queue is equal to the old queue with the item appended to the end. Though the meaning might be the same, each postcondition checks a new aspect of the state: the first uses the `IsEmpty` property, the second uses `Count`, the third and fourth use extension methods, and the fifth uses the enumerator. Each contract can fail independently, which makes it much more obvious exactly where the problem is. Each new contract makes it increasingly more difficult to have an undetected error, and the more independent the contracts are of each other, the more ground they will cover. It is this kind of cross validation that makes Code Contracts so powerful.

Consider the postconditions on `IIndexed<T>.RemoveIndexRange()` in listing 4.8. It is easy to check that the collection contains the correct items after the method call using the `SkipRange()` extension method.

```
1  // Only the items in the index range are removed
2  Ensures(this.IsSameSequenceAs(OldValue(this.SkipRange(startIndex, count).ToList())));
3
4  // Removing an item decreases the count by one
5  Ensures(Count == OldValue(Count) - count);
```

Listing 4.8: The postconditions on `IIndexed<T>.RemoveIndexRange()`.

The postconditions can also be used to validate return values, as seen with the properties that return enumerations like `Speed` and `EnumerationDirection`. The postcondition in listing 4.9 ensures that the `enum` has a defined value, since C# does not ensure that.

```
1  // Result is a valid enum constant
2  Ensures(Enum.IsDefined(typeof(Speed), Result<Speed>()));
```

Listing 4.9: A postcondition found on any method that returns a `Speed` enumeration.

Some methods can be rather difficult to add postconditions to. Consider `IList<T>`'s `Shuffle()`, which randomly shuffles the items in a list. We cannot say anything about the order of the items afterwards, not even that it is different, which also causes problems when testing (section 7.2.2). This leaves us with one postcondition on the shuffle methods:

```
1  // The collection remains the same
2  Ensures(this.HasSameAs(OldValue(ToArray())));
```

Listing 4.10: The only postcondition found on `IList<T>`'s shuffle methods.

It is easier with methods like `Sort()`, where we also check that the collection is sorted afterwards:

```
1  // List becomes sorted
2  Ensures(IsSorted());
```

Listing 4.11: A postcondition ensuring that the collection is sorted after calling `Sort()` on a list.

4.3 Challenges with Contracts in C6

4.3.1 One Interface for Sets and Bags

The interface hierarchy in C5 introduced the `AllowsDuplicates` property to avoid having almost identical interfaces for sets and bags. The property affects how methods behave, which makes it more difficult to write code contracts. All code contracts must therefore take the different scenarios into account. `IExtensible<T>`'s `Add()` method for instance has the following contract, because the return value might change depending on the item and the value of `AllowsDuplicates`:

```
1 // Bags return true, sets the opposite of whether the collection already contained the item
2 Ensures(AllowsDuplicates ? Result<bool>() : OldValue(!this.Contains(item, EqualityComparer)));
```

Listing 4.12: A postcondition on `IExtensible<T>.Add()`'s return value.

If the collection is a bag, the item is always added, hence `Result<bool>()`. If the collection is a set, the item is only added if the collection did not already contain the collection, hence `OldValue(!this.Contains(item, EqualityComparer))`. The consequences are rarely too big, but the effect of `AllowsDuplicates` must be kept in mind when writing contracts or unit tests (see section 5.3.4).

4.3.2 Inheriting from Multiple Methods

Code Contracts does not allow us to strengthen (or weaken) our preconditions on any members we might inherit or implement [56, section 3.2]. This is a problem when members appear in more than one implemented interface. The member's effective precondition would be the conjunction of the inherited preconditions, possibly making it stronger, and since the contract tools cannot easily determine whether that is the case, Code Contracts simply forbids it.

One example of a method with multiple roots is `C6.IList<T>.IndexOf()`, which is inherited from both `SCG.IList<T>` and `IIndexed<T>`. If `C6.IList<T>` did not override the inherited members from the base interfaces, working on an instance of `C6.IList<T>` would make calls to `IndexOf()` ambiguous. One solution to this problem could simply be not to inherit from the `SCG.IList<T>` and `SCG.ICollection<T>` interfaces in C6. This does, however, seem to be an undesirable solution as it limits the use of the C6 collection.

We can override or redeclare the inherited members, but problems arise when we start implementing them. Besides the compiler issuing a warning, the contracts are inherited differently depending on how the members are implemented. Consider again `IndexOf()` in C6: if the method is called with an item not in the collection, the method returns the ones' complement of the index at which `Add()` would put the item. This is different from the standard libraries, which simply return -1. `SCG.IList<T>.IndexOf()` has a postcondition, as seen in listing 2.24, which says that the result must be greater than or equal to -1. If `IndexOf()` is only implements once in a class, the standard library's postcondition is violated. The postcondition is, however, removed from that implementation as soon as `SCG.IList<T>.IndexOf()` is implemented explicitly. The contract inheritance changes depending on which members are declared.

It took me a while to discover this odd behavior – I only discovered it when I finally implemented IList<T> in ArrayList<T>. Judging by my previous tests, I thought that contracts were always inherited and it therefore seemed redundant to redeclare them. I therefore mistakenly removed them from the contract classes [40]. It seems that contracts must be manually copied and maintained for each overwriting member, and implementing classes must explicitly implement all overridden methods. This has yet to be done in C6. It is still unknown whether we actually can strengthen our postconditions [41], especially when methods are explicitly implemented.

4.3.3 Is Valid

C5's IList<T> supports so-called views [31, section 1.4.11], also known as sublists in Java's collections [111]. Views allow operations on a section of items in an underlying list as if the view were a list of its own. Changes to the view are reflected in the underlying list, and vice versa. The use of views can make certain operations both more simple and more efficient, however, they pose a great challenge for Code Contracts.

Certain operations, such as shuffling or sorting the underlying list will invalidate the view (which is why C5.IList<T> is disposable). To signal whether a view is valid or not, IList<T> introduces the property IsValid. To call any other member of a view, the view must be valid:

```
1 Requires(IsValid);
```

Listing 4.13: A precondition required on all methods from ICollectionValue<T> to IList<T>.

This contract must be added to every method on IList<T>, i.e. also methods found higher up in the hierarchy on ICollectionValue<T> and ICollection<T>. Since Code Contracts does not allow extra preconditions on inherited members [56, section 3.2], this requires the new precondition to be stated when a member is first declared. IsValid must therefore be moved from IList<T> up to ICollectionValue<T>, which introduces a new property that has little to no relevance for most collections implementing the other interfaces in the hierarchy.

Alternatively, if-then-throw statements could be used in the classes that implement IList<T>. This would, however, go completely against the idea of using Code Contracts.

In C6, returned ICollectionValue<T>s work as enumerables: they are only valid as long as the original collection is unchanged. However, there is currently no way to figure that out; you need to keep track of it yourself (the same is true for an IEnumerable<T>). Adding IsValid to ICollectionValue<T>, would allow users to check whether the object was still valid and usable. Moreover, this would be usable in code contracts.

This could also help users understand the difference between an IEnumerable<T>, an ICollectionValue<T> and a view: an ICollectionValue<T> is an intelligent version of an IEnumerable<T> that breaks if the original collection changes in any way; a view is an intelligent version of an ICollectionValue<T> that breaks only if the original collection changes in a certain way.

Partly due to the challenges with IsValid, views have not made it into C6 yet.

4.3.4 Returned Collection Values

C6 uses both `ICollectionValue<T>` and `IDirectedCollectionValue<T>` to return *intelligent* enumerables as described in section 3.5.4.2. The use of the collection values instead of simple enumerables gives a lot of flexibility, because the collection values for instance know their own size and can be reversible if they are directed. But with this extra functionality comes a lot of extra work: all the members of a returned collection value must be verified. To make matters worse, this must be done for all the different scenarios in which a collection value is returned. With at least five methods on `ArrayList<T>` returning a collection value (section 6.8), that makes for a lot of tests.

To assess the result of any method returning an `ICollectionValue<T>` or its directed counterpart `IDirectedCollectionValue<T>`, we use two tricks to help make the task a lot easier: we add extensive postconditions and we compare the result to a reference object. This section shows the code contracts, whereas section 5.4.3 explains the unit tests.

In order to check all the members of an `ICollectionValue<T>` returned by a collection query, we add a postcondition for each member on the method that returns it. All members except `CopyTo()` are pure and are therefore callable in our contracts. Listing 4.14 shows all the contracts on `IDirectedCollectionValue<T>`'s `Backwards()`. The use of contracts ensures that everything is always checked, even if our tests were to miss something. This drastically reduces the risk of errors.

```
1  public IDirectedCollectionValue<T> Backwards() {
2      // No preconditions
3
4      // Result is non-null
5      Ensures(Result<IDirectedCollectionValue<T>>() != null);
6      // Result enumeration is backwards
7      Ensures(Result<IDirectedCollectionValue<T>>().IsSameSequenceAs(this.Reverse()));
8      // Result allows null if this does
9      Ensures(Result<IDirectedCollectionValue<T>>().AllowsNull == AllowsNull);
10     // Result has same count
11     Ensures(Result<IDirectedCollectionValue<T>>().Count == Count);
12     // Result count speed is constant
13     Ensures(Result<IDirectedCollectionValue<T>>().CountSpeed == Speed.Constant);
14     // Result direction is opposite
15     Ensures(Result<IDirectedCollectionValue<T>>().Direction.IsOppositeOf(Direction));
16     // Result is empty if this is
17     Ensures(Result<IDirectedCollectionValue<T>>().IsEmpty == IsEmpty);
18     // Result array is backwards
19     Ensures(Result<IDirectedCollectionValue<T>>().ToArray()
           .IsSameSequenceAs(ToArray().Reverse()));
20
21     return default(IDirectedCollectionValue<T>);
22 }
```

Listing 4.14: The contracts on `IDirectedCollectionValue<T>.Backwards()` *validate all the members.*

4.3.5 Equality Comparers Versus Order Comparers

An ordered collection has both an `IEqualityComparer<T>` and an `IComparer<T>`. The two comparers must agree: items that have the same order according to the `IComparer<T>` must also be equal according to the `IEqualityComparer<T>`. Code contracts cannot enforce this, just like they cannot ensure that the equality comparer generates equal hash codes for equal objects. This remains the responsibility of the library user.

4.3.6 Sequenced and Unsequenced Equality

`ICollection<T>` and `ISequenced<T>` contain methods to compare and generate hash codes for unordered and ordered collections. It can be difficult to write contracts for the hash codes without having some kind of reference. C6 therefore uses the helper classes `UnsequencedEqualityComparer` and `SequencedEqualityComparer` to define the expected hash code of a collection.

Unfortunately, the classes cannot implement `IEqualityComparer<ICollection<T>>` and `IEqualityComparer<ISequenced<T>>`, since comparing the collection and calculating a hash code for all its items requires an `IEqualityComparer<T>`. The two collections would be required to have equal equality comparers, and there is no consistent way to ensure that with code contracts. Instead, the comparer classes take an item equality comparer, which is then used when comparing.

The comparer classes' internal "random" values used to calculate the hash codes are currently fixed in order to make them work with serialization and caching. The same is true for C5.

4.4 Solution Configurations and Distributions

One of the big benefits of Code Contracts is that it allows contracts to be distributed along with the code. There are many different ways to handle the contracts in production and this section takes a closer look at how C6 should handle it.

4.4.1 Solution Configurations

C6 contains four different solution configurations that each handle a different Code Contracts scenario. Code Contracts has rather fine-grained settings that allow us to precisely configure how our contracts should be compiled in Visual Studio. After installing the Visual Studio Code Contracts extension [24][69], a project gets a new configuration tab. The Code Contracts settings for the four different configurations are seen in figure 4.1. The first two are used for development and two for production:

`Debug` is the default configuration for development and simply has all code contracts enabled, but not static checking.

`Debug with Static Contract Checking` is the same as `Debug`, but with static checking enabled.

`Release` is the default release configuration. It only has public surface preconditions enabled that do not use quantifiers. This corresponds to the normal release configuration from C5.

`Release with Contract Reference Assembly` is the only configuration completely without contracts written into the code. The contracts are instead accessible in a *contract reference assembly*, which can be used in third-parties solutions. This allows users to use C6's contracts when using the C6 interfaces.

4.4.2 NuGet

C6 is intended to be distributed with the package manager NuGet [18] just like C5. It does, however, seem that multiple packages might be needed: one for `Release` and one for `Release with Contract Reference Assembly`. Though there are multiple sources on how to include contracts in a NuGet package [9][12], I have not been able to make it work properly with C6. It seems to be a general problem [30][89], and requires more time to properly solve. An alternative solution is to provide only the `Release` build on NuGet, and then provide the contract reference assembly in the GitHub repository [34] for Code Contracts users to download separately.

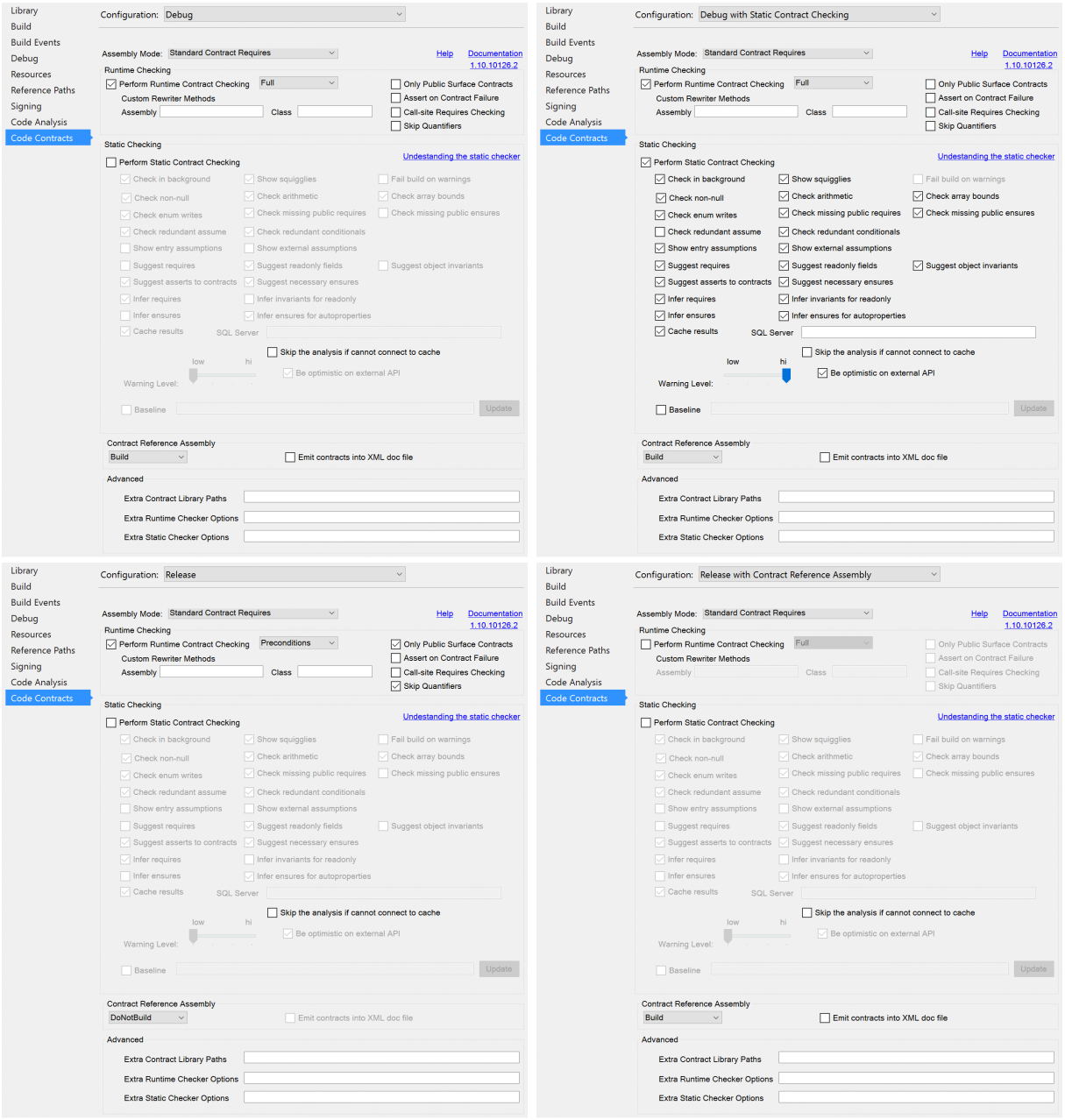


Figure 4.1: The four different Code Contracts solution configurations for C6. The top left is `Debug`*, top right is* `Debug with Static Contract Checking`*, bottom left is* `Release` *and the bottom right is* `Release with Contract Reference Assembly`*.*

Chapter 5

Unit Testing in C6

The test suite in C6 was completely rewritten although generally inspired by the unit tests found in C5. This chapter describes how the new tests were written and what the challenges were.

5.1 C6.Tests – The Test Project

All tests in C6 reside in the `C6.Tests` project in the `C6` solution. All the tests were rewritten, both in order to reflect the library's new behavior, but also to allow us to adopt a newer style of testing than the one applied in C5. The new testing style is presented in the rest of this section.

C6 has only been unit tested in this project. Though performance tests would have been informative and convention tests were considered (see appendix C), the time was simply not there.

5.1.1 The NUnit Testing Framework

C5 uses the open-source testing framework NUnit [109]. With NUnit 3.0 released in November 2015 [104], the testing framework was completely rewritten and many new and exciting things were introduced. It only seemed natural to use the newest version of the testing framework for the unit tests in C6.

NUnit has a very active open-source community, which is open to changes and new ideas. During this project, I have contributed to many of the open-source projects used in C6 [45], and among them NUnit, and I have even become an official NUnit.org contributor [108]. The currently latest version (NUnit 3.2.1 [105], released in April 2016) even contains the `Is.Zero` notation that I suggested and implemented [49]:

```
1 Assert.That(0, Is.Zero);
```

Listing 5.1: An assertion that compares the actual value (here `0`) with zero.

The new and recommended syntax in NUnit 3 is the very fluent *constraint syntax*, which focuses on making assertions easily readable and writable, demonstrated in listing 5.2. The classic syntax is now actually just a wrapper around the constraint syntax [107]. The constraint syntax becomes its own little test specific language, which can be easily extended to make writing tests even more easy than previously. Sections 5.4.1 and 5.4.2 describe how the syntax helps testing both Code Contracts' preconditions and C6's events in a very compact and elegant way.

```
// Classic syntax
Assert.AreEqual(expected, actual);

// Constraint syntax
Assert.That(actual, Is.EqualTo(expected));
```

Listing 5.2: A comparison of the classic NUnit syntax with the new fluent constraint syntax.

5.1.2 Test-Driven Development

The mantra of Test-Driven Development (TDD) is *red-green-refactor* [4, preface]: we first create a test that fails (goes *red*); we then quickly make the test pass (go *green*); and finally we *refactor* our code to eliminate duplication. Each step has its own purpose. We first make a test that goes red – see it fail within the test runner – to verify that the tested behavior is not yet implemented. This is a mild form of testing the test to ensure that it will and *can* fail, when we expect it to. In the next step we make the test pass as quickly as possible, potentially by introducing code duplication by copying test values into the implementation. The last step (*refactor*) is then used to remove the hard-coded duplication by refactoring the code, i.e. actually implementing the method.

I find that a less ideological approach to the last two steps is often more beneficial. Instead of simply hard-coding the expected test values, the *green* step should be used to implement the method. Once implemented, the tests work as a safety net, while the code is refactored to make it more readable or efficient.

Test-Driven Development is well suited for exploring new domains or extending old software, because the tests naturally guide the development and help shape the software and its interface. Doing TDD while writing interface contracts works great, as both promote test of the behavior rather than the implementation, simply because the implementation does not yet exist. Test-*driven* development was, however, less helpful when rewriting tests for C5, as the behavior of the library was already defined. I instead used the idea of test-*first* development [115], where I first wrote all the tests for each interface method, then verified that they all failed for the right reasons, and finally implemented the method in the data structure to make the tests pass. I found this to be a helpful and beneficial way to write code.

5.1.3 The Art of Unit Testing

The trendsetting book *The Art of Unit Testing* by Roy Osherove [114] has helped shape the tests in C6. Osherove's ideas have for instance influencing test names and internal structure.

5.1.3.1 Test Naming

Osherove recommends constructing the test names of three parts [114, section 7.3.1]:

- The name of the method being tested.
- The scenario under which it is being tested.
- The expected behavior when the scenario is invoked.

The three parts are written in CamelCase and separated by an underscore (_) in the following fashion: `MethodUnderTest_Scenario_Behavior()`. This helps readers of the tests to quickly get an overview of what is being tested, either when reading the test code or when reading the test method name in the test runner.

C6 has adopted this test naming convention for all interface and data structure tests. A proper naming convention specifically for C6's general test cases (section 5.3.1) has not yet been made; convention tests described in appendix C could, however, be one way to help enforce such a convention.

5.1.3.2 Arrange, Act, Assert (AAA)

Tests are divided into three parts: arrange, act and assert [114, section 2.4]. We first *arrange* all the necessary objects for our test. We then *act* on an object (perform the action we wish to test). And finally we *assert* what we expect from the test. We add comments to help better distinguish the parts:

```
1  [Test]
2  public void Count_EmptyCollection_Zero() {
3      // Arrange
4      var collection = GetEmptyCollectionValue<int>();
5
6      // Act
7      var count = collection.Count;
8
9      // Assert
10     Assert.That(count, Is.Zero);
11 }
```

Listing 5.3: A test divided into three parts: arrange, act and assert.

Using this subdivision makes it very clear what is needed to perform the test, what should be tested, and what is asserted. It can sometimes be necessary to delay execution, for instance when testing preconditions (section 5.4.1.1) or events (section 5.4.2.1), in which case the *act* is inlined using a delegate:

```
1  [Test]
2  public void IsSorted_NonComparables_ThrowsArgumentException() {
3      // Arrange
4      var items = GetNonComparables(Random);
5      var collection = GetList(items);
6
7      // Act & Assert
8      Assert.That(() => collection.IsSorted(), Throws.ArgumentException.Because("At least one
           object must implement IComparable."));
9  }
```

Listing 5.4: Two of the parts (act and assert) are joined to improve readability, when execution is delayed.

5.1.3.3 Multiple Asserts in a Single Test

The C5 library has a tendency to use multiple asserts within a single test method. This is strongly advised against [114, 7.2.5] as it covers up the bigger picture when tests fail. When assertions fail in NUnit, they throw an exception halting the execution of the test method. Any asserts following the failing assert are never executed, so their results remain

unknown to the tester. This makes it more difficult to assess the problem, since pieces of the puzzle are missing. By moving assertions into separate tests – either by using NUnit's `TestCase`s or stand-alone test methods – we see exactly which asserts fail.

In C6, most tests contain only a single assertion. There are certain scenarios that do require multiple asserts, if we wish to keep code duplication to a minimum. This is especially the case, when methods have `ref` or `out` parameters:

```
[Test]
public void UpdateOut_UpdateNewItem_False() {
    // Arrange
    var items = GetUppercaseStrings(Random);
    var collection = GetCollection(items);
    var item = GetLowercaseString(Random);
    string oldItem;

    // Act
    var update = collection.Update(item, out oldItem);

    // Assert
    Assert.That(update, Is.False);
    Assert.That(oldItem, Is.Null);
}
```

Listing 5.5: A test with two assertions, which assert the different returned values.

5.1.4 Randomly-Generated Test Data

C6 mainly uses randomly-generated test data. This approach has many benefits: it removes the focus from the actual values used and helps better convey the intent of the test (C5 extensively uses hard-coded test values, making the reader wonder why exactly those values were chosen); it makes it difficult to (wrongly) code an implementation that only matches the current test data (which is what ideological TDDers tend to do); and it allows us to run a test with new data simply by changing the seed.

To help generate random data, NUnit contains a `Randomizer` class [110]. Each test has its own random number generator accessible with `TestContext.CurrentContext.Random`. The randomizer uses a fixed seed that only changes under certain conditions, e.g. when new tests are added. This makes a failing test repeatable, which can otherwise be a challenge with randomly-generated data. All test classes in C6 inherit from the abstract `TestBase` class, which makes the randomizer easily accessible:

```
public abstract class TestBase {
    protected static Randomizer Random => TestContext.CurrentContext.Random;
}
```

Listing 5.6: The base class from which all test classes inherit.

I have suggested improvements to NUnit to make it even easier to work with random values [50]. These changes have currently not been designed or developed, but they could be a great asset to both C6 and NUnit.

5.1.4.1 FsCheck

In relation to randomly-generated test data, I think it is worth mentioning FsCheck [19], which is a .NET port of Haskell's QuickCheck [22] testing framework. FsCheck will generate random data and check that the result of your methods has certain properties. The test in listing 5.7 uses a randomly-generated string list to create an `ArrayList<string>`, which reversed twice should again be equal to the initial list. The test is repeated a fixed number of times with a new list of strings each time.

```
1  public static Arbitrary<IList<int>> IntLists => Gen.ListOf(Arb.Generate<int>())
2                                                    .Select(ints => new ArrayList<int>(ints))
3                                                    .ToArbitrary();
4
5  [Test]
6  public void Reverse() {
7      Prop.ForAll(IntLists, list => {
8          var expected = list.ToArray();
9          list.Reverse();
10         list.Reverse();
11         return list.SequenceEqual(expected);
12     }).QuickCheckThrowOnFailure();
13 }
```

Listing 5.7: A FsCheck test that checks if a reversed, reversed list is equal to its initial self.

FsCheck was not used in this project. FsCheck has a version that works together with NUnit, but only version 2.6.4. As the name suggests, FsCheck is written in F#, but can be used in C#. The documentation for C# does, however, seem to be sparse. Most tutorials available online are written for the first version of the library, which is unfortunate since the library changed greatly with the second version.

5.2 Test Helpers

Just as with code contracts, it can be very beneficial to have helper methods when testing. The test helper methods are found in the `TestHelper` class. The class is statically imported and its methods can be used directly without the class name.

A handful of the methods are used to generate test data: `GetStrings()` generates random 25-character strings; `GetLowercaseStrings()` and `GetUppercaseStrings()` generate random 25-character strings in lower- and uppercase respectively; `GetIntegers()` generates random integers. The methods all return arrays to avoid new random data to be generated if the return values are enumerated more than once. They are often used together with the abstract factory methods (see section 5.3) to easily generate non-empty collections, as seen in listing 5.9.

The helper methods can also be used to generate randomized test data quickly and descriptively: `GetCount()` generates a random integer between 10 and 20 that decides the number of items being generated; `GetIndex()` generates a random index for a given collection; `Choose()` selects a random item from an array; `Repeat()` will either repeat an item or repeatedly call an item-generating function a fixed number of times; `ShuffledCopy()` will copy and shuffle the items of an enumerable; `WithNull()` will insert a `null` value at a random position in an array; `WithRepeatedItem()` will concatenate a repeated item to an enumerable and then shuffle it.

Several of the test helpers mimic the behavior of C6 methods: `InsertItem()` inserts an item into an enumerable at a specified index; `InsertItems()` similarly inserts an enumerable of items into another enumerable at a specified index; `IndexOf()` finds the first index of an item in an array, and `LastIndexOf()` finds the last. This allows us to compare the result of the helper methods with the result from the collection.

Some of the helper methods are used to complement the constraint syntax, such as `Raises()`, `RaisesCollectionChangedEventFor()` and `RaisesNoEventsFor()` described in section 5.4.2.1. Likewise, `Because()` is used to shorten exception message tests:

```
1  // Without helper method
2  Assert.That(() => enumerator.MoveNext(),
       Throws.InvalidOperationException.With.Message.EqualTo(CollectionWasModified));
3
4  // With helper method
5  Assert.That(() => enumerator.MoveNext(),
       Throws.InvalidOperationException.Because(CollectionWasModified));
```

Listing 5.8: The `Because()` *extension methods help us assert exception messages.*

5.3 Interface Tests

The overall idea for testing the collection classes is to have a unit test class hierarchy that matches the C6 interface hierarchy. Each interface in `C6` has a matching abstract test class in `C6.Tests`, which is responsible for testing all aspects of the interface. The test class has the same name as the interface it tests, but with a `Tests` suffix, e.g. `ICollectionTests` tests `ICollection<T>` and `IListTests` tests `IList<T>`. If the tested interface has a parent interface, the test class inherits from the parent interface's test class, e.g. `ICollectionTests` inherits from `IExtensibleTests`, which inherits from `IListenableTests` and so on. Figure 5.1 shows the test class hierarchy. Notice that because multiple inheritance is not possible with classes in C#, test classes that test interfaces with multiple inheritance, such as `IIndexedSorted<T>`, must decide on a single parent; this choice can be somewhat arbitrary. The interface test hierarchy contains black-box tests written against the interfaces. The tests can be used for different implementations, which reduces duplication in our tests. C5 contains many duplicate tests acting on different implementations, and it can be very difficult to figure out if they differ.

All test classes have two abstract factory methods that return the interface under test: one must create an empty collection and the other a collection containing the items in a provided enumerable (the actual items in the collection are decided by enumeration order and the collection type). The method names are the interface under test prefixed with `GetEmpty` and `Get` respectively, as seen in lines 5 and 7 in listing 5.9. To lower the number of abstract methods, inheriting interface test classes implement the immediate parent's abstract factory methods using its own, as seen in lines 14 and 17 in listing 5.9.

To test a collection class, we simply inherit from the appropriate test class(es) and implement the abstract methods. `ArrayList<T>` uses three test classes: one inheriting from `IListTests`, one inheriting from `IStackTests`, and one inheriting from `TestBase`. The first two classes are the black-box interface tests, whereas the last class contains white-box tests for constructors and other `ArrayList<T>`-specific behavior.

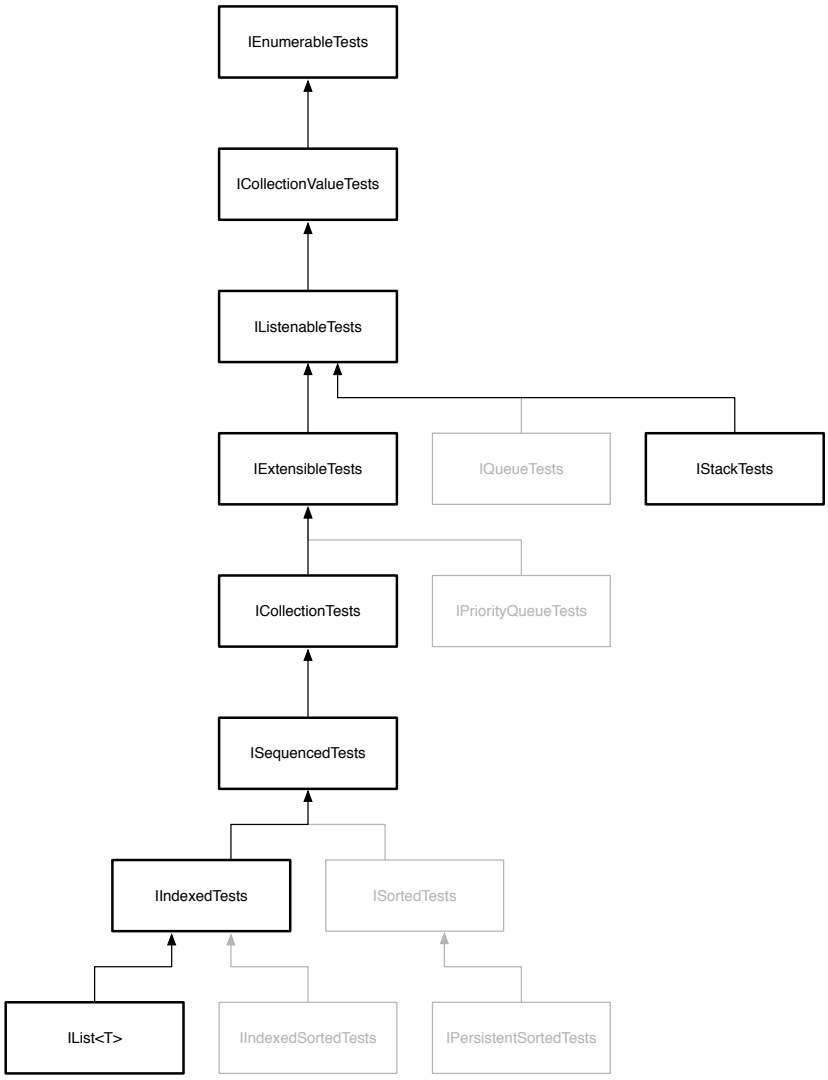


Figure 5.1: The test class hierarchy in C6. The dimmed classes were not relevant for this project.

```
1  [TestFixture]
2  public abstract class IExtensibleTests : IListenableTests {
3      #region Factories
4
5      protected abstract IExtensible<T> GetEmptyExtensible<T>(IEqualityComparer<T>
           equalityComparer = null, bool allowsNull = false);
6
7      protected abstract IExtensible<T> GetExtensible<T>(IEnumerable<T> enumerable,
           IEqualityComparer<T> equalityComparer = null, bool allowsNull = false);
8
9      #endregion
10
11     #region Inherited
12
13     protected override IListenable<T> GetEmptyListenable<T>(bool allowsNull = false)
14         => GetEmptyExtensible<T>(allowsNull: allowsNull);
15
16     protected override IListenable<T> GetListenable<T>(IEnumerable<T> enumerable, bool
           allowsNull = false)
17         => GetExtensible(enumerable, allowsNull: allowsNull);
18
19     #endregion
20
21     #region Helpers
22
23     private IExtensible<string> GetStringExtensible(Randomizer random,
           SCG.IEqualityComparer<string> equalityComparer = null, bool allowsNull = false)
24         => GetExtensible(GetStrings(random, GetCount(random)), equalityComparer, allowsNull);
25
26     #endregion
27
28     // Remaining test class...
29 }
```

Listing 5.9: The two abstract factory methods in `IExtensibleTests` *used to implement the inherited methods from* `IListenableTests`*. A private helper is used to generate a populated collection more easily.*

5.3.1 Test Reference List

As with preconditions, many methods require the same subset of unit tests to ensure that often trivial cases are properly handled. A test reference list was used to ensure that all necessary tests were included for each interface method. An excerpt of the list is shown in appendix D. Though the list is not exhaustive, these are some of the tests that reappear independently of the method's specific behavior. Method specific tests are still required and do not appear on the list. The most generic of the tests can easily be written using ReSharper' Live Templates as further shown in section 8.5, which makes the "boilerplate" tests rather easy to write.

5.3.2 Typical Interface Member Test

Listing 5.10 shows a typical C6 unit test. This particular test inserts an item at a random index in a list and checks that the list correctly contains the item. Lines 4 to 7 set up the needed objects by first generating a random string list, item and index. In line 7, the item is then inserted into a copy of the collection at the given index and the result is saved as an array. The operation under test is performed in line 10 and the result is asserted in line 13. Notice that we check that the enumerables contain the same items by using a custom reference equality comparer, since NUnit has no support for this yet [48].

```
[Test]
public void Insert_RandomCollectionRandomIndex_ItemAtPositionIndex() {
    // Arrange
    var collection = GetStringList(Random);
    var item = GetString(Random);
    var index = GetIndex(collection, Random, includeCount: true);
    var expected = collection.InsertItem(index, item);

    // Act
    collection.Insert(index, item);

    // Assert
    Assert.That(collection, Is.EqualTo(expected).Using(ReferenceEqualityComparer));
}
```

Listing 5.10: The test checks that an inserted item is at the right place and no other items have changed.

Consider how structured and concise the test is due to the extensive use of helper methods. It is rather easy to read the test, even without knowing exactly what the helper methods do.

5.3.3 Fixed-Value Tests

Some properties always return a fixed value for a given data structure. These are the properties that specify a collection's behavior. To allow us to test the data-structure-specific values, abstract properties are introduced in the interface classes. The data structure test class must simply implement the abstract properties to specify the data structure's behavior. The `IList<T>` test for `ArrayList<T>` can be seen in listing 5.11:

```
[TestFixture]
public class ArrayListListTests : IListTests {
    protected override bool AllowsDuplicates => true;
    protected override Speed ContainsSpeed => Speed.Linear;
    protected override bool DuplicatesByCounting => false;
    protected override Speed IndexingSpeed => Speed.Constant;
    protected override bool IsFixedSize => false;
    protected override bool IsReadOnly => false;
    protected override EventTypes ListenableEvents => All;

    // Factory constructors omitted...
}
```

Listing 5.11: The fixed properties for `ArrayList<T>` as used by the `IList<T>` tests.

The properties can then easily be tested:

```
[Test]
public void AllowsDuplicates_RandomCollection_AllowsDuplicates() {
    // Arrange
    var collection = GetStringExtensible(Random);

    // Act
    var allowsDuplicates = collection.AllowsDuplicates;

    // Assert
    Assert.That(allowsDuplicates, Is.EqualTo(AllowsDuplicates));
}
```

Listing 5.12: A trivial test that verifies that the property returns the same value as specified in the tests.

5.3.4 Logic in Tests

It is often recommended that tests contain little to no logic [114, section 7.1.2]. Logic can introduce bugs that are hard to track down without testing the tests themselves. Furthermore, errors in unit tests often turn out to be harder to find, as we tend to look for the mistakes in the system under test and not in the tests.

The use of interface tests in C6 unfortunately results in logic within the tests. If we test an interface's method, it may behave differently depending on the type or state of the collection. What happens when a duplicate item is added to the collection depends on whether the collection is a bag or a set. The method cannot even be called, if the collection is only readable. Writing interface tests therefore often requires us to adapt the tests to the given collection. A simple example of this is demonstrated by the test in listing 5.13, where `Add()` returns different results based on the collection type:

```
1  [Test]
2  public void Add_AddDuplicateItem_AllowsDuplicates() {
3      // Arrange
4      var items = GetUppercaseStrings(Random);
5      var collection = GetExtensible(items, CaseInsensitiveStringComparer.Default);
6      var duplicateItem = items.Choose(Random).ToLower();
7
8      // Act
9      var result = collection.Add(duplicateItem);
10
11     // Assert
12     Assert.That(result, Is.EqualTo(AllowsDuplicates));
13 }
```

Listing 5.13: The test expects a different result based on the value of `AllowsDuplicates`.

5.3.5 Guarding Against Missing Tests

Though the idea is that the interface tests should work for any collection type, the tests in C6 have currently only been run on `ArrayList<T>`. To ensure that we do not forget to implement the missing tests for other collection types, the interface tests also include tests that fail, if the particular collection type is not covered:

```
1  [Test]
2  [Category("Unfinished")]
3  public void Add_FixedSizeCollection_Fail() {
4      Assert.That(IsFixedSize, Is.False, "Tests have not been written yet");
5  }
6
7  [Test]
8  [Category("Unfinished")]
9  public void FindDuplicates_DuplicatesByCounting_Fail() {
10     Assert.That(DuplicatesByCounting, Is.False, "Tests have not been written yet");
11 }
12
13 [Test]
14 [Category("Unfinished")]
15 public void RemoveRange_ReadOnlyCollection_Fail() {
16     Assert.That(IsReadOnly, Is.False, "Tests have not been written yet");
17 }
```

Listing 5.14: The tests fail if the collection type has not been properly tested yet.

5.4 Challenges with Testing in C6

5.4.1 Preconditions

Preconditions – and code contracts in general – are part of the specification of a program. It is natural to want to test that an implementation fulfils its specifications, but Code Contracts does not allow us to catch the contract exception thrown when a contract is violated (see section 5.4.1.1 below). Though Code Contracts would like us not to catch the exceptions, there are reason as to why we would like to do it anyway (an interesting discussion about this can be found among the comments on a StackOverflow question by Shaun Finglas [124]).

One reason is to ensure that the preconditions are correctly added by the rewriter, where we expect them to be. Contract inheritance can be tricky (see section 2.7.4), and it can be difficult to know if the contracts were correctly added without explicitly testing it. If preconditions are not tested, wrong input might not cause problems immediately and only manifest themselves later, when seemingly unrelated code is run.

Another reason is to ensure that a precondition correctly catches the invalid input. Contracts can easily become complicated, especially if they contain logic, and off-by-one mistakes are not uncommon here either (see section 7.2.1). Testing the preconditions is the best way to catch such errors.

Finally, we might like to be sure that a contract is not removed, e.g. in the case someone thought it was redundant. It could go unnoticed, if we do not test that.

5.4.1.1 Testing Preconditions with NUnit

Testing preconditions is slightly more difficult than one might expect. The problem is that Code Contracts' `ContractException` cannot be caught explicitly, because the class is private. We might want to write a simple test that checks whether a violated precondition throws the exception:

```
1  [Test]
2  public void CopyTo_ArrayIsNull_ViolatesPrecondition() {
3      // Arrange
4      var list = new ArrayList<int>();
5
6      try {
7          // Act
8          list.CopyTo(null, 0);
9      }
10     // Assert
11     catch (System.Diagnostics.Contracts.ContractException) {
12         Assert.Pass();
13     }
14     Assert.Fail();
15 }
```

Listing 5.15: The test will not compile because the exception class is inaccessible.

However, the compiler will give us a warning, long before we get to run the test:

```
'ContractException' is inaccessible due to its protection level.
```

Listing 5.16: The compiler warning when we try to compile listing 5.15.

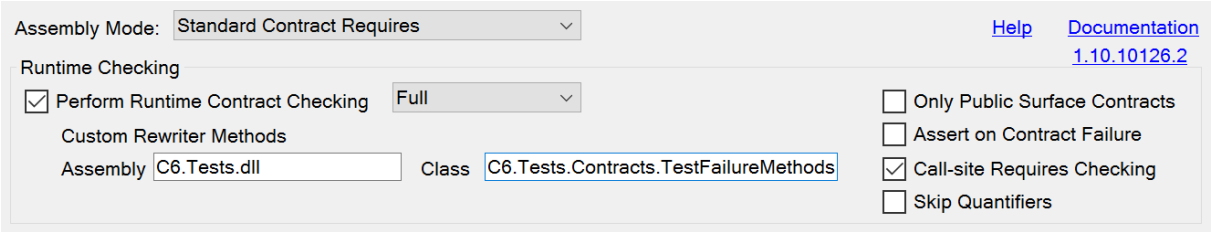


Figure 5.2: The Custom Rewriter Methods settings in `C6.Tests`.

The contract exception is inaccessible for a very good reason: a contract exception should never be caught in production code as the exception indicates a bug in the program from which there is no way to recover. Though sensible, it does make it more difficult to test preconditions.

Luckily, there are several ways to overcome this problem. Jon Skeet suggests the following solution [124] (here slightly modified to match the example above and take advantage of C# 6.0's `when` keyword):

```
1  const string ContractExceptionName =
       "System.Diagnostics.Contracts.__ContractsRuntime+ContractException";
2
3  [Test]
4  public void CopyTo_ArrayIsNull_ViolatesPrecondition() {
5      // Arrange
6      var list = new ArrayList<int>();
7
8      try {
9          // Act
10         list.CopyTo(null, 0);
11     }
12     // Assert
13     catch (Exception e) when (e.GetType().FullName.Equals(ContractExceptionName)) {
14         Assert.Pass();
15     }
16     Assert.Fail();
17 }
```

Listing 5.17: A possible but unattractive solution for testing preconditions.

Though this solution works, it is brittle and not remarkably pretty. It relies on the `FullName` string, which could change depending on platform and software version [137]. Furthermore, it contains a lot of boilerplate code, which requires the reader to analyse the test code to figure out its actual intent. Refactoring the code into a utility method could help, but a better solution would be much appreciated.

The Code Contracts User Manual [56, section 7.8] describes some better solutions. One solution uses the `Contract.ContractFailed` event, which allows the test fixture to be notified when a contract is violated, and to handle it in a sensible way. This does require the test fixture to register a delegate each time a precondition is tested and seems to have to rely on shared memory to tell the test that a contract failure occurred.

The probably most elegant solution is, however, to use a custom contract runtime class [56, section 7.7]. It works by providing the Code Contracts rewriter with a custom class that is used when the assembly is rewritten. For the test assembly, we simply specify which contract runtime class the rewriter should use, as shown in figure 5.2, and activate *Call-site Requires Checking*. The contract runtime class is then called each time a contract

method is invoked and the *Call-site Requires Checking* ensures that `Contract.Requires()` is handled by the custom class and not the assembly containing the precondition. Our custom runtime class is presented in listing 5.18 and only contains the two methods to handle standard and typed preconditions.

```
public static class TestFailureMethods {
    public static void Requires(bool condition, string userMessage, string conditionText) {
        if (!condition) {
            throw new PreconditionException(userMessage, conditionText);
        }
    }

    public static void Requires<TException>(bool condition, string userMessage, string
        conditionText) where TException : Exception {
        if (!condition) {
            throw new PreconditionException(userMessage, conditionText, typeof(TException));
        }
    }
}
```

Listing 5.18: `TestFailureMethods` *throws custom* `PreconditionException`*s instead of contract exceptions.*

When a precondition is checked, we assert that the condition is satisfied, and if not throw a custom `PreconditionException`, which can then be caught in the tests:

```
[Test]
public void CopyTo_ArrayIsNull_ViolatesPrecondition() {
    // Arrange
    var list = new ArrayList<int>();

    // Act & Assert
    Assert.That(() => list.CopyTo(null, 0), Throws.TypeOf<PreconditionException>());
}
```

Listing 5.19: The custom `PreconditionException` *is caught by NUnit.*

We further introduce the utility class `Violates`:

```
public static class Violates {
    public static ExactTypeConstraint Precondition => Throws.TypeOf<PreconditionException>();
}
```

Listing 5.20: `Violates.Precondition` *wraps the NUnit constraint to help readability.*

This allows us to write our tests in an even more readable manner, leaving only the most necessary code there:

```
[Test]
public void CopyTo_ArrayIsNull_ViolatesPrecondition() {
    // Arrange
    var list = new ArrayList<int>();

    // Act & Assert
    Assert.That(() => list.CopyTo(null, 0), Violates.Precondition);
}
```

Listing 5.21: The test uses `Violates.Precondition` *to make the test from listing 5.19 more readable.*

Listing 5.21 is indubitably more readable than listing 5.17 and completely removes all boilerplate code. To improve the tests, we also assert that the right precondition is violated. `Violates` therefore also contains a static method for asserting that the user message is correct:

```
1 [Test]
2 public void CopyTo_ArrayIsNull_ViolatesPrecondition() {
3     // Arrange
4     var list = new ArrayList<int>();
5
6     // Act & Assert
7     Assert.That(() => list.CopyTo(null, 0), Violates.PreconditionSaying("Argument must be
          non-null"));
8 }
```

Listing 5.22: The test asserts that the precondition exception has the correct user message.

To avoid typos in the user messages, C6 uses the `ContractMessage` class to store all precondition user messages instead of the inline messages as shown here.

Unfortunately, Code Contracts fails to use the provided runtime class in certain situations [42], so `Violates` also provides a method to catch these "uncaught" preconditions.

5.4.2 Events

Ensuring that a collection properly raises the expected events requires extensive unit testing as Code Contracts does not support contracts on events – or even delegates. This is not without its own set of challenges as C6 raises many different kinds of events for many different kinds of collections.

The event order for one is in many cases underspecified in both C5 and C6: it is always required that `CollectionChanged` is the last event raised, but the order of the other events may be unknown. Take for instance `RetainRange()`, which removes items from a collection if the items are not in a specified enumerable. The order in which the items are removed from the collection is unknown in general and depends on the specific implementation, e.g. a hash set will remove items depending on the internal order, which depends on things like the size of the internal hash table, the item insertion order and the items' hash function. We can therefore only assert that the set of events raised is equal to the actual events raised. To make it even more complicated, `IExtensible<T>`'s `DuplicatesByCounting` can also affect how events are raised. Instead of raising the same `ItemsRemoved` event for each duplicate of an item, a collection like `HashBag<T>` may just raise a single event with the appropriate item count.

A related challenge is that events may be raised pairwise: methods like `IList<T>`'s `InsertRange()` will raise both `ItemInserted` and `ItemsAdded` for each item. A simple set will not be enough, if we wish to ensure that the two events are raised consecutively.

And last but not least is item equality. When an event argument contains an item, we prefer to ensure that the item is the same reference item, e.g. when removing an item at a specific index. This is, however, not always possible, for instance when removing an item from a collection with duplicates. If the item has duplicates, we do not know which instance is removed, and we can only compare equality and not identicality.

5.4.2.1 Testing Events with NUnit

Taking all the above scenarios into account is a *big* task. Instead of trying to solve them all at once, I have chosen to use this project to lay the foundation for event testing in C6. The test project can then be extended to address the different problems in later versions.

It should be noted that we in many cases can predict both the specific event order and item(s) without problems; at least for `ArrayList<T>`. I have therefore chosen a simple solution, that eliminates as much boilerplate code from the tests as possible, but at the same time improves on C5's event tests. C6 uses the same testing strategy as C5 for event testing: each event test provides a list of expected events which is then compared to the actual events raised by the collection. Listing 5.23 shows a typical event test in C6, and listing 5.24 shows the C5 equivalent, which has been formatted to match the C6 test.

```
1  [Test]
2  public void Add_AddItem_RaisesExpectedEvents() {
3      // Arrange
4      var items = GetUppercaseStrings(Random);
5      var collection = GetExtensible(items);
6      var item = GetLowercaseString(Random);
7
8      var expectedEvents = new[] {
9          Added(item, 1, collection),
10         Changed(collection)
11     };
12
13     // Act & Assert
14     Assert.That(() => collection.Add(item), Raises(expectedEvents).For(collection));
15 }
```

Listing 5.23: A typical event test in C6, which asserts that the collection raises the specified events.

```
1  [Test]
2  public void Add() {
3      // Arrange
4      var items = GetUppercaseStrings(Random);
5      var collection = GetExtensible(items);
6      var item = GetLowercaseString(Random);
7
8      var expectedEvents = new[] {
9          new CollectionEvent<string>(EventTypeEnum.Added,
               new ItemCountEventArgs<string>(item, 1), collection),
10         new CollectionEvent<string>(EventTypeEnum.Changed, new EventArgs(), collection)
11     };
12
13     // Act
14     listen(collection);
15     collection.Add(item);
16
17     // Assert
18     seen.Check(expectedEvents);
19 }
```

Listing 5.24: An event test equivalent to the test in listing 5.23, but written as it was in C5.

There are two main differences between the event tests in C5 and C6. The inline construction of the two `CollectionEvent<T>`s in C5 in lines 9 and 10 in listing 5.24 is hidden away by using the new `CollectionEvent` class in C6, as seen in lines 9 and 10 in listing 5.23. The class is statically imported to allow us to call its methods `Added()`, `Changed()`, `Cleared()`, `Inserted()`, `Removed()`, and `RemovedAt()` without the class name.

Both C5 and C6 uses the `CollectionEvent<T>` class to hold event information such as the event type, the event argument and the sender, which allows us to easily compare expected and raised events.

In C5, the `listen()` method in line 14 (which contained a bug [38]) is responsible for assigning event listeners to all the listenable events of the collection. The tested method is then called on the next line and the raised events are asserted a couple of lines further down with a call to `seen.Check()`. This requires the test writer to properly set up listeners and check them later on.

C6 instead introduces a new `CollectionEventConstraint<T>` (here created using the `Raises()` method on `TestHelper`), which uses a delegate to wrap the tested method call. The constraint class will first assign the event listeners (similarly to how `listen()` does it), then call the delegate, and finally assert that the raised events match the expected ones (similarly to `seen.Check()`). This gives an easily readable assertion without any boilerplate code.

`TestHelper` also contains methods for improving readability of tests that only raise the `CollectionChanged` event or no events at all, as seen in listings 5.25 and 5.26. In the future, `TestHelper` can easily be extended to handle more of the tricky scenarios, just as `CollectionEventConstraint<T>` can be extended to handle the list of expected events differently depending on the collection type.

```
1  [Test]
2  public void Shuffle_RandomCollection_RaisesExpectedEvents() {
3      // Arrange
4      var collection = GetStringList(Random);
5
6      // Act & Assert
7      Assert.That(() => collection.Shuffle(), RaisesCollectionChangedEventFor(collection));
8  }
```

Listing 5.25: An event test asserting that the collection only raises a `CollectionChanged` *event.*

```
1  [Test]
2  public void AddRange_AddEmptyEnumerable_RaisesNoEvents() {
3      // Arrange
4      var collection = GetStringExtensible(Random, ReferenceEqualityComparer);
5      var empty = Enumerable.Empty<string>();
6
7      // Act & Assert
8      Assert.That(() => collection.AddRange(empty), RaisesNoEventsFor(collection));
9  }
```

Listing 5.26: An event test asserting that the collection raises no events when nothing changes.

5.4.3 Returned Collection Values

As described in section 4.3.4, we can use both code contracts and unit tests to ensure that collection values returned from collection queries are correct. This section describes how we use NUnit to test the query results.

We would like to only have a single assertion in each unit test (see section 5.1.3.3), but it can seem a bit limiting, when all the assertions test different aspects of the same object. Consider this (incomplete) test that checks a backwards directed collection value of an empty collection:

```
1  [Test]
2  public void Backwards_EmptyCollection_Empty()
3  {
4      // Arrange
5      var collection = GetEmptySequence<string>();
6
7      // Act
8      var backwards = collection.Backwards();
9
10     // Assert
11     Assert.That(backwards, Is.Empty);
12     Assert.That(backwards.Direction, Is.EqualTo(Backwards));
13     Assert.That(backwards.IsEmpty, Is.True);
14     Assert.That(backwards.Count, Is.Zero);
15     // Remaining tests...
16 }
```

Listing 5.27: The test checks each member of the returned collection value explicitly.

All the assertions here must be repeated for each new test scenario and for each method returning a collection value. The problem can instead be solved much more elegantly using a reference object [114, section 7.2.6]. A reference object equal to the expected collection value is constructed and compared to the actual result:

```
1  [Test]
2  public void Backwards_EmptyCollection_Expected()
3  {
4      // Arrange
5      var collection = GetEmptySequence<string>();
6      var expected = new ExpectedDirectedCollectionValue<string>(
7          items: Enumerable.Empty<string>(),
8          equalityComparer: ReferenceEqualityComparer,
9          allowsNull: collection.AllowsNull,
10         direction: Backwards
11     );
12
13     // Act
14     var backwards = collection.Backwards();
15
16     // Assert
17     Assert.That(backwards, Is.EqualTo(expected));
18 }
```

Listing 5.28: The reference object is created at lines 6 to 11 and all its members are compared to the actual collection value in line 17 using `ExpectedDirectedCollectionValue`*'s own* `Equals()` *method.*

The `EqualConstraint` in line 17 will automatically check whether the *expected* object implements `IEquatable<TActual>` where `TActual` is the type of the *actual* object [106]. The reference object `ExpectedDirectedCollectionValue<T>` therefore implements the `IEquatable<IDirectedCollectionValue<T>>` interface, whose `Equals()` method is used to compare all the members of the expected object with the actual one. This leaves all the checks in one single place, where new can be added and automatically apply to all test methods.

C6 also contains an `ExpectedCollectionValue<T>` class for normal collection values, which the directed version inherits from.

The collection value tests revealed two errors in `NUnitEqualityComparer` [46][47]. I fixed the first bug so it could be used in C6. Since the update has yet to be released, C6 uses a patched version of NUnit, instead of the one available on NuGet [103].

Chapter 6

Implementing ArrayList in C6

One of the goals of this project was to port C5's `ArrayList<T>` to C6. Though much of the code is essentially the same, many changes were made to the data structure to make it work properly with C6's new interfaces and code contracts. This chapter looks closer at some of those changes and the difficulties encountered while porting the data structure.

6.1 Implementation Strategy

Porting `C5.ArrayList<T>` to C6 is on paper a trivial task: just copy the class from C5 and paste it into C6. In reality, it is a bit more complex. `C5.ArrayList<T>` is not a simple class; as with the extensive interface hierarchy, C5 uses extensive inheritance when it comes to implementations. In C5, `ArrayList<T>` inherits from `ArrayBase<T>`, `SequencedBase<T>`, `DirectedCollectionBase<T>`, `CollectionBase<T>`, `CollectionValueBase<T>`, and finally `EnumerableBase<T>`. This makes it rather difficult to figure out where the implementation of a method is.

To better get an idea of how it works, I first made `C5.ArrayList<T>` a single, non-inheriting class that directly implemented all the methods previously inherited. My strategy to port `ArrayList<T>` from C5 to C6 was not to copy the code directly; this would simply not be possible due to the many changes made to the interfaces in C6. Simply copying it would also make it more difficult to ensure that things worked as intended. I took a rather simple approach instead: I started from the top and worked my way down. The idea was to start with the highest, least dependant interface, and implement that fully before moving down to the next interface in line. This means first creating an `ArrayList<T>` that implemented `IEnumerable<T>`. Once implemented and tested, I would move on to `ICollectionValue<T>`, `IListenable<T>`, and so on until all interfaces required for `IList<T>` had been implemented.

Instead of making premature design decision, I implemented `ArrayList<T>` as a single class. Without other data structures, it is difficult to create reasonable base classes and C6's `ArrayList<T>` therefore only inherits from `CollectionValueBase<T>`, which contains the little functionality shared between `ArrayList<T>` and the collection value classes returned from collection queries. Once finished, C6 is likely to contain a base class hierarchy similar to that of C5.

6.2 Constructors

The interface tests in C6 require us to construct data structures that already contain items. We cannot add new items to a collection value, so creating a collection from an enumerable of items is essential. Besides allowing us to decide whether the collection

allows `null` items, `C6.ArrayList<T>`'s constructor now also accepts an enumerable of items. This matches the constructors in `SCG.List<T>` [98].

6.3 Expression-Bodied Members

A nice addition to C# 6.0 is expression-bodied members. Though statement-bodied members (listing 6.1) might be simple, they have a lot of boilerplate code. The expression-bodied versions (listing 6.2) leave only the implementation-specific code and can therefore greatly shorten code for one-line members. There should be no performance penalty by using expression body methods, as they compile to *nearly* the same IL code [87].

```
1  // Normal statement-bodied getter property
2  public bool AllowsDuplicates
3  {
4      get { return true; }
5  }
6
7  // Normal statement-bodied method
8  public T Choose()
9  {
10     return _array[Count - 1];
11 }
```

Listing 6.1: Statement-bodied members

```
1  // Expression-bodied getter property
2  public bool AllowsDuplicates => true;
3
4  // Expression-bodied method
5  public T Choose() => _array[Count - 1];
```

Listing 6.2: Expression-bodied members

This is even more beneficial when working with contracted interfaces. Compare the C5 implementation of `ArrayList<T>.First` presented in listing 6.3 with the corresponding implementation in C6 presented in listing 6.4:

```
1  public T First {
2      get {
3          if (_count == 0) {
4              throw new
                   NoSuchItemException();
5          }
6
7          return _array[_offset];
8      }
9  }
```

Listing 6.3: C5's `ArrayList<T>.First`

```
1  public T First => _items[0];
```

Listing 6.4: C6's `ArrayList<T>.First`

The parameter validation in C5 is explicit, whereas in C6 the validation is handled by the interface contracts. Here Code Contracts together with C# 6.0 reduce nine lines of code to one. Many of `ArrayList<T>`'s properties and methods that are redirecting calls to other methods, like the explicit implementations from legacy interfaces or the overloaded methods, are implemented using expression bodies.

Expression bodies must be one-liners, so adding extra contracts is simply not possible without expanding it first. Fortunately, most public methods in collection classes are implementations of interface methods, so it is mostly private members and constructors that require their own contracts.

6.4 Removing Items in Bulk

Several methods on `ArrayList<T>` require us to enumerate the collection and remove items based on a given condition. These methods currently include `ICollection<T>`'s `RemoveDuplicates()`, `RemoveRange()`, and `RetainRange()`. To avoid code duplication, they all use the same private helper method `RemoveAllWhere()`. The method takes a predicate which takes an item. When enumerating the collection, an item is removed, if the predicate returns `true` for the given item.

`RemoveAllWhere()` is currently implemented using a simple for loop that starts from the beginning of the list and moves items forwards if gaps arise from deleted items. This method could potentially be done with bulk move operations, possibly increasing the performance of all methods that use `RemoveAllWhere()`.

Using the helper method allows us to implement the aforementioned methods with very few lines of code, as seen in listing 6.5. In reality, the methods contain more checks to improve performance when dealing with empty collections/enuemrables.

```
1  public bool RemoveDuplicates(T item) => RemoveAllWhere(x => Equals(item, x));
2
3  public bool RemoveRange(IEnumerable<T> items)
4  {
5      var itemsToRemove = new HashBag<T>(items, EqualityComparer, AllowsNull);
6      return RemoveAllWhere(item => itemsToRemove.Remove(item));
7  }
8
9  public bool RetainRange(IEnumerable<T> items)
10 {
11     var itemsToRemove = new HashBag<T>(items, EqualityComparer, AllowsNull);
12     return RemoveAllWhere(item => !itemsToRemove.Remove(item));
13 }
```

Listing 6.5: Methods in `C6.ArrayList<T>` *that use the private method* `RemoveAllWhere()` *to shorten code.*

Notice, that the examples in listing 6.5 here use `HashBag<T>` (as they should). `HashBag<T>` is, however, not implemented in C6 yet, so the current implementation uses `ArrayList<T>` instead. This is in no way optimal and is likely to result in quadratic running times. This will be changed when `HashBag<T>` is introduced.

6.5 Using Array's Utility Methods

`C6.ArrayList<T>` extensively uses the static utility methods found on `Array` [94] to manipulate arrays instead of implementing trivial methods directly in C6. `Array` is used both to copy and resize arrays, clear ranges of an array, and reverse or sort its elements. Using `Array` has the potential of speeding up these operations, because the methods can be implemented as optimized C/C++ code in the CLR [143].

6.6 List Resizing

`ArrayList<T>` stems from a need to have resizable arrays in a world where only fixed-size arrays exist. It is therefore important that `ArrayList<T>` properly handles the resizing of the internal array.

6.6.1 C5 Bug: Overflow When Resizing Arrays

I found (and fixed) a subtle bug in `ArrayBase<T>` that would cause an infinite loop when adding very large enumerables to array-based collections like `ArrayList<T>` [39]:

```
1 var arrayList = new ArrayList<bool>();
2 var items = new bool[(1 << 30) + 1]; // 2^30 + 1 = 1,073,741,825
3 arrayList.AddAll(items);
```

Listing 6.6: Code example that would trigger an infinite loop in C5.

When adding the large array to the `ArrayList<T>`, which inherits from `ArrayBase<T>`, the collection first has to allocate an inner array big enough to contain all the enumerable's items. Listing 6.7 shows the old version of the `expand()` method [33] in `ArrayBase<T>` (here slightly reformatted). `expand()` is used to enlarge the underlying array.

```
1 protected virtual void expand(int capacity, int count) {
2     int length = _items.Length;
3     while (length < capacity) {
4         length *= 2;
5     }
6     var array = new T[length];
7     Array.Copy(_items, array, count);
8     _items = array;
9 }
```

Listing 6.7: The old `expand()` *method in* `C5.ArrayBase<T>`*, which caused an infinite loop in certain cases.*

`ArrayBase<T>` stores its items in the `_items` array, whose length is always a positive power of two. The method is called with the required capacity and the collection's current count – in the example above, `capacity` is $2^{30} + 1$ and `count` is 0.

The old method would take the current array length and double it (line 4) until it was no longer less than the required capacity (line 3). The problem arose when the integer `length` was doubled for too long: when `length` reached 2^{30} it would still be less than `capacity` ($2^{30} + 1$), requiring us to double it again, giving an expected value of 2^{31}. The integer range in C# [61], however, is $-2,147,483,648$ (-2^{31}) to $2,147,483,647$ ($2^{31} - 1$), which caused `length` to overflow. Doubling the integer again made it zero. The loop condition in line 3 would now always be true, since `length` would remain zero even when doubled.

6.6.1.1 Fix in C5

The easiest fix is wrapping the multiplication in line 4 in `checked`:

```
1 protected virtual void expand(int capacity, int count) {
2     int length = items.Length;
3     while (length < capacity)
4         length = checked(length * 2); // Throws a run-time exception if length * 2 overflows
5     }
6     var array = new T[length];
7     Array.Copy(items, array, count);
8     items = array;
9 }
```

Listing 6.8: The fix used in C5 to avoid the infinite loop by instead getting an exception at runtime.

When the multiplication causes an overflow, an `OverflowException` is thrown. This does not solve the problem, but it does ensure that the problem does not go unnoticed.

One could ask whether that is not good enough. It seems that the bug is actually rather difficult to "provoke" without getting an out-of-memory exception first. As described in documentation of `System.Array` by Microsoft [64], the highest index for an array in C# is not even the maximum integer value, but rather the seemingly arbitrary value of `0x7feffffff` (2,146,435,071). Furthermore, the maximum size of an `Array` is 2 gigabytes (GB) (though this restriction can be avoided on 64-bit systems), which means that arrays with lengths even less than 2^{30} could cause problems.

6.6.2 Capacity

To avoid this problem, `C6.ArrayList<T>` uses the same approach as `SCG.List<T>`. Each method that adds items, will first call the `EnsureCapacity()` method to ensure that the inner array has the required capacity:

```
1  private void EnsureCapacity(int requiredCapacity)
2  {
3      if (Capacity >= requiredCapacity) return;
4
5      var capacity = IsEmpty ? MinArrayLength : Capacity * 2;
6      if ((uint) capacity > MaxArrayLength) {
7          capacity = MaxArrayLength;
8      }
9      if (capacity < requiredCapacity) {
10         capacity = requiredCapacity;
11     }
12     Capacity = capacity;
13 }
```

Listing 6.9: C6 uses `EnsureCapacity()` *instead of* `expand()` *to resize the inner array.*

The method will try to double the current capacity (line 5). If this is larger than largest possible array size (line 6), the capacity is set to that (line 7). If the new capacity is too small (line 9), it is set to the required capacity (line 10). Setting the `Capacity` property (line 12) will create a new array, copy the items from the old array, and store the new array instead. If the required capacity is larger than the largest possible array size, calls inside the property throws an `OutOfMemoryException`. The `Capacity` property is made public to allow users to directly set the size of the underlying array.

6.7 Minor Optimizations

`ArrayList<T>` contains minor optimizations inspired by `SCG.List<T>`. One of them is to cast an `IEnumerable<T>` into a more information-rich interface such as `ICollection<T>` whenever possible. Since most collections in C# implement `IEnumerable<T>`, so the interface is used whenever methods accept a collection of items. C6 will first try to cast an enumerable to `ICollectionValue<T>`, which is useful because the interface exposes members like `Count`, `CopyTo()` and `ToArray()`. If the enumerable is not a C6 collection, it will try to cast it to `SCG.ICollection<T>` instead, which exposes both `Count` and `CopyTo()`. In all other cases, the methods will just use the `IEnumerable<T>` directly.

To avoid the overhead of the collection's equality comparer, certain operations are optimized when searching for `null` references. Instead of calling the equality comparer, a direct comparison with `null` is performed, for instance in `IIndexed<T>`'s `IndexOf()`:

```
[Pure]
public int IndexOf(T item) {
    if (item == null) {
        for (var i = 0; i < Count; i++) {
            // Explicitly check against null to avoid using the (slower) equality comparer
            if (_items[i] == null) {
                return i;
            }
        }
    }
    else {
        for (var i = 0; i < Count; i++) {
            if (Equals(item, _items[i])) {
                return i;
            }
        }
    }
    return ~Count;
}
```

Listing 6.10: Explicitly check against `null` to avoid using the equality comparer for `null` reference types.

6.8 Collection Values

`ArrayList<T>` contains five methods that return collection values. The first two are `Backwards()` and `GetIndexRange()`, which both are implemented using the private class `Range`, also in C5. `Range` basically contains a start and end index and a reference to its parent list, and is able to enumerate a range of indices in either direction lazily. `Range` does not need to cache any items.

The `FindDuplicates()` uses the private `Duplicates` class, which keeps a reference to the searched item and lazily enumerates the list to find the next item equal to it. Each new item found is stored in an internal bag (a new `ArrayList<T>`) so the items can be enumerated again, if needed. `UniqueItems()` works in a similar way, but uses a set to store items. The enumerator lazily looks for the next item that is successfully added to the set and returns it.

`ItemMultiplicities()` requires a `HashBag<T>` to be properly implemented, which is why the method has been left unimplemented for now.

6.9 Check and Invariants

C5 makes an attempt to get an object invariant that can verify a collection's internal state during testing. An object invariant is checked through the interface member `IExtensible<T>.Check()` that *"performs a comprehensive integrity check of the collection's internal representation. [It is] relevant only for library developers"* [31, section 4.5]. The idea is that developers manually call the method during testing to ensure that everything is as intended: a return value of `true` signals a valid inner state; `false` means

something went wrong (the problem is further logged using `Logger.Log()`). The following code is an excerpt from `C5.ArrayList<T>.Check()`:

```
public bool Check() {
    if (Count > Capacity) {
        Logger.Log($"Bad size ({Count}) > array.Length ({Capacity})");
        return false;
    }

    for (var i = 0; i < Count; i++) {
        if (_items[i] == null) {
            Logger.Log($"Bad element: null at index {i}");
            return false;
        }
    }

    //...
}
```

Listing 6.11: The C5 version of an object invariant on `ArrayList<T>`.

This approach has two problems: it exposes a completely irrelevant member to users of the library and it relies on the developer to remember to call the method. We can solve the first problem by wrapping the method in an `#if DEBUG` directive to remove it from release versions of the assembly. The second requires Code Contracts: for each implementation of `IExtensible<T>`, we move the code from `Check()` into an object invariant method annotated with `ContractInvariantMethod` (see section 2.4) and convert the checks into calls to `Contract.Invariant()`:

```
[ContractInvariantMethod]
private void ObjectInvariant()
{
    Invariant(Count <= Capacity);
    Invariant(AllowsNull || ForAll(this, item => item != null));
    //...
}
```

Listing 6.12: The object invariants on `C6.ArrayList<T>` *matching* `Check()` *in listing 6.11.*

Notice how much shorter the code is. We no longer need to log which invariant failed, since it will throw an exception and stop execution. Just as with pre- and postconditions, the checks changes from stating what we do not want to what we actually want, making it slightly easier to read.

6.10 Stacks and Queues

`C5.ArrayList<T>` implements three interfaces: `IList<T>`, `IStack<T>`, and `IQueue<T>`. There is no doubt that the list should implement `IList<T>`, and implementing `IStack<T>` also makes a lot of sense. However, implementing `IQueue<T>` seems a bit misguiding. It is definitely true that an array list can behave like a queue, but it is in no way efficient. When calling `IQueue<T>`'s `Enqueue()`, `ArrayList<T>` can easily add items to one end of the inner array, but when calling `IQueue<T>`'s `Dequeue()`, `ArrayList<T>` has to remove the first item in the array and push the remaining items one index towards its beginning. Consequently, `C6.ArrayList<T>` does not implement `IQueue<T>`, however, users are still

able to get the interface's behavior by using `IList<T>.RemoveFirst()` if necessary. An alternative solution would be to implement `IQueue<T>` explicitly in `ArrayList<T>` in order to still provide the functionality, but requiring the collection to be explicitly cast to a queue.

Both `LinkedList<T>` and `CircularQueue<T>` do not care in which end you add or remove items, and they will therefore (still) implement both `IStack<T>` and `IQueue<T>` in C6.

Chapter 7

Bugs in C6

There seems to have been two kinds of bugs in C6: those that were *caught* by code contracts and those that were *caused* by code contracts. This chapter reviews the bugs found during the project.

7.1 Bugs Caught by Code Contracts

The first two bugs caught by Code Contracts were bugs in the unit tests. The other two were implementation errors not caught by the unit tests.

7.1.1 Allowing Null Value Type Values

`ICollectionValue<T>`'s `AllowsNull` can only be `true` for reference types. This is checked by the postcondition on `AllowsNull`, as seen in listing 7.1:

```
1 // Value types must return false
2 Ensures(!typeof(T).IsValueType || !Result<bool>());
```

Listing 7.1: The postcondition ensures that `ICollectionValue<T>.AllowsNull` *is* `false` *for value types.*

The contract caught a bug in a test that tried to set `AllowsNull` to `true` for an integer collection, which is not possible.

7.1.2 Empty Index Range from an Empty Collection

To test the preconditions of `IIndexed<T>.GetIndexRange()`, a test was written that tried to get a range of size zero from an empty collection. The test was initially intended to violate a precondition, but revealed a flawed test premise.

When reviewing the test results before implementing the method in `ArrayList<T>`, I noticed that the test failed with a `NotImplementedException`; I expected it to pass due to the test violating a precondition before the implementation was called. The lack of precondition violation made me review the contracts and the test. The problem was in the expected outcome of the test: taking nothing out of nothing should indeed be possible – just think of the binomial coefficient $\binom{0}{0}$ which says there is exactly one way to do it – and the test did correctly not violate the contracts.

7.1.3 Updating the Internal Version When Shuffling or Sorting

The internal version number described in section 3.5.4.1 should always change when the collection is changed. This causes any of the collection's enumerators or collection values

to be invalidated. Ensuring that the version number is properly updated cannot be done in the interface contracts, because the version is stored in a private field in the class and is not a part of the interface. The following contract was added to all the methods in `ArrayList<T>` that could potentially change the collection:

```
1 // If collection changes, the version is updated
2 Ensures(this.IsSameSequenceAs(OldValue(ToArray())) || _version != OldValue(_version));
```

Listing 7.2: The postcondition ensures that the version number is updated, if the collection changes.

Adding the contract to `ArrayList<T>`'s `Shuffle()` and `Sort()` made several tests fail: the version number was not updated when the collection had more than one item and was changed either by shuffling it or by sorting it when it was unsorted. The postcondition revealed that the unit tests that should check whether the enumerator would break were missing. Once the tests were added, the bug was fixed and both the contracts and the unit tests passed again.

7.1.4 Reversing a List

`C6.ArrayList<T>` uses `SCG.Array` to lower the risk of bugs in trivial, array-related bulk operations (see section 6.5). The original implementation of the method looked as follows:

```
1  public virtual void Reverse() {
2      if (Count <= 1) {
3          return;
4      }
5
6      // Only update version if the collection is actually reversed
7      UpdateVersion();
8
9      Array.Reverse(_items);
10     RaiseForReverse();
11 }
```

Listing 7.3: The original version of `ArrayList<T>.Reverse()`.

It uses `Array.Reverse()` [95] in line 9 to reverse the order of the items in the internal `_items` array. The method on `IList<T>` contains the following postcondition:

```
1 // The collection is reversed
2 Ensures(this.IsSameSequenceAs(OldValue(Enumerable.Reverse(this).ToList())));
```

Listing 7.4: The postcondition uses `Enumerable.Reverse()` to ensure that the result is correct.

Everything worked great while testing, but the postcondition was suddenly violated when I was writing the user guide example for `C6.UserGuideExamples`. The alert reader has probably already realized, that the problem is in line 9 in listing 7.3: `Array.Reverse()` reverses the whole array and not just the part occupied by user items. This works fine as long as the array is full, which unfortunately always was the case in the unit tests. This is because the capacity is set in the constructor to the number of items. The user guide example removes items, which revealed the problem when the half-full array was reversed.

7.2 Bugs Caused by Code Contracts

Where the previous bugs were found by Code Contracts, these bugs were hiding inside the code contracts themselves.

7.2.1 Off-By-One Contract

Verifying why a test fails before implementing the tested functionality is always a good idea. All failing tests should fail because of a `NotImplementedException`. I have discovered several mistakes in the contracts simply by verifying failing tests. One bug was found in `IIndexed<T>`'s `RemoveIndexRange()`, where a count equal to the collection size would wrongly violate the preconditions:

```
1  // Argument must be within bounds (collection must be non-empty)
2  Requires(0 <= startIndex, ArgumentMustBeWithinBounds);
3  Requires(startIndex + count < Count, ArgumentMustBeWithinBounds); // Should be <=, not <
```

The comparison in line 3 should have been a *less-or-equal* comparison to properly include the last item.

7.2.2 Testing Shuffling

Any method that contains some kind of randomness is difficult to test. `C6.IList<T>`'s `Shuffle()` methods shuffle the items in the list with a default/provided random-number generator. The interface methods only contain a single postcondition, which checks that the list still contains the same items as before it was shuffled:

```
1  // The collection remains the same
2  Ensures(this.HasSameAs(OldValue(ToArray())));
```

Listing 7.5: The only postcondition on `C6.IList<T>.Shuffle()`.

Unfortunately, we cannot ensure that the items are in another order:

```
1  // The item order is changed
2  Ensures(!this.IsSameSequenceAs(OldValue(ToArray())));
```

Listing 7.6: A postcondition that would be violated occasionally by a working implementation.

The postcondition would on average fail one in $N!$ times, where N is the collection size, since the shuffle could leave all items in their original place.

In an attempt to at least catch clearly wrong implementations, a test method was added that shuffled a list thrice and checked that the three resulting lists were not equal. If they were, the test would warn that the `Shuffle()` method *likely* contained an error, since the probability of the three shuffles all returning the same order is very low: for 10 items (which is the lowest number of items in a random test collection) the probability is one in 13 trillion ($1/13.168.189.440.000 = 10!^{-2}$).

Though the test might seem silly or trivial, it actually revealed a problem with the extension method `CollectionExtensions.Shuffle()`, which was caused by an erroneous

code contract. The bug was hiding in `ContractHelperExtensions.UnsequenceEqual()`, which (as explained in section 4.1.1) converts two enumerables `first` and `second` into arrays, before sorting them based on their hash code:

```
1  var firstArray = first as T[] ?? first.ToArray();
2  var secondArray = second as T[] ?? second.ToArray();
```

Listing 7.7: Part of the old implementation of `ContractHelperExtensions.UnsequenceEqual()`.

The problem was the cast: if the enumerables were already arrays (which was the case with `Shuffle()`), the original arrays would get sorted. What appeared to be a shuffle, was just a sorting on hash code. The correct code should instead have been:

```
1  var firstArray = first.ToArray();
2  var secondArray = second.ToArray();
```

Listing 7.8: The enumerables are converted into arrays using LINQ even if they already are arrays.

Comparing the two enumerables should of cause not affect either of them, which is why the following contracts were added once the bug was discovered:

```
1  // first remains unchanged
2  Ensures(first == null || first.IsSameSequenceAs(OldValue(first.ToList())));
3
4  // second remains unchanged
5  Ensures(second == null || second.IsSameSequenceAs(OldValue(second.ToList())));
```

Listing 7.9: The two postconditions ensure that the items in the enumerables do not change.

The example demonstrates how a problem with Code Contracts can be solve simply by adding even more code contracts.

7.2.3 Bad Enumerables Are Always Bad

All methods that accept an enumerable have a test ensuring that a bad enumerable (one that throws an exception) does not leave the data structure in an invalid state. Even though the method was not implemented in `ArrayList<T>`, the bad exception was thrown and the collection was unchanged – just as we would expect, had the method been implemented.

The problem was a bit subtle, and the sinner was actually a contract. The following preconditions are found on any method that accepts an enumerable of items:

```
1  // Argument must be non-null
2  Requires(items != null, ArgumentMustBeNonNull);
3
4  // All items must be non-null if collection disallows null values
5  Requires(AllowsNull || ForAll(items, item => item != null), ItemsMustBeNonNull);
```

Listing 7.10: The contracts ensure that the enumerable is non-`null` *and only contains* `null`*s if allowed.*

The collections used in the tests incidentally disallowed `null` items, i.e. `AllowsNull` was `false`. This causes `ForAll()` (the universal quantifier) to enumerate the enumerable and

since none of the items in `items` were `null`, the enumerator finished and threw the bad enumerator exception. The collection did not change as the method was never entered.

From the perspective of the test, everything worked as intended, but the actual implementation was not responsible for that. Furthermore, this test would likely have failed in a release build, where quantifiers/preconditions were disabled.

Chapter 8

Tools that Work with Contracts

Contracts open up new possibilities for tools to assess our code more intelligently. Code Contracts for instance comes with its own static checker, that will analyze the code statically to help us avoid problems at runtime, and Visual Studio 2015 comes with IntelliTest for automatically generating unit tests that can be guided by our contracts. This project has surveyed how tools can help the development of larger projects like C6, and this chapter describes them and my findings.

8.1 Code Contracts' Static Checker

Code Contracts' static checker [56, section 6.6] is probably the first new tool you encounter, when you start working with Code Contracts. The tool is included in the framework and will try to prove a project's explicit code contracts (the assertions, pre- and postconditions and invariants written with Code Contracts, including preconditions in other assemblies and inherited postconditions) at compile time to help avoid contract violations at run time. If a contract cannot be proved, a warning is issued by the checker.

The static checker comes with an overwhelming number of settings, as seen in figure 8.1. The tool can perform its analysis in the background and will for instance try to infer and suggest new contracts.

```
1 [Pure] public static bool IsEmpty<T>(this IEnumerable<T> enumerable) => !enumerable.Any();
```

Listing 8.1: The `IsEmpty()` *extension method that checks whether an enumerable is empty.*

For the `IsEmpty()` extension method in listing 8.1, the static checker will correctly suggest adding a precondition for the enumerable:

```
CodeContracts: Missing precondition in an externally visible method. Consider adding
    Contract.Requires(enumerable != null); for parameter validation.
```

Listing 8.2: Code Contracts suggests a precondition on `IsEmpty()` *from listing 8.1.*

By looking at existing contracts, the tool will even warn you (possibly in a slightly odd way), if it detects violated contracts, e.g. when `IsEmpty()` is called on a `null` enumerable:

```
CodeContracts: Invoking method 'Main' will always lead to an error. If this is wanted,
    consider adding Contract.Requires(false) to document it.
CodeContracts: requires is false: enumerable != null
```

Listing 8.3: Code Contracts' static checker detects that `IsEmpty()` *is called on a* `null` *object in* `Main`.

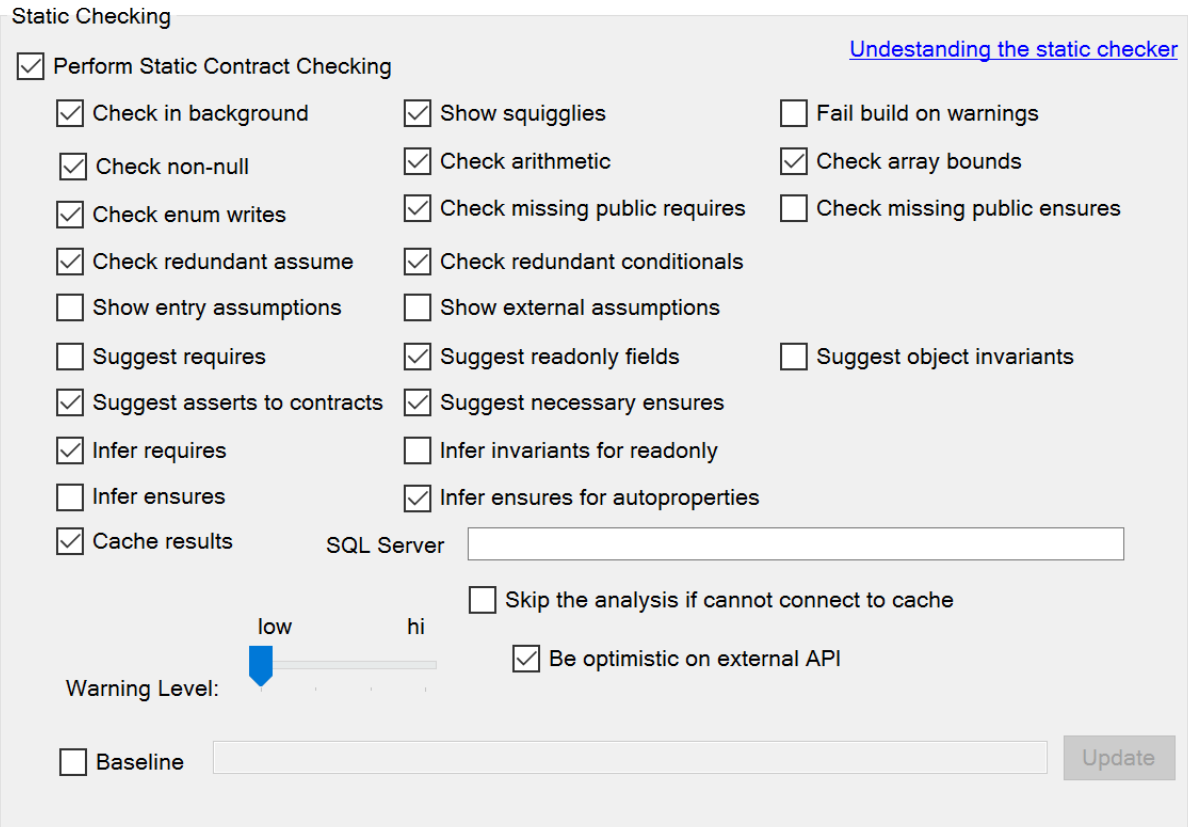


Figure 8.1: The default settings for the static checker found in the Code Contracts framework. Notice the spelling error in the top right corner, which has been there since the framework was first released.

Beside detecting `null` values, the checker will also try to validate arithmetic operations, array index bounds, and enum values. The type of checks can be easily adjusted, as is the case with the warning level.

8.1.1 Working with the Static Checker

It is quite the challenge to work with Code Contracts' static checker, and the people behind the framework are not afraid to admit it:

First, a word of caution: static code checking or verification is a difficult endeavor. It requires a relatively large effort in terms of writing contracts, determining why a particular property cannot be proven, and finding a way to help the checker see the light.

Before you start using the static contract checker in earnest, we suggest you spend enough time using contracts for runtime checking to familiarize yourself with contracts and the benefits they bring in that domain. [...]

If you are still determined to go ahead with contracts [...] create a separate configuration, [to avoid] slowing down regular build flavors.

[56, section 6.6]

These are not exactly words of encouragement, but they do give a good indication of what it is like to work with the verification tool. It often seems to be necessary to nurse

the static checker into realizing that certain things are true, or as the manual puts it: "help the checker see the light". Consider the simple `ItemCountEventArgs` class from C6:

```
1  public class ItemCountEventArgs<T> : EventArgs {
2      [ContractInvariantMethod]
3      private void ObjectInvariant() {
4          Invariant(Count > 0);
5      }
6
7      public ItemCountEventArgs(T item, int count) {
8          Requires(count > 0, ArgumentMustBePositive);
9          Item = item;
10         Count = count;
11     }
12
13     public int Count { get; }
14     public T Item { get; }
15 }
```

Listing 8.4: The `ItemCountEventArgs` from C6, which requires that `Count` is always positive.

The two properties are both auto-implemented, read-only properties that must be assigned during construction. The class' only constructor requires that the parameter `count` is positive and assigns its value to the `Count` property.

Though the example is simple, the static checker is not able to prove the object invariant on `ItemCountEventArgs`. If we do not wish to declare a private setter for `Count` (which would make it writable) or explicitly implement the property with a read-only field, we must add an assumption to the end of the constructor:

```
1  Assume(Count > 0);
```

Listing 8.5: The static checker will just believe our assumption without trying to prove it.

We must therefore explicitly alter the code, just to help static checker see, what appears to be obvious.

8.1.2 Verdict

Unfortunately, the example is much in line with my previous experiences with the static analysis tool. When you start working with the tool, you hope that it will find all your missing contracts and spot your contract-violating method calls. The reality is that most of the issued warnings are related to the checker not being able to prove the postconditions and invariants. Instead of saving you a headache, the tools gives you one, while you try to cater to its needs.

I do consider myself an experienced user of Code Contracts, having used it in all my C# projects the last five years, both academically and professionally. But I must admit, I have yet to find a way that helps facilitates the static verifier, and maybe therefore yet to benefit from using the static checker in Code Contracts in any real-life scenario.

The idea of the tool is absolutely great, but its current capabilities are not much of a benefit once we look at real-life code. This might change in the future [144], but judging by the changes made within the last half decade, I do not have high hopes.

8.2 Visual Studio's IntelliTest

IntelliTest [59] is an automated tool in Visual Studio 2015 that generates test data and suites of unit tests from your code. It will analyse a method and generate different and extreme inputs that together cover as much of the method as possible. The input sets are saved as different unit tests and a test passes if it does not throw an exception; the test's expected result simply becomes the value returned from the method, when the test was generated.

One might naturally ask, if IntelliTest generates the tests from our code, how will it ever be able to discover semantic errors? The short answer: it won't! IntelliTest will try to trigger new execution paths with different values, and as long as the input does not cause an exception to be thrown, IntelliTest will just believe whatever result, the method returns. This is basically the opposite of Test-Driven Development, since we let the code decide the specifications [121]. This can make perfect sense, if we wish to generate tests that can help us refactor previously untested legacy code. So how do we make this useful when writing new code? We use our secret weapon: Code Contracts.

8.2.1 Simple Demonstration

Let us look at a simple code example to better demonstrate IntelliTest's capabilities:

```
1  public static int Min(this int[] array) {
2      var min = 0;
3      for (var i = 0; i < array.Length - 1; i++) {
4          var item = array[i];
5          if (item <= min) {
6              min = item;
7          }
8      }
9      return min;
10 }
```

Listing 8.6: A bug-filled extension method for finding the minimum element of an integer array.

As the name indicates, the method should find the minimum value in an array of integers, but the code is filled with errors. We mistakenly do *not*:

- check whether `array` is `null`.
- require that the array has at least one element.
- ensure that the result comes from the array.
- check the integer at the last index of the array.

Beside these bugs, we could improve the method by starting at the second item in the array, and only check whether integers are *less than* the current minimum. Our goal in this example is to guide IntelliTest to find all these bugs for us, and then use the generated tests to ensure that making the suggested improvements does not break the code.

When we first run IntelliTest on `Min()` it generates seven tests, one of them failing, as seen in figure 8.2.

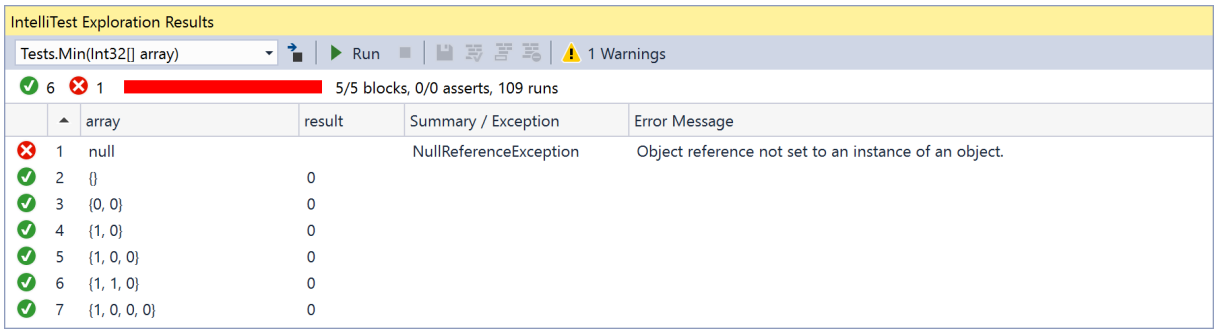


Figure 8.2: IntelliTest generates seven tests, the first fails because `array` *is* `null`.

IntelliTest has tried to input a `null` value, which fails in line 3, when we try to get the array's length. To fix the problem, we simply add a code contract to the method:

```
1 Requires(array != null);
```

Listing 8.7: A precondition requiring that the array may not be `null`.

The generated tests are saved in the test project, but adding a contract will still throw an exception, so we must rerun IntelliTest to generate new test cases. When IntelliTest again tries with a `null` array, the test passes, as seen in figure 8.3, because IntelliTest considers precondition violations acceptable. Notice how IntelliTest generates a new set of tests, now that the method has changed.

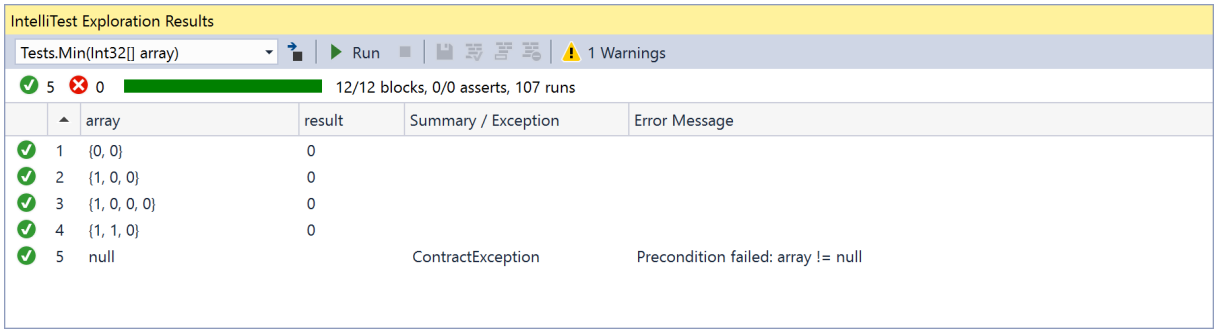


Figure 8.3: With only a single contract, IntelliTest only generates passing tests.

All tests pass, so we are done! Well, not quite. This perfectly illustrates IntelliTest's shortcomings when used alone: no (unexpected) exceptions were thrown, so IntelliTest just believes whatever the method returns. The method still contains several semantic errors. To help IntelliTest, we add another code contract:

```
1 Ensures(ForAll(array, element => Result<int>() <= element));
```

Listing 8.8: A postcondition ensuring that the result is less than or equal to all elements in the array.

The postcondition says that the method's result will be at least as small as all the elements in the array. Notice how this makes no attempt to solve the problem. Our postcondition would not be able to produce the result, only validate it. The postcondition's role is only

to act as a certificate. If we rerun our tests, everything still passes. We need to rerun IntelliTest to generate new test scenarios. The results are shown in figure 8.4.

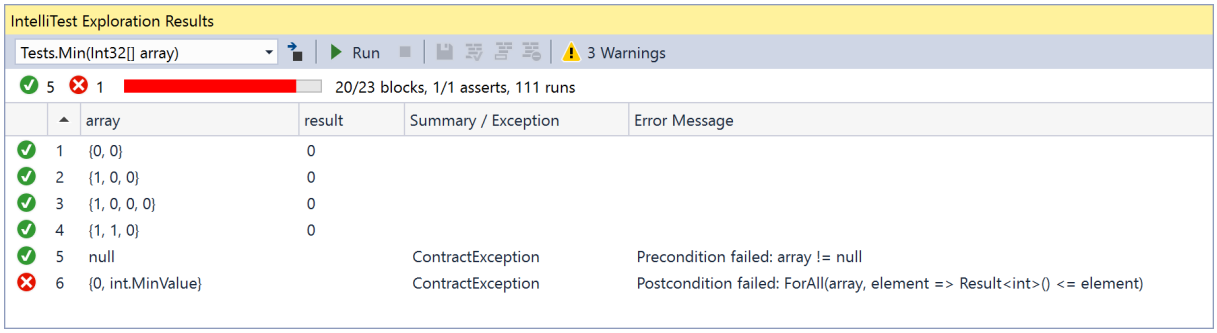


Figure 8.4: The new postcondition helps IntelliTest find a scenario, where the postcondition does not hold.

A violated postcondition is not deemed acceptable, since that would indicate an implementation error. The postcondition has helped IntelliTest generated a test that found a semantic error: we set `min` to an initial value of 0, which only works if the minimum is non-positive.

We correctly replace the initial value with the first in the array (instead of just `int.MinValue`, which would make our tests pass for now):

```
1  public static int Min(this int[] array) {
2      Requires(array != null);
3      Ensures(ForAll(array, element => Result<int>() <= element));
4
5      var min = array[0];
6      for (var i = 0; i < array.Length - 1; i++) {
7          var item = array[i];
8          if (item <= min) {
9              min = item;
10         }
11     }
12     return min;
13 }
```

Listing 8.9: The initial value of `min` *is set in line 5 to be the first element in the array.*

When we once again run IntelliTest we get two new failing tests, as seen in figure 8.5: one is caused by an index-out-of-range exception in line 5, because we do not require that the array contains anything, and the other is when the last element is the minimum.

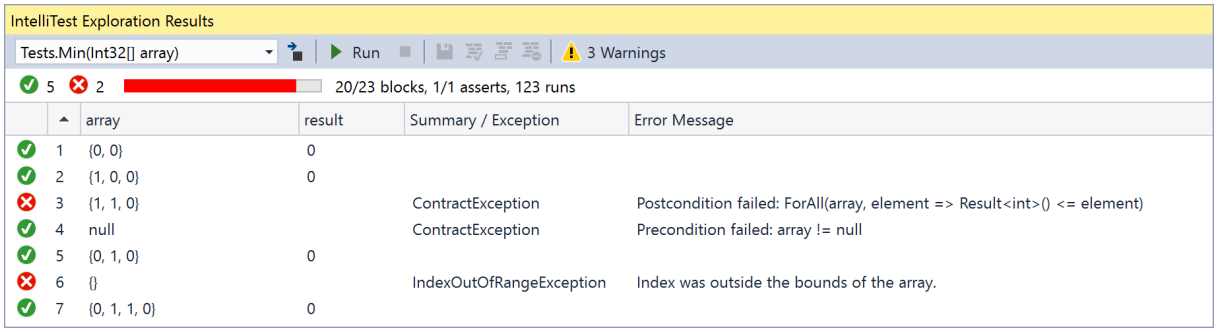


Figure 8.5: IntelliTest generates two new tests that fail when the last item is the minimum and when the array is empty.

We add another contract and fix the loop condition:

```
public static int Min(this int[] array) {
    Requires(array != null);
    Requires(array.Any());
    Ensures(ForAll(array, element => Result<int>() <= element));

    var min = array[0];
    for (var i = 0; i < array.Length; i++) {
        var item = array[i];
        if (item <= min) {
            min = item;
        }
    }
    return min;
}
```

Listing 8.10: The next version of `Min()` *with an extra precondition and the proper loop condition.*

The next run of IntelliTest, seen in figure 8.6, produces no failing tests.

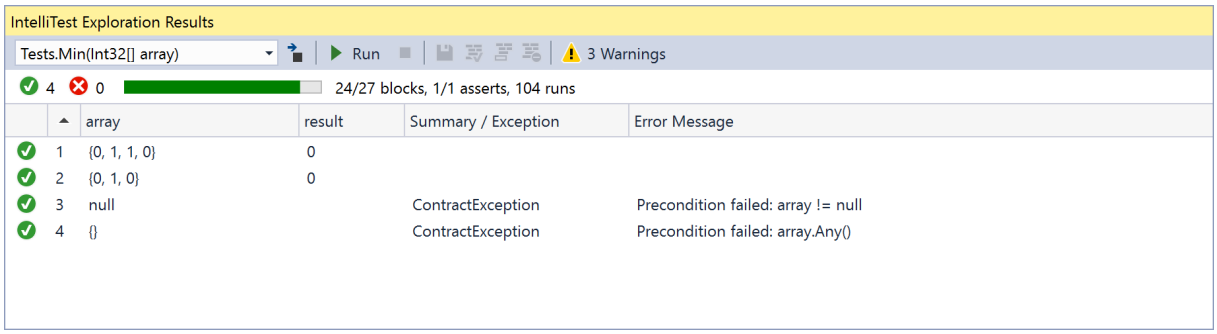


Figure 8.6: IntelliTest's new tests, which input ones and zeros or test the preconditions.

We therefore add the postcondition in listing 8.11 to ensure that the result is equal to one of the values in the array:

```
Ensures(Exists(array, element => Result<int>() == element));
```

Listing 8.11: A postcondition ensuring that the result is one of the element in the array.

The new postcondition ends up generating a slightly more interesting test case, as seen in figure 8.7.

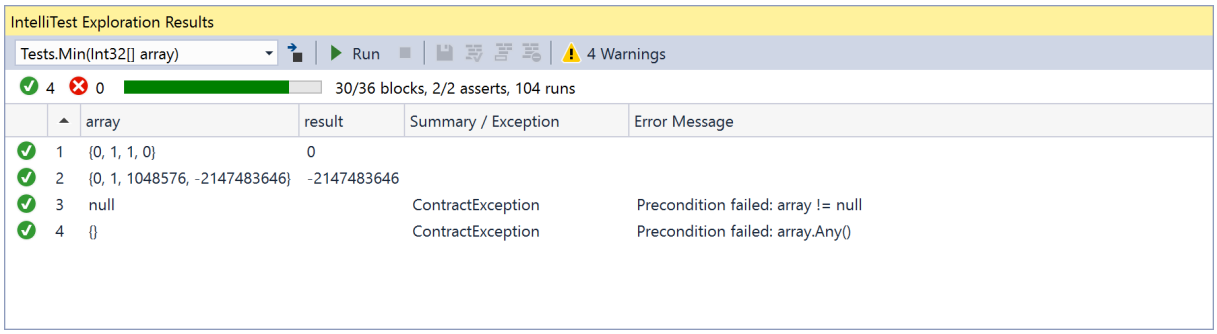


Figure 8.7: The final tests generated by IntelliTest.

We can now use the tests while optimizing the implementation. The final solution looks as follows:

```
public static int Min(this int[] array) {
    Requires(array != null);
    Requires(array.Any());
    Ensures(Exists(array, element => Result<int>() == element));
    Ensures(ForAll(array, element => Result<int>() <= element));

    var min = array[0];
    for (var i = 1; i < array.Length; i++) {
        var item = array[i];
        if (item < min) {
            min = item;
        }
    }
    return min;
}
```

Listing 8.12: The final version of `Min()`.

Notice how our code contracts turned out to exactly match the listed shortcomings in the beginning of the section.

8.2.2 Verdict

If you look closely at the generated input in the example above, you will see that its quality varies a lot; only a single test used `int.MinValue`. While writing this small example, I have received completely different results depending on the order in which changes were made and contracts were added. It seems difficult to control the outcome, and IntelliTest quickly becomes challenged, when the number of contracts get too high – figure 8.6 only has four tests, two violating preconditions, and two inputting ones and zeros in slightly different order. The final postcondition did luckily improve one of those tests, introducing 2^{20} and `int.MinValue` + 2.

The controlled example was challenging, but nothing compared to a project like C6. I tried testing `Add()` for `ArrayList<T>`, when it only implemented `IExtensible<T>`, but I got no useful results. Though the method is simple, it is still too big a mouthful for IntelliTest.

IntelliTest is the commercial version of the research tool Pex [60] and is now an integrated part of Visual Studio 2015. Though IntelliTest has made it into the real world, it seems that it has not gained too much traction yet; there are less than three dozen questions on StackOverflow tagged with IntelliTest [130], and the available information about the tool is rather sparse, mainly consisting of the same introductory tutorials over and over again. When Microsoft's own MVPs talk about the product, they even say they only use it for creating test suites for legacy code [121].

IntelliTest seems to be able to do some good, but does require a lot of attention if it is expected to do some real good. My current impression is that IntelliTest at best can help with missing preconditions by find exception-causing input. IntelliTest was not used as part of C6.

8.3 PostSharp

PostSharp [128] is a tool that will post-process the output from the C# compiler and add the behaviors of different user-defined aspects. This allows developers to eliminate a lot of boilerplate code and leave the tedious task of writing and maintaining it to the tool.

What makes PostSharp interesting in relation to C6, is its abilities to generate pre-conditions using attributes [116]. PostSharp provides several standard preconditions, for instance for requiring that a reference type is non-null, that a string is non-empty, or that an integral type is within certain bounds:

```
[Required]
public IEqualityComparer<T> EqualityComparer { get; }
[NonEmpty]
public string Name { get; set; }
[Range(0, int.MaxValue)]
public int Size { get; private set; }
```

Listing 8.13: Property contracts written using PostSharp's standard preconditions.

This can shorten the contract code quite considerably. Compare the Microsoft's Code Contracts in listing 8.14 with the PostSharp equivalent in listing 8.15:

```
// Normal precondition with Code Contracts
public int Count {
    get {
        Ensures(Result<int>() >= 0);
        return _count;
    }
    private set {
        Requires(value >= 0);
        Ensures(_count >= 0);
        _count = value;
    }
}
private int _count;
```

Listing 8.14: Code Contracts precondition.

```
// Attribute precondition with PostSharp
[Positive]
public int Count { get; private set; }
```

Listing 8.15: PostSharp precondition.

This is quite a reduction in code, and the PostSharp version even ensures that we do not write to a private backing field, thereby circumventing the contracts. Even if we need a private backing field, PostSharp allows us to put contracts on it, which is currently not possible with Code Contracts [37]:

```
[Positive]
private int _count;
```

Listing 8.16: Precondition on private field using PostSharp.

Beside the out-of-the-box contracts, PostSharp also allows us to create our own contract attributes that can do more complex things [117], and even localize the error messages [118]. A big advantage – one that Code Contracts also has – is that preconditions are inherited, i.e. preconditions will automatically be enforced in all implementations of a method [116].

8.3.1 Verdict

PostSharp seems like a very interesting tool, that could make it noticeably more efficient to work with code contracts. There are, however, some issues that make PostSharp less attractive to C6.

For starters, PostSharp only do preconditions and instead of using Microsoft's Code Contracts, it throws normal exceptions using *if-then-throw* statements. PostSharp is therefore not simply a compact way of writing contracts for Code Contracts, but an actual alternative. This also means that we do not get the benefits of the analysis tools that work with Code Contracts, when we use PostSharp – considering the results with the static checker, that might not be such a big loss after all.

Probably the biggest issue is that PostSharp would add a new dependency on C6. One of the goals of C6 is to keep the core library independent of third-party libraries – this does not apply to the other projects like the testing project `C6.Tests`, which for instance uses NUnit. The benefit of using Code Contracts is that it is included in the .NET Frameworks and therefore does not add any new dependencies, even if the user does not compile with Code Contracts. Moreover, PostSharp is a not-so-cheap commercial product [119], and even though they provide free tools for open-source projects [129], it would still require people, that download the C6 source, to install the tool and acquire C6's license.

For these reasons, PostSharp was only evaluated for this project, but it did not make it into C6 this time around. I do, however, consider PostSharp a tool that could potentially replace Code Contracts' precondition in the future, depending on how the Code Contracts project evolves as further discussed in section 9.2.

8.4 PVS-Studio

PVS-Studio [132] is a static analysis tool used to detect bugs in C, C++ and C# code. The tool was listed in my project description, because I mistakenly thought that it used Code Contracts in its analysis, due to a blog post titled "Analysis of Microsoft Code Contracts" [131]. The article was, however, about how PVS-Studio was used to find bugs in the Code Contracts open-source project. I did nonetheless use the tool in the project, because OOO "Program Verification Systems" was kind enough to provide a free license.

The tool gave a handful of warnings in C6. One issue was an old abstract implementation of `Dispose()` from the `IDisposable` interface found in `IListContract<T>`. `C5.IList<T>` implements the interface because of views, which C6 currently does not implement. The abstract method could therefore rightly be removed for now.

The rest of the warnings were all due to the use of the Boolean `&` operator [93]. The `&` operator is used in `ArrayList<T>` in relation to `CheckVersion()`, which returns `true` if a local version number is equal to the list's current version number, but otherwise throws an exception. By returning a Boolean value, we can inline the call, as seen in lines 6 and 12 of listing 8.17. Contrary to the `&&` operator [96], `&` evaluates both sides instead of short-circuiting the evaluation if the first expression is `false`. Since `CheckVersion()` throws an exception instead of returning `false`, the right side of the expression is never evaluated unless needed. `&` is used here because it is simpler (and therefore potentially faster) than `&&`, which requires a conditional jump. PVS-Studio warned because using `&` normally is an error, but here the use was intentional and the warnings were ignored.

```
// ArrayList<T>.GetEnumerator()
public IEnumerator<T> GetEnumerator() {
    var version = _version;
    // Check version at each call to MoveNext() to ensure an exception is thrown
    // (even when the enumerator was really finished)
    for (var i = 0; CheckVersion(version) & i < Count; i++) {
        yield return _items[i];
    }
}

// ArrayList<T>.Range.AllowsNull
public bool AllowsNull => CheckVersion() & _base.AllowsNull;
```

Listing 8.17: `C6.ArrayList<T>` *members using* `CheckVersion()` *with the* `&` *operator.*

8.5 JetBrains' ReSharper

ReSharper [28] is one of the best known tools used for developing C#, and for that reason I will assume that the reader is familiar with the tool, and only discuss it in a Code Contracts context.

ReSharper is not a tool that benefits from Code Contracts; quite the opposite in fact. ReSharper will complain about reference types possibly being `null`, even though a precondition prohibits it, and ReSharper will dim contracts (figure 8.8), because it thinks the methods are skipped, even though they are enabled.

8.5.1 Code Templates – Live Templates

ReSharper is however also able to help us tremendously when working with Code Contracts. Most noticeably when it comes to writing them. Live Templates are ReSharper's improved version of Visual Studio's code snippets [29]. We can use Live Templates to reimplement Code Contracts' code snippets [56, section 6.3] using our own style, which includes things like comments and full qualifiers (the latter can be shortened automatically by ReSharper using the qualified references). The Code Contracts Live Templates are saved in the team-wide solution settings file, which is a part of the C6 code base.

The live templates were used extensively through out the project to write both code contracts and unit tests. Most of the unit tests on the test reference list (section 5.3.1) were coded as a Live Templates. One of them can be seen in figure 8.9.

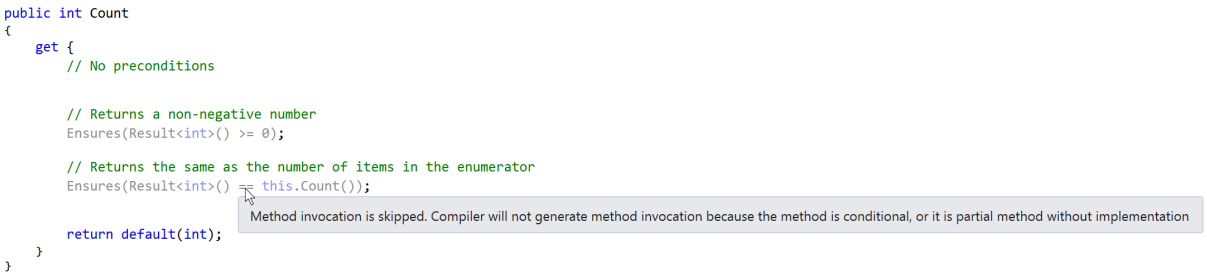


Figure 8.8: ReSharper dims Code Contracts because it thinks the methods are skipped. This can be avoided by wrapping the contracts in `// ReSharper disable InvocationIsSkipped` *and* `// ReSharper enable InvocationIsSkipped` *as done in C6 for contract classes and object invariant methods.*

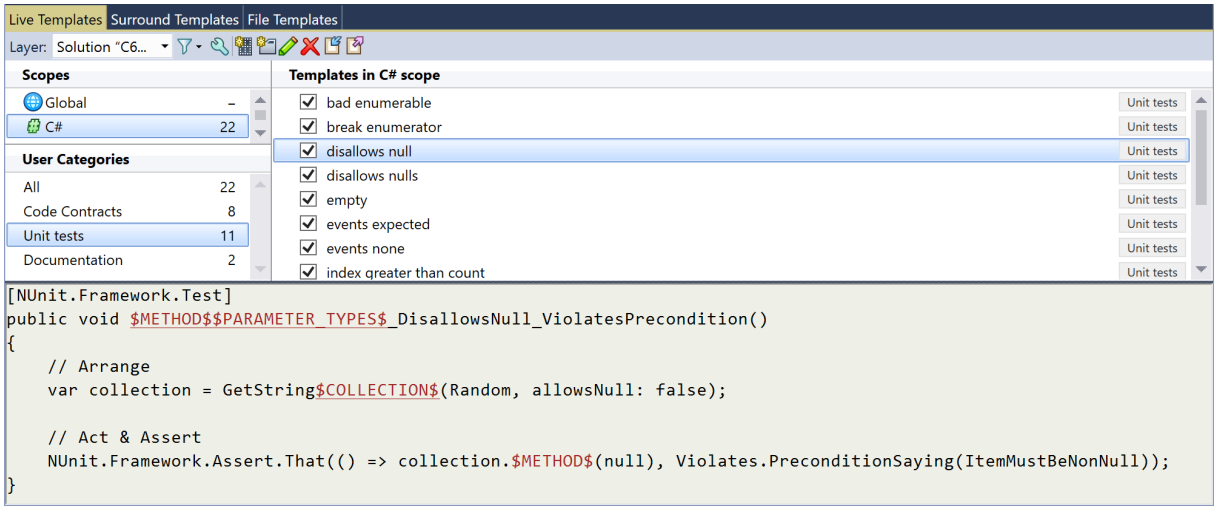


Figure 8.9: Live Template for writing a precondition-violating unit test on a method that takes an item.

Chapter 9

Discussion

9.1 Code Contracts in C6

Adding code contracts to C5 was a big task; putting contracts on the interface hierarchy alone took around a month and a half. The workload would most likely have been lessened, if the contracts were written along with the project. When you design an API, you are already considering the expected behavior, and adding contracts is only a matter of formalising these considerations. However, writing the contracts now forced me to closely consider how every member on the interface worked, allowing me to fix any illogical behavior and update the documentation.

With only a single data structure implemented in C6, many of the benefits gained from Code Contracts seem to still await us, yet its effects have already shown. I have spent surprisingly little time debugging during this project (though I have had the code from `C5.ArrayList<T>` and `SCG.List<T>` as references, I have always first implemented the method myself). By writing the tests first, the small things that I might have forgotten while implementing a method – for instance ensuring that the version number was properly changed – were found instantly. Bugs that were not caught by my unit tests, were quickly caught by the code contracts. The longest time I spent searching for a bug was one hour when trying to figure out why the `Shuffle()` extension method failed (section 7.2.2), and in this case, it was ironically enough my code contracts that contained the bug.

Though I have not properly used the library yet, it is my experience that libraries written with extensive use of Code Contracts end up with a lot fewer errors, than those without. My C5.Intervals project [52], which is currently heavily used by the Australian company Simply Effective Solutions [126], has yet to reveal any hidden bugs.

9.2 The Future of Code Contracts

The Code Contracts framework never really made it out of the research department. Though it has been included in the `System.Diagnostics.Contracts` namespace [73], it was never directly available as part Visual Studio, and originally required Visual Studio 2010 Premium or Ultimate just to run the static checker [2]. Microsoft Research open-sourced Code Contracts in the beginning of 2015 after having worked on it for more than six years [5]. Though it might seem like a positive thing that Microsoft opened it up to the community, it does seem that they abandoned the project from there on.

Concerns about Code Contracts' future have already previously been raised [20], but it now seems that Code Contracts is slowly coming to a halt. During this thesis project, the Code Contracts' GitHub repository only received 18 commits [88] as shown by the highlighted section in figure 9.1. As the concerned citizen I am, I created an issue on

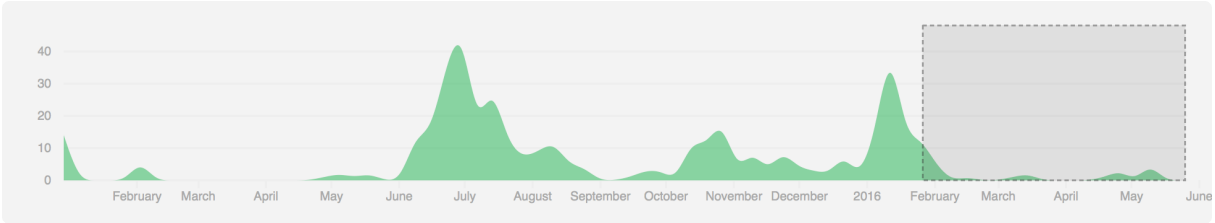


Figure 9.1: The graph shows the contributions (commits) made to the master branch of the Code Contracts GitHub repository from January 2015 to May 2016. The highlighted section covers the activity during this master thesis.

the GitHub asking what the future of Code Contracts would look like [44]. Though the question got quite the attention from the community (receiving eight "thumbs up" in a couple of days), it took a bit longer before the maintainers of the library responded. The general response was that most of the developers involved in the project all worked on it more or less as a hobby project. The contributor Yaakov [145] said that "[Code Contracts] appears to have been defunded and largely abandoned within Microsoft, with Francesco [Logozzo, original Code Contracts developer at Microsoft] uploading the project to GitHub at the same time he left Microsoft" [44]. The main contributor Sergey Teplyakov [133] said that he did use the framework professionally within Microsoft and that he would try to his best to keep working on the project. So even though it does seem that Code Contracts is still breathing, it is currently impossible to say for how much longer.

9.2.1 Replacing Code Contracts

If Code Contracts dies out, it will be necessary to find a way to replace it in C6. Code Contracts is roughly speaking used for three reasons: preconditions, postconditions and object invariants. We must therefore look at the alternatives to replace all three of them.

Preconditions seem to be the easiest to replace with other technologies. The most obvious choice would be to use *if-then-throw* statements as in C5. This would add no extra dependencies to the library and at the same time be a solution that developers can be expected to be familiar with. The downside to this is that developers using C6's interface will not inherit the preconditions, but must implement them themselves. Alternatively, tools like PostSharp [128] or Fody [15] could be used to consistently add contracts to implementing and inheriting classes, but it would be at the cost of using commercial/third-party tools.

The biggest problem with not using Code Contracts does seem to be the missing postconditions and object invariants. Though users of the library may not know that they even exist, they are a safety net for the library developer and greatly help ensure that code is correct. And this might actually be the argument for why we do not *need* to find a replacement for Code Contracts' postconditions and object invariants. The postconditions and invariants could simply be left alone, because Code Contracts is a part of the standard libraries in .NET, so users of the library could easily ignore them. We would still have the unit tests to help ensure that everything works as intended. So even though replacing Code Contracts' is difficult, it might not be necessary.

9.2.2 Improvements to Code Contracts

I think the best way to improve Code Contracts to make it more valuable for projects like C6, would be to actively develop the tool again. With Code Contracts slowly coming to a halt, it is becoming less valuable. Code Contracts' biggest problem is not missing features, but the lack of bug fixing, which could only be fixed if the framework were properly maintained. Beside maintainence, there are a few things that could make the framework even better.

9.2.2.1 Contracts on Delegates and Events

A big improvement to Code Contracts would be to allow users to write contracts on delegates and events. The Code Contracts User Manual [56, section 11.2] mentions that it is not possible to add contracts to delegates, but says that it will be added in the future. To the best of my knowledge [43], this still has not made it into the framework – even after the project was open sourced – but neither was it ever high on the priority list [122].

It is also not possible to have contracts for event handling. We cannot say that a method should raise an event, nor can we say what its arguments should be. We currently have to test that. Ideally, methods could have contracts that state that events should be raised:

```
1 Contract.Ensures(!Contract.Result<bool>() || Contract.RaisesEvent(nameof(CollectionChanged)));
```

Listing 9.1: An example of a possible notation for an event contract.

It may, however, not even be possible to have pure contracts that could verify that events were raised. In order to know whether an event was raised, we must add an event handler to that collection's event. This is strictly not a side-effect-free operation, as it would alter the object – in C5 and C6, `ICollectionValue<T>`'s `ActiveEvents` would possibly change. Code Contracts would have to intercept the event raising without having to register any handlers, but that seems a bit hacky and complicated.

9.3 The Future of C5 and C6

A question that has often come up in my discussions with Rasmus Lystrøm is whether C5 has a future, or whether it has simply outlived itself. Even though Lystrøm is the maintainer of the C5 Collection Library at GitHub [32], he says, he has yet to use it in a real-life setting. Many of the things needed in our everyday programming is already covered by the data structures in `System.Collections.Generic`, so do we really need more libraries?

I think C5 still has great things to offer. C5 does have features that are not matched by the standard libraries, making it great in certain situations. Just consider its extensive interface hierarchy, collection events, views and data structures like the priority queue `IntervalHeap<T>`, which is being highly recommended [26]. I do, however, still think that C5 has become too outdated, but that was exactly why I wanted to update it.

With C6 – when all C5's features have been fully ported – the library is once again up-to-date. While preserving what C5 did best, C6 adds extensive contracts to the library,

makes (directed) collection values even more useful and efficient, and gives the whole library a much-needed facelift. With C6, the library even gets the possibility of becoming a data structure development framework.

9.3.1 C6 as a Data Structure Development Framework

Imagine you had invented a new data structure – for instance the sorted collection Sorted Split List [6] – and you wanted to implement it in C#. You would have to make up a sensible interface, and to be sure that everything worked correctly, you would likely want some unit tests and maybe even code contracts. You would probably also want to have some similar data structures that you could compare Sorted Split List's performance to. This would likely take some time – little of which would actually be spent writing the data structure, but instead be spent on the time-consuming boilerplate code around it. Now imagine you were given everything, but the actual implementation, from the beginning; imagine you had a *data structure development framework*. This would be possible with C6.

You would not have to think about a good interface for the data structure; just implement `IIndexedSorted<T>` and its members. You would not have to write all the contracts on the interface; just enable Code Contracts in your project and add the invariants of the data structure to your own class. You would not have to write extensive unit tests; just create a test class that inherits from `IIndexedSortedTests` and implement the abstract methods. And finally, you would not have to write the performance tests or similar data structures; just run those already in C6 for `IIndexedSorted<T>` and your implementation would automatically be tested along with those already in C6.

This is almost the reality of C6 now. To make it actually work, the unit tests would have to be completed for all types of data structures and the performance tests would have to be written. But in its current state, C6 already provides most of this functionality, since all interfaces have the required contracts and unit tests. One could even reuse the base classes in C6, if they were reintroduced (this is likely to happen as the number of data structures increases).

This is all a very theoretical and academic idea, because few actually have to implement their own data structures. It is, however, not an unrealistic scenario. I have personally had a need for this, when I had to implement Boudoux's Sorted Split List [6] so I could use it as an underlying data structure for other data structures. I had to make my own interface, my own unit/performance tests and I had to find similar data structures to compare the performance to [51]. With C6, I could have simply focused on what I actually needed: the Sorted Split List.

9.4 Future Work

A very natural next step would be to port all the remaining data structures from C5 to C6. I expect data structures, like `LinkedList<T>` and `CircularQueue<T>` that behave similarly to `ArrayList<T>` will be easier to port, as they require little additional testing. It is likely to be more time consuming to port the sets and sorted collections, as the necessary tests have not fully been written yet.

Chapter 10

Conclusion

The C5 Collection Library was successfully upgraded to become a contemporary collection library in the shape of the new C6 Collection Library. The new library embraces the recent language and platform additions by getting rid of old constructs made obsolete by newer technologies like LINQ and the language features of C# 6.0.

C6 improves on C5's features like the new lazy collection values that take advantage of the lazy enumerators in C# to help increase the performance in an easy-to-understand fashion. The way to a finished library is still long, but with this update C6 has proven its worth and relevance in the future, maybe even as data structure development framework.

The implemented data structure `ArrayList<T>` showed great benefits from the use of Code Contracts, making it possible to implement the class in an even more concise fashion than seen in C5. The library has shown that Code Contracts can be a great addition to larger projects like C6. In hindsight, other data structures could possibly have showed Code Contracts from an even better side, as the invariants in `ArrayList<T>` are rather trivial compared to those of a self-balancing binary tree or a hashed linked list. But taking the rather short time frame of this project into account, I believe that `ArrayList<T>` worked well. I am for one looking forward to expanding the library with Code Contracts. Code Contracts posed many challenges for the library; some of which were fixed, and some of which still require a satisfactory solution.

The project was unable to show any significant gains from using some of the many tools that take advantage of the Code Contracts framework. Though the tools can do interesting things on paper, the tend to fall short when used in extensive projects like the C6 Collection Library.

Appendix A

Example of Postcondition Inheritance for Members with Multiple Roots

```
1  using System.Diagnostics.Contracts;
2
3  interface IFoo {
4      int Number { get; }
5  }
6
7  [ContractClass(typeof(IBarContract))]
8  interface IBar : IFoo {
9      new int Number { get; } // <-- overriding makes the postcondition being checked sometimes!
10 }
11
12 [ContractClassFor(typeof(IBar))]
13 abstract class IBarContract : IBar {
14     public int Number {
15         get {
16             Contract.Ensures(Contract.Result<int>() >= 0);
17             return default(int);
18         }
19     }
20 }
21
22 internal class Foo1 : IFoo {
23     public int Number => -1;
24 }
25
26 internal class Bar1 : Foo1, IBar {} // <-- Will *not* have its postcondition checked
27
28 internal class Bar2 : IBar {
29     public int Number => -1; // <-- Will have its postcondition checked, because Bar2
           overrides Number
30 }
31
32 internal class Foo3 : IFoo {
33     public virtual int Number => -1;
34 }
35
36 internal class Bar3 : Foo3, IBar {} // <-- Will *not* have its postcondition checked
37
38 internal class Bar4 : Foo3, IBar {
39     public override int Number => -1; // <-- Will have its postcondition checked, because
           Bar4 overrides Number
40 }
```

Listing A.1: The example demonstrates the subtle differences that affect postcondition inheritance in Code Contracts for members with multiple roots [41].

Appendix B

Implications using an Extension Method

As explained in section 2.8.1, neither C# nor Code Contracts support implications. It is, however, possible to write an extension method to provide this functionality:

```
1 [Pure]
2 public static bool Implies(this bool x, bool y) => !x || y;
```

Listing B.1: The `Implies()` *extension method, which could be used in contracts.*

This allows us to write the implication from listing 2.20 as:

```
1 // Contains(item) -> Result<int>() > 0
2 Ensures(Contains(item).Implies(Result<int>() > 0));
```

Listing B.2: The implication from listing 2.20 written using the extension method in listing B.1.

Though this may initially seem like a good idea, my experience is that it really does not improve readability. It tends to read badly, especially if the antecedent of the conditional is negated, in which case we must wrap the expression in parentheses:

```
1 // Using normal Boolean logic
2 Requires(AllowsNull || item != null);
3
4 // Using Implies()
5 Requires((!AllowsNull).Implies(item != null));
```

Listing B.3: Using the `Implies()` *extension method from listing B.1 tends not to improve readability.*

For these reasons, C6 does not use any helper methods for implications.

Appendix C

Convention Tests

Convention tests are a way to ensure that your code adheres to your conventions by using unit tests to programmatically test your code. They were first presented by Krzysztof Koźmic at NDC 2012 [35] and have since been realized in a C# library [134]. The prepackaged convention tests [135] that come with the library give a good intuition of use cases and capabilities of convention tests:

- All classes have default constructor.
- All methods are virtual.
- Class type has specific namespace.
- Files are embedded resources.
- Project does not reference dlls from `Bin` or `Obj` directory.

Convention tests can also help ensure that each file contains a library-specific header or that implementations are tested with the proper test classes. The convention tests require a dependency on the TestStack.ConventionTests library [134] and can naturally be stored either in the test project or a separate convention test project. The library provides a framework for making the tests easier to write, but it mainly relies on reflection to look at your code.

C.1 All Classes Must Be Serializable

C5 has previously had problems ensuring that all classes were properly annotated with the `Serializable` attribute [127]. It has been left to the developer to remember to add the attribute to any classes that needed it. We would much rather let the tools help ensure this is not forgotten. This is easily done with a convention test. The test is written and run as a simple NUnit test:

```
1 [Test]
2 public void Convention_AllClassesMustBeSerializable() {
3     Convention.Is(new AllClassesAreSerializable(), Types.InAssembly(C6));
4 }
```

Listing C.1: The convention test checks that all classes are serializable.

Listing C.2 shows the full implementation of the `AllClassesAreSerializable` class that handles the types found in the C6 assembly:

```
public class AllClassesAreSerializable : IConvention<Types> {
    public void Execute(Types data, IConventionResultContext result) {
        var nonSerializableTypes = data.Where(type => !type.IsSerializable &&
                                                      !type.IsInterface &&
                                                      !type.IsStatic() &&
                                                      !type.IsCompilerGenerated() &&
                                                      !type.Name.Equals("SerializableAttribute"));

        result.Is("Classes must be serializable", nonSerializableTypes);
    }

    public string ConventionReason => "All classes should be serializable:
        https://github.com/sestoft/C5/issues/8";
}
```

Listing C.2: The `AllClassesAreSerializable` *class finds classes that should be serializable.*

If the test fails it lists the classes, where the attribute is missing:

```
ConventionFailedException: 'Classes must be serializable' for 'Types in C6'
-----------------------------------------------------------------------------------------------
    ArrayList<T>
```

Listing C.3: The output from the test in listing C.1 when `ArrayList<T>` *is not serializable.*

The test can be run along with the regular unit tests and any missing attributes can be caught early, ensuring that they never make it into production.

C.2 Testing Unit Test Conventions

A place where convention tests could be beneficial is in unit testing: all methods on an interface should be tested, and depending on the type of method, certain tests should always be created for a method, as suggested by the test reference list in appendix D. It can easily become difficult to ensure that we tested all the necessary things for each method on each interface, especially as the number of tests increase.

Imagine a convention test that would look at each interface in C6 that inherits from `ICollectionValue<T>` and find its matching test class. For each interface method, the convention test would check whether the method was tested in certain ways based on the method's parameter types and attributes, in much the same way as described in section 5.3.1. The convention test could identify the unit tests based on a naming convention and report any tests that were missing. This would be greatly beneficial for the developer writing tests as missing standard tests would quickly be pointed out, allowing the developer to focus on the test logic. This would also be helpful when adding new interface methods.

Regardless of how great these convention tests would be, they have not become a part of C6 (yet). Writing these tests seems rather time consuming, as many aspects have to be considered. Would a method like `Add()` for instance be tested on each interface (`IExtensible<T>` and `ICollection<T>`) where it appears? There currently is a problem with signed assemblies, which might be caused by ConventionTests, NUnit, the test runners or something completely different. Furthermore, it seems that many convention tests can be covered using Visual Studio Analyzers [136], which would result in compiler support instead of test support.

Appendix D

Excerpt of Test Reference List

- If a method's precondition is violated it must throw an exception. Certain preconditions require certain tests:
  - A method that accepts a generic item must be given `null` (violates precondition depending on the value of `ICollectionValue<T>.AllowsNull`).
  - A method that takes an enumerable of generic items must be given a `null` enumerable in one test (violates precondition) and an enumerable containing `null` values in another (violates precondition depending on the value of `ICollectionValue<T>.AllowsNull`).
  - The method must be given extreme data depending on the parameter type, e.g. 0, 1, the last item, negative value, the collection count, something greater than the collection count, the minimum value, and the maximum value.
- Events:
  - A non-pure *operation* must raise events.
  - A pure *operation* must not raise events.
- A pure method must be called on a random collection without changing it.
- The method must be called on an empty collection (result depends on method).
- A method that takes a generic item must try with a new item (one that is not already in the collection) and an existing item (one that has an equal in the collection, but preferably not the same object).
- A method that takes an enumerable of generic items must be given a *bad* enumerable that throws an exception. Throwing the exception must not change the collection.
- A method with a Boolean result must be tested where it returns `true` and `false` (in different tests).
- Enumerator tests:
  - A non-pure *operation* must break the enumerator.
  - A pure *operation* must not break the enumerator.

Notice that we need to consider whether the operation – not the method – is pure, when testing whether the call breaks the enumerator, raises events, and so on. This is because non-pure methods might not alter the collection as discussed in section 3.5.4.1.

Bibliography

[1] AlexArchive. C# 6 Equivalents in C# 5, 2015. URL https://github.com/AlexArchive/c6-equivalents-in-c5.

[2] Melitta Andersen. I just installed Visual Studio 2010, now how do I get Code Contracts?, 2010. URL https://goo.gl/64Vdl2.

[3] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2004.

[4] Kent Beck. *Test-Driven Development by Example*. Addison-Wesley Professional, 2002.

[5] Somasegar's blog. DevLabs: Code Contracts for .NET, 2016. URL https://blogs.msdn.microsoft.com/somasegar/2009/02/23/devlabs-code-contracts-for-net/.

[6] Aurélien Boudoux. SortedSplitList - An Indexing Algorithm in C#, 2013. URL http://www.codeproject.com/Articles/610399/SortedSplitList-An-Indexing-Algorithm-in-Csharp.

[7] Ada Information Clearinghouse. Ada, 2012. URL http://www.adaic.org/.

[8] Software Freedom Conservancy. Git, 2016. URL https://git-scm.com/.

[9] Jim Counts. Packaging Contract Assemblies Like A Pro, 2014. URL https://ihadthisideaonce.com/2014/10/13/packaging-contract-assemblies-like-a-pro/.

[10] James Craig. Process is terminated due to StackOverflowException. #169, 2016. URL https://github.com/Microsoft/CodeContracts/issues/169.

[11] Krzysztof Cwalina and Brad Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Pearson Education, 2nd edition, 2008.

[12] Aaron Dandy. Using Code Contracts from New Project to NuGet, 2014. URL http://mediocresoft.com/things/code-contracts-from-new-project-to-nuget.

[13] Vincent Driessen. A Successful Git Branching Model, 2010. URL http://nvie.com/posts/a-successful-git-branching-model.

[14] EWSoftware. Sandcastle Help File Builder (SHFB), 2016. URL https://github.com/EWSoftware/SHFB.

[15] Fody. Fody, 2016. URL https://github.com/Fody/Fody.

[16] Fody. FodyDependsOnTargets, 2016. URL https://github.com/Fody/Fody/wiki/FodyDependsOnTargets.

[17] .NET Foundation. .NET Core, 2016. URL https://dotnet.github.io.

[18] .NET Foundation. NuGet, 2016. URL https://www.nuget.org.

[19] FsCheck. FsCheck, 2016. URL https://github.com/fscheck/FsCheck.

[20] Juan Gelós. Is Code Contracts dead?, 2013. URL https://social.msdn.microsoft.com/Forums/en-US/038d6e15-3bbf-4d2e-8e32-865b00978d5c/is-code-contracts-dead?forum=codecontracts.

[21] GitHub, Inc., 2016. URL https://github.com/.

[22] Hackage. The QuickCheck package, 2016. URL https://hackage.haskell.org/package/QuickCheck.

[23] Rich Hickey. Clojure, 2016. URL http://clojure.org/.

[24] RiSE (Research in Software Engineering) (Microsoft). Code Contracts for .NET, 2016. URL https://visualstudiogallery.msdn.microsoft.com/1ec7db13-3363-46c9-851f-1ce455f66970.

[25] ECMA International. Standard ECMA-335 Common Language Infrastructure (CLI), 2012. URL http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[26] jaras. Priority Queue in .NET, 2008. URL http://stackoverflow.com/a/1114323/234910.

[27] Matthias Jauernig. Code Contracts #5: Method purity, 2009. URL http://www.minddriven.de/index.php/technology/dot-net/code-contracts/code-contracts-method-purity.

[28] JetBrains. ReSharper – Visual Studio Extension for .NET Developers, 2016. URL https://www.jetbrains.com/resharper/.

[29] JetBrains. ReSharper – Code Templates, 2016. URL https://www.jetbrains.com/resharper/features/code_templates.html.

[30] julealgon. How do I include contract assemblies in the nupkg automatically?, 2015. URL http://stackoverflow.com/q/28192428/234910.

[31] Niels Kokholm and Peter Sestoft. The C5 Generic Collection Library for C# and CLI. Technical Report TR-2006-76, IT University of Copenhagen, January 2006.

[32] Niels Kokholm, Peter Sestoft, and Rasmus Lystrøm. C5 Generic Collection Library for C#/.NET, 2016. URL https://github.com/sestoft/C5.

[33] Niels Kokholm, Peter Sestoft, and Rasmus Lystrøm. C5 - GitHub, 2016. URL https://github.com/sestoft/C5/blob/c5bc045abc939a527b210b52de8a5df852642b0c/C5/Collections.cs#L1152.

[34] Niels Kokholm, Peter Sestoft, Rasmus Lystrøm, and Mikkel Riise Lund. C6 Generic Collection Library for C#/.NET, 2016. URL https://github.com/C6/C6.

[35] Krzysztof Koźmic. Beyond the Compiler - Going up to 11 with Conventions in a Statically Typed Language, 2012. URL https://vimeo.com/43676874.

[36] Eric Lippert. Vexing exceptions, 2008. URL https://blogs.msdn.microsoft.com/ericlippert/2008/09/10/vexing-exceptions/.

[37] Mikkel Riise Lund. Question "Can you put Code Contracts on Private Fields?", 2015. URL http://stackoverflow.com/q/32438195/234910.

[38] Mikkel Riise Lund. Bug in Tests Means that Only Added Events are Tested #44, 2016. URL https://github.com/sestoft/C5/issues/44.

[39] Mikkel Riise Lund. Bug in ArrayBase<T> Makes ArrayList<T>.AddAll() Enter an Infinite Loop for Large Enumerables #45, 2016. URL https://github.com/sestoft/C5/issues/45.

[40] Mikkel Riise Lund. Commit a180a51, 2016. URL https://github.com/C6/C6/commit/a180a51d72061d80700d9ff1a98f35e6d81af829.

[41] Mikkel Riise Lund. Rewriter Gives Warning CC1076 For Postcondition #331, 2016. URL https://github.com/Microsoft/CodeContracts/issues/331.

[42] Mikkel Riise Lund. Call-Site Requires Checking Not Working in Constructor #392, 2016. URL https://github.com/Microsoft/CodeContracts/issues/392.

[43] Mikkel Riise Lund. Contracts on Delegates #402, 2016. URL https://github.com/Microsoft/CodeContracts/issues/402.

[44] Mikkel Riise Lund. What does the Future of Code Contracts Look Like? #409, 2016. URL https://github.com/Microsoft/CodeContracts/issues/409.

[45] Mikkel Riise Lund. GitHub Issues, 2016. URL https://github.com/search?q=author%3Alundmikkel.

[46] Mikkel Riise Lund. EqualConstraint does Not use Equals Override on the Expected Object #1439, 2016. URL https://github.com/nunit/nunit/issues/1439.

[47] Mikkel Riise Lund. Comparing Equality using IEquatable<T> Should Use Most Specific Method #1484, 2016. URL https://github.com/nunit/nunit/issues/1484.

[48] Mikkel Riise Lund. Check that an Enumerable Contains the Same Items, 2016. URL https://groups.google.com/forum/#!topic/nunit-discuss/JaCmqrZTpWY.

[49] Mikkel Riise Lund. Introduce Is.Zero syntax to test for zero #1246, 2016. URL https://github.com/nunit/nunit/issues/1246.

[50] Mikkel Riise Lund. Repeatable Random Tests with Fixed/Additional Seed #1461, 2016. URL https://github.com/nunit/nunit/issues/1461.

[51] Mikkel Riise Lund. Sorted Lists, 2016. URL https://github.com/lundmikkel/SortedLists.

[52] Mikkel Riise Lund, Christian Lykke-Rasmussen, and Anders Bech Mellson. C5.Intervals, 2016. URL https://github.com/lundmikkel/C5Intervals.

[53] Dino Mandrioli and Bertrand Meyer. *Advances in Object-Oriented Software Engineering*, chapter Design by Contract, pages 1–50. Prentice-Hall, Inc., 1992.

[54] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[55] Microsoft. Framework Design Guidelines, 2009. URL https://msdn.microsoft.com/en-us/library/ms229042.aspx.

[56] Microsoft. *Code Contracts User Manual*. Microsoft Corporation, November 2013.

[57] Microsoft. SCG.ICollection.cs Contracts, 2015. URL https://github.com/Microsoft/CodeContracts/blob/master/Microsoft.Research/Contracts/MsCorlib/System.Collections.Generic.ICollection.cs.

[58] Microsoft. SCG.IList.cs Contracts, 2015. URL https://github.com/Microsoft/CodeContracts/blob/master/Microsoft.Research/Contracts/MsCorlib/System.Collections.Generic.IList.cs.

[59] Microsoft. Generate unit tests for your code with IntelliTest, 2015. URL https://msdn.microsoft.com/en-us/library/dn823749.aspx.

[60] Microsoft. Pex and Moles - Isolation and White box Unit Testing for .NET, 2015. URL http://research.microsoft.com/en-us/projects/pex/.

[61] Microsoft. int (C# Reference), 2016. URL https://msdn.microsoft.com/library/5kzh1b5w.aspx.

[62] Microsoft. Object.ReferenceEquals Method (Object, Object), 2016. URL https://msdn.microsoft.com/library/system.object.referenceequals.aspx.

[63] Microsoft. Enumerable.SequenceEqual<TSource> Method (IEnumerable<TSource>, IEnumerable<TSource>), 2016. URL https://msdn.microsoft.com/library/bb348567.aspx.

[64] Microsoft. Array Class, 2016. URL https://msdn.microsoft.com/library/System.Array.aspx.

[65] Microsoft. ValueType.Equals Method (Object), 2016. URL https://msdn.microsoft.com/library/2dts52z7.aspx.

[66] Microsoft. DotNet CodeContracts v.1.10.10126.2-rc1, 2016. URL https://github.com/Microsoft/CodeContracts/releases/tag/v1.10.10126.2.

[67] Microsoft. Source code for the CodeContracts tools for .NET – Repository, 2016. URL https://github.com/microsoft/codecontracts.

[68] Microsoft. Code Contracts, 2016. URL https://msdn.microsoft.com/library/dd264808.aspx.

[69] Microsoft. Code Contracts – Releases, 2016. URL https://github.com/Microsoft/CodeContracts/releases.

[70] Microsoft. Visual Studio Downloads, 2016. URL https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx.

[71] Microsoft. IEnumerator.MoveNext Method (), 2016. URL https://msdn.microsoft.com/library/system.collections.ienumerator.movenext.aspx#Anchor_1.

[72] Microsoft. Reference Source – List<T>– .NET Framework 4.6.1, 2016. URL http://referencesource.microsoft.com/#mscorlib/system/collections/generic/list.cs.

[73] Microsoft. System.Diagnostics.Contracts Namespace, 2016. URL https://msdn.microsoft.com/library/system.diagnostics.contracts.aspx.

[74] Microsoft. Array.Sort<T> Method (T[]), 2016. URL https://msdn.microsoft.com/library/kwx6zbd4.aspx#Anchor_2.

[75] Microsoft. Deferred Execution and Lazy Evaluation in LINQ to XML (C#), 2016. URL https://msdn.microsoft.com/library/mt693152.aspx.

[76] Microsoft. Enumerable.All<TSource> Method (IEnumerable<TSource>, Func<TSource, Boolean>), 2016. URL https://msdn.microsoft.com/library/bb548541.aspx.

[77] Microsoft. Enumerable.Any<TSource> Method (IEnumerable<TSource>, Func<TSource, Boolean>), 2016. URL https://msdn.microsoft.com/library/bb534972.aspx.

[78] Microsoft. Enumerable.FirstOrDefault<TSource> Method (IEnumerable<TSource>), 2016. URL https://msdn.microsoft.com/library/bb340482.aspx.

[79] Microsoft. Enumerable.Select Method, 2016. URL https://msdn.microsoft.com/library/system.linq.enumerable.select.aspx.

[80] Microsoft. Enumerable.Where<TSource> Method (IEnumerable<TSource>, Func<TSource, Boolean>), 2016. URL https://msdn.microsoft.com/library/bb534803.aspx.

[81] Microsoft. List<T>.ConvertAll<TOutput> Method (Converter<T, TOutput>), 2016. URL https://msdn.microsoft.com/library/73fe8cwf.aspx.

[82] Microsoft. List<T>.Exists Method (Predicate<T>), 2016. URL https://msdn.microsoft.com/library/bfed8bca.aspx.

[83] Microsoft. List<T>.FindAll Method (Predicate<T>), 2016. URL https://msdn.microsoft.com/library/fh1w7y8z.aspx.

[84] Microsoft. List<T>.ForEach Method (Action<T>), 2016. URL https://msdn.microsoft.com/library/bwabdf9z.aspx.

[85] Microsoft. List<T>.TrueForAll Method (Predicate<T>), 2016. URL https://msdn.microsoft.com/library/kdxe4x4w.aspx.

[86] Microsoft. System.Collections.Specialized Namespace, 2016. URL https://msdn.microsoft.com/library/system.collections.specialized.aspx.

[87] Mike. What is the => assignment in C# in a property signature, 2015. URL http://stackoverflow.com/a/31764549/234910.

[88] Microsoft. Code Contracts Contributions to master, 2016. URL https://github.com/Microsoft/CodeContracts/graphs/contributors?from=2016-02-01&to=2016-05-31.

[89] Cory Nelson. Controlling Code Contract references in a .nuspec, 2012. URL http://stackoverflow.com/questions/13687995/controlling-code-contract-references-in-a-nuspec.

[90] Microsoft | Developer Network. Enhancing Debugging with the Debugger Display Attributes, 2016. URL https://msdn.microsoft.com/library/ms228992.aspx.

[91] Microsoft | Developer Network. .NET Framework Class Library, 2016. URL https://msdn.microsoft.com/library/mt472912.aspx.

[92] Microsoft | Developer Network. Language-Integrated Query (LINQ) (C#), 2016. URL https://msdn.microsoft.com/library/mt693024.aspx.

[93] Microsoft | Developer Network. & Operator (C# Reference), 2016. URL https://msdn.microsoft.com/library/sbf85k1c.aspx.

[94] Microsoft | Developer Network. Array Class, 2016. URL https://msdn.microsoft.com/library/system.array.aspx.

[95] Microsoft | Developer Network. Array.Reverse Method (Array), 2016. URL https://msdn.microsoft.com/library/d3877932.aspx.

[96] Microsoft | Developer Network. && Operator (C# Reference), 2016. URL https://msdn.microsoft.com/library/2a723cdk.aspx.

[97] Microsoft | Developer Network. List<T>.AddRange Method (IEnumerable<T>), 2016. URL https://msdn.microsoft.com/library/z883w3dc.aspx.

[98] Microsoft | Developer Network. List<T> Class – Constructors, 2016. URL https://msdn.microsoft.com/library/6sh2ey19.aspx#Anchor_2.

[99] Microsoft | Developer Network. List<T>.InsertRange Method (Int32, IEnumerable<T>), 2016. URL https://msdn.microsoft.com/library/884ee1fz.aspx.

[100] Microsoft | Developer Network. List<T>.RemoveRange Method (Int32, Int32), 2016. URL https://msdn.microsoft.com/library/y33yd2b5.aspx.

[101] Microsoft | Developer Network. #region (C# Reference), 2016. URL https://msdn.microsoft.com/library/9a1ybwek.aspx.

[102] Rasmus Nielsen. The Future of C5 – Developing the C5 Generic Collection Library for .NET 4.0 and beyond, May 2011.

[103] NuGet. NUnit, 2016. URL https://www.nuget.org/packages/NUnit/.

[104] NUnit. NUnit 3.0, 2015. URL https://github.com/nunit/nunit/releases/tag/3.0.0.

[105] NUnit. NUnit 3.2.1, 2015. URL https://github.com/nunit/nunit/releases/tag/3.2.1.

[106] NUnit. EqualConstraint, 2016. URL https://github.com/nunit/docs/wiki/EqualConstraint#user-content-notes.

[107] NUnit. Assert.Equality.cs, 2016. URL https://github.com/nunit/nunit/blob/master/src/NUnitFramework/framework/Assert.Equality.cs#L126.

[108] NUnit.org. NUnit.org Contributors, 2016. URL https://github.com/orgs/nunit/people?query=lundmikkel.

[109] NUnit.org. NUnit, 2016. URL http://nunit.org/.

[110] NUnit.org. Randomizer Methods, 2016. URL https://github.com/nunit/docs/wiki/Randomizer-Methods.

[111] Oracle. Class ArrayList<E>– subList(int, int), 2016. URL https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html#subList-int-int-.

[112] Oracle. Unchecked Exceptions — The Controversy, 2016. URL https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html.

[113] C6 Organization. Copenhagen Comprehensive Collection Classes with Contracts for C#, 2016. URL https://github.com/C6.

[114] Roy Osherove. *The Art of Unit Testing: With Examples in .NET*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2009. ISBN 1933988274, 9781933988276.

[115] Thomas Owens. Is there a difference between TDD and Test First Development (or Test First Programming)?, 2008. URL http://stackoverflow.com/q/334779/234910.

[116] PostSharp. Contracts, 2016. URL http://doc.postsharp.net/contracts.

[117] PostSharp. Creating Custom Contracts, 2016. URL http://doc.postsharp.net/custom-contracts.

[118] PostSharp. Localizing Contract Errors, 2016. URL http://doc.postsharp.net/contract-localization.

[119] PostSharp. Purchase, 2016. URL https://www.postsharp.net/purchase.

[120] Microsoft Research. Spec#, 2006. URL http://research.microsoft.com/en-us/projects/specsharp/.

[121] Terje Sandstrom and SSW TV. New Unit Testing Features in Visual Studio 2015, 2015. URL https://youtu.be/ANg1Nol6UvU?t=20m07s.

[122] Dave Sexton. Contracts on Delegates, 2010. URL https://social.msdn.microsoft.com/Forums/en-US/1a36832c-4dad-4f23-acfc-32360b8c9449/contracts-on-delegates?forum=codecontracts.

[123] Jon Skeet. Answer to "C#: Where should you place event handler delegates?", 2009. URL http://stackoverflow.com/a/776275/234910.

[124] Jon Skeet. Answer to "How come you cannot catch Code Contract exceptions?", 2010. URL http://stackoverflow.com/a/2640011/234910.

[125] Eiffel Software. Eiffel, 2016. URL https://www.eiffel.com/.

[126] Simply Effective Solutions, 2016. URL http://www.sesolutions.net.au/.

[127] spahnj. BinaryFormatter – Classes not marked as serializable, 2012. URL https://github.com/sestoft/C5/issues/8.

[128] SharpCrafters s.r.o. PostSharp, 2016. URL https://www.postsharp.net/.

[129] SharpCrafters s.r.o. PostSharp – FAQ, 2016. URL https://www.postsharp.net/purchase/faq.

[130] StackOverflow. Questions Tagged with IntelliTest, 2016. URL http://stackoverflow.com/questions/tagged/intellitest.

[131] OOO "Program Verification Systems". PVS-Studio, 2016. URL AnalysisofMicrosoftCodeContracts.

[132] Program Verification Systems. PVS-Studio, 2016. URL http://www.viva64.com/en/pvs-studio/.

[133] Sergey Teplyakov, 2016. URL https://github.com/SergeyTeplyakov.

[134] TestStack. TestStack.ConventionTests, 2016. URL https://github.com/TestStack/TestStack.ConventionTests.

[135] TestStack.ConventionTests. Getting Started, 2016. URL http://teststackconventiontests.readthedocs.org/en/latest#getting-started.

[136] Alew Turner. C# and Visual Basic – Use Roslyn to Write a Live Code Analyzer for Your API, 2016. URL https://msdn.microsoft.com/magazine/dn879356.aspx.

[137] user341451. Answer to "How come you cannot catch Code Contract exceptions?", 2010. URL http://stackoverflow.com/a/2836005/234910.

[138] Dimitri van Heesch. Doxygen, 2016. URL http://doxygen.org.

[139] Wikipedia. Design by contract, 2016. URL https://en.wikipedia.org/wiki/Design_by_contract.

[140] Wikipedia. Design by contract – History, 2016. URL https://en.wikipedia.org/wiki/Design_by_contract#cite_ref-DbC_tm_words2_6-0.

[141] Wikipedia. Gotcha (programming), 2016. URL https://en.wikipedia.org/wiki/Gotcha_(programming).

[142] Wikipedia. Timsort, 2016. URL https://en.wikipedia.org/wiki/Timsort.

[143] WPFNovice_HTS. Why Buffer.BlockCopy runs slower than Array.Copy, 2008. URL https://social.msdn.microsoft.com/Forums/vstudio/en-US/e3a08e63-7188-4f87-bb0a-fed6c8acf553/why-bufferblockcopy-runs-slower-than-arraycopy.

[144] Valentin Wüstholz. Add Support for Bounded Static Analysis #412 , 2016. URL https://github.com/Microsoft/CodeContracts/pull/412.

[145] Yaakov, 2016. URL https://github.com/yaakov-h.