1. Introduction
   1. Chapters
   2. Projects goals
   3. Scope
2. Microsoft’ s Code Contracts

The chapter introduces the Microsoft’s project Code Contracts, including description of the framework and discussion of its the advantages and disadvantages.

* 1. Design by contract

Design by contract is an approach in software design, known also as contract programming, where the assumptions of the developers to a routine (method or property) are made formal by clearly specifying conditions that should be fulfilled.

In the core of the approach is extending the routines with assertions like preconditions, post-conditions and invariants. Preconditions are the conditions that should be fulfilled, so that the method or property to work correctly. Violation of a precondition indicates a bug in the side of the caller of that method. Postconditions represent those conditions that should hold when a routine successfully finished its task. Violation of these conditions though points to bug in the method or the property, not in the caller side. While the pre- and postconditions apply for a concrete routine, the invariants on the other side are related to the whole class, to its internal correctness. The conditions specified like invariants should hold by every routine publicly available to clients. The invariants characterize a class as whole. [1]

Design by Contract approach is introduced by Bertrand Meyer in 1986 for the language Eiffel. Later other languages also incorporated the contracts ideas. Among the languages with native support of Design by Contracts are Ada [4], Eifell [6], Spec# [3]. There are also languages, supporting the approach with third party like Java (in Bean Validation [2]), JavaScript [5] and many others.

* 1. Spec# and the framework Code Contracts

While Spec# been used no longer updated, in return influenced the creation of a framework, incorporating the contract ideas. Microsoft created a framework, so that it is available for all languages of .Net, not only C#. The Contract class is placed in **mscorlib** under the namespace **System.Diagnostics.Contracts** [9]. The Code Contracts became first available in C# 4.0 and supported directly in VS 2010 since 2010, while for VS 2008 and C# 3.5 (and earlier) it is available as a separate library, called **Micrsoft.Contracts.** Since its first release the Code Contracts project is not seriously updated. It becomes community driven since 2015. Since then, there are two community-driven pre-releases of .Net Code Contracts. The first one is from the beginning of 2016 and 2nd one from the middle of 2016, where the latter focuses only on some minor issues. This is the version used in the project.

Unfortunately, Visual studio 2017 has problems with the support of Code Contracts. It has not caught up to the new studio. To be able to build a project in Visual studio (VS) 2017, a file should be copied to a proper place in the VS installation folder [7]. This will only allow a successful build; however, the Code Contracts settings will be unavailable in the Project Properties menu. Therefore, the project should also be opened in VS 2015. An open question in MSDN’s forum about Code Contract support was not answered yet at the time I wrote these lines [8] Because of that I chose to work on Visual Studio 2015.

* 1. Quick Overview – Requires, Ensures

Preconditions

A postcondition specifies a condition that a method should fulfill when it terminates, prior to exiting. If the returned value of a method should be non-negative, the code in the listing 2.2 could be used

// Code

There are some special methods that can only be used within Ensures(). To use the return value, if any, in the postconditions, Contracts.Result<T> is used, where T is the type of the returned value. Contracts.ValueAtReturn<T>(out x) is called when the value of an out parameter is needed, where T is x’s type. Contract.OldValue<T>(e) gives the value of e in the methods pre-state. It means that OldValue<T>(e) is first evaluated when the method is called, before any of the methods statements are executed. [10]

The exceptional postconditions Contract.EnsuresOnThrow<T> (this.F > 0 ) should hold when a method throws an exception.

// Code from C6 Continued

Code Contracts offers a quantifier that has two overloads. The first one takes an enumerator and a generic predicate: Contract.ForAll(enumerator, x => x != null), where enumerator could be every data structure implementing IEnumerable<T>. While the second overload takes an inclusive lower bound and exclusive upper bound, instead of enumerator, and a non-generic predicate:

Contract.Ensures(Contract. ForAll (0, Contract.Result<int[]>().Length, index => Contract.Result<int[]>()[index] > 0)).

This means that the predicate should take an int and return a boolean. ForAll will return true if and only, if the predicate returns true for all items in the collection (first version) or range (second version). If the predicate returns false on one item, then the quantifier returns false and stops its execution. The ForAll can be used both with preconditions and postconditions.

The existential quantification is written by the help of Contracts.Exists(). The overloads and paramters are the same as with ForAll. Exists() returns true if the predicate returns true for at least one item in the collection/range. When true is evaluated, the quantifier stops its execution.

// Code from

Object Invariant are the conditions that should hold after the execution of each public method. These conditions assure that the object is in “good” state after each manipulation of the class by the client. These invariants are checked after the end of the method and if there are re-entrant call - after the outermost public method.

// code from c6

Object invariants are useful also for setting conditions on automatic properties. Automatic properties don’t have explicit implementations. In this case defining preconditions and postconditions happens through the invariants.

// Code from used guide

As the code shows invariants on auto-property turn into a precondition on a setter, a postcondition on getter and invariant on the backing field. They behave like the other invariants and are called (user guide, p.10, down) TODO

Assume

Assert

* 1. Plusses, minuses
  2. Readability and usability
     1. Static import
     2. Regions
     3. Split the conditions in many
     4. Ordering
     5. Comments and user messages
  3. Pitfalls?
     1. OldValue

1. From C5 to C6 Continued
   1. Testing and doc.
   2. Interface hierarchy

Interfaces - the same

Removed members

Added members – 2

Changed or moved signature

All suffixed methods

Priority queue

Duplicates

Index range

Fixed size collection

Modified behavior

Collection changed – when is thrown

Coll values at return

Default Value for Duplicates By Counting

Bad Enumerables

* 1. Altered and removed classes

…

* 1. View methods + contracts

RequireValidity

1. Implementing the data structures

Collection from Enumerables

Removing Items in Bulk

ArrayBased:

Array’s Utility Methods

List resizing

Do I first cast to IColl then to IColl ?

Collection values returned by 5 methods

... 2 more things: Check and Invariants + Stacks and Queues

* 1. ArrayList with View
  2. HashedArrayList + View
  3. LinkedList + View
  4. HashedLinkedList

1. Testing

The tests from C6 were kept and reused with some changes. The test suit was extended with new tests for view functionality inspired by C5’s unit test. This chapter reflects on the unit testing in C6 Continued and discusses the challenges.

// Something about Test driven development - yes

C6 uses the approach, called Test-*First* [??], where the tests are written first, but this approach doesn’t play essential role in the forming the public API of the library, since it is already defined.

// About NUnit – flexible constraint syntax

// The principles I followed and why? Mikkel’s approach, but it is not enough

“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.”

* 1. Changes made in C6

// something about the existing tests. Did I change anything?

C6 creates unit tests class hierarchy in form of abstract classes that matches the interface hierarchy [C6, 5.3] Each abstract class tests the public methods of one interface. This approach eliminates the duplication of test TODO. I continued with the same approach and the data structures, implemented within this project, used the existing unit tests without the need of write completely new tests for each of the data structures. It was enough to extend the abstract test class and to implement its abstract members. The newly added test for views TODO

Some properties return a fixed value for a given data structure, demanded by its indented behavior. For example, hashed version of the array list doesn’t allow duplicates. This requires the test methods for the IList interface to be configured so that they test properly the methods of any data structures implementing that interface. The tests in C6 generally take care of that fact, but some methods missed to consider it.

[Test]

public void ContainsRange\_SubsetWithDuplicates\_False()

{

Run.If(AllowsDuplicates); - was missing

// Arrange

var count = GetCount(Random) / 2;

var items = GetStrings(Random);

var newItems = items.Take(count).Append(items.First()).ShuffledCopy(Random);

var collection = GetCollection(items, ReferenceEqualityComparer);

// Act

var containsRange = collection.ContainsRange(newItems);

// Assert

Assert.That(containsRange, Is.False);

}

In listing TODO we can see that the method ContainsRange is called with newItems, which contains a duplicated item, items.First(). The result of the assertion should be False, but only if the data structure allows duplicates. This is not the case for HashedArrayList and HashedLinkedList, where they by design behave like a set. If a hashed version of list contains a number x and we call ContainsRange() with only xs, than the method will return true. That’s why the assertion in the listing fails, expecting True. For that reason, couples of test units were updated, putting the given restriction in the beginning of the test methods. Do I know the list TODO.

There are two ICollectionTest methods updated for another reason - *Update\_IntegerCollectionUpdateExistingItem\_RaisesExpectedEvents and* UpdateOrAdd\_IntegerCollectionUpdateExistingItem\_RaisesExpectedEvents

*[Test]*

*public void Update\_IntegerCollectionUpdateExistingItem\_RaisesExpectedEvents()*

*{*

*// Arrange*

*var items = new[] { 4, 54, 56, 8 };*

*var collection = GetCollection(items, TenEqualityComparer.Default);*

*Run.If(!(collection is IList<int>)); -- was missing*

*var count = DuplicatesByCounting ? 2 : 1;*

*var item = 53;*

*var expectedEvents = new[] {*

*Removed(54, count, collection),*

*Added(item, count, collection),*

*Changed(collection)*

*};*

*// Act & Assert*

*Assert.That(() => collection.Update(item), Raises(expectedEvents).For(collection));*

*}*

The test unit in the listing TODO uses TenEqualityComparer. This means that if the integer division by 10 of two items is equal than they are equal. The other important thing in the code is the property DuplicatesByCounting. It says whether the collection stores the duplicates. If it is true, then only one copy is stored and for the other copies an internal counter is incremented. This property is very important for the bags or treebags [C5, 1.4.14], but it is not on the same way important for IList’s classes. Because of their set semantic, HashedArrayList and HasedLinkedList don’t have duplicates, although the DuplicatesByCounting is True by design. However, C5 documentation doesn’t say explicitly that the true value doesn’t have real meaning and it is only to stick to the ??? semantic. The test will fail for this two data structures, because Update method will update only one item equal to the item parameter, therefore it will not raise Remove event for 2 deleted items, as it is forced by the True value of DuplicatesByCounting. The test is edited by putting a condition that this test is relevant only for non-IList data structures (i.e. bag, treebag)

* 1. View tests

A view refers to the items of the underlying list and any change in the items of the underlying list affects the view and vice versa. A modification on a view (remove, insert, add etc.) affects the underlying list [C5, 1.4.11] and as a result it affects all the related views (overlapping, situated in the right ???). This requires me to test the reflections of a view operation on the underlying list. I also need to test how an operation on an underlying list changes its views. This introduces complexity in the tests since I need to check not only the correctness of the object under the question, but correctness of the other objects related to it (the underlying list and the other views).

C6 suggests a reference list of tests that handle often trivial cases and must be executed on each interface method [C6]. The tests cover cases like giving extreme data (negative value) to a method, value greater than count, index out of range, expected events etc. I followed the given list and implemented the relevant once for all view methods. These cases cover part of the aspects that need to be tested for the views, but not the most important parts, which are often method specific.

There are 8 (with Span!) methods for getting a view from a list (no! there are 4) and all the IList<T> public methods that can be executed on views, modifying them. The first approach in testing would be to create views, using the different view creation methods, and for each of these views to write a test for all the public methods. The question here is what exactly is appropriate to assert, since each public method modifies not only the view, but the underlying list and possibly the other views on that list. Having that in mind, the scenarios, we might need to test, explode. All the scenarios available for a list + extra methods… TODO

The second approach would be to test how one operation on a random view affects views in the corner cases (i.e. a view in the beginning and at the end of a list). Then we can assume that if the views in the corner case are changed accordingly, then this would be also true for the other view cases. Following this idea, I decided to implement the subsequent approach: I consider 3 different view types (zero-item view, one-item view and n-item view) in three different positions in the beginning, in the middle and at the end of a list, see TODO picture. They form nine different scenarios. First, I take one of the scenarios, for example a n-item view in the middle, and create a list of views, using all the possible view creation methods (View, ViewOf etc.), see the code TODO. Then I call the public method, that I test, on a view in the list and assert that the auxiliary views’ offset has changed or not, also the reference equality of the view items with the list item. This is repeated for the rest of the views. I will assert only the view.Offset, because it signals that … TODO.

I use multiple asserts in the view tests, although this is one of the strongly unadvised things in testing [7.3.5]. On the one hand this is, because when an assert fails, it throws an exception, and the other asserts will not be executed, if any. But each of the asserts should run at least once, no matter if the previous asserts fail or not. This can be solved by using Assert.Mutliple ?? constrain, introduced in NUnit 3.8. On the other hand, multiple asserts in most cases are completely independent and thus can be separated in different tests. This is not true in these tests, because with the multiple assertions in the code it’s checked the correctness of other objects, which depend on each other, possibly affected of the operation. [Art] also advices to avoid multiple asserts on the same object, but create one expected object instead and check for equality with an actual object. This is again not the case in these tests, because I don’t check the aspects of one object as mentioned above. Thus, multiple asserts are kept in the general view tests.

As can be seen in [code] the test contains for-cycle, iterating of the views. One thing that can be improved her is creating a new test for each view in the test. However, this can lead to too many test units which I avoided here. Instead I output the failed view index TODO

TODO an effect of overlapping views on the underlying list and the list on the views is disregarded in this project, because … TODO

TODO: Anything more about C5’s WRONG view (testing) approach

In C5 the tests use hard-coded test values. A disadvantage of hard-coded values is that the tests could pass for the given values, but can fail for some other. I avoided using collections with fixed item values (really?), random collections are created instead, but I gave fixed parameters to some of the view methods. TODO code

// Code

The reason for that is to create appropriate auxiliary and testing views by the view methods, which is not easy to control by random generators.

TODO: Copy multipleview test from C5, can’t I?

// I could copy the tests, but I didn’t. Instead I updated with this and this.

// duplications

* 1. New test helpers introduced – TODO. Getview, Getcount.

Bibliography

[1] Bertrand Meyer. Applying “Design by Contract”. *Computer*, 25(10):40–51, 1992.

[2] Apache Software Foundation, Apache 2017, URL <http://tomee.apache.org/examples-trunk/bean-validation-design-by-contract/>

[3] Microsoft Research. Spec#, 2006. URL <http://research.microsoft.com/en-us/projects/specsharp>

[4] Ada Information Clearinghouse. Ada, 2012. URL <http://www.adaic.org/>

[5] Nil Packet Minimizer. Dbc Project, 2017, URL <https://www.npmjs.com/package/dbc-code-contracts>

[6] Eiffel Software. Eiffel, 2017. URL <https://www.eiffel.com/>

[7] Code Contracts in Visual Studio 2017, <http://blog.xoc.net/2017/04/code-contracts-in-visual-studio-2017.html>

[8] “Will Code Contracts support Visual Studio 2017?”, URL <https://social.msdn.microsoft.com/Forums/en-US/1f130049-1b15-48b4-ad63-8645be92d972/will-code-contracts-support-visual-studio-2017?forum=codecontracts>

[9] Microsoft. System.Diagnostics.Contracts Namespace, 2017. URL <https://msdn.microsoft.com/library/system.diagnostics.contracts.aspx>

[10] Microsoft. Code Contracts User Manual. Microsoft Corporation, November 2013.