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 is disregarded in this project, because …

TODO: Anything more about C5’s view 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. The reason for that is to create long enough auxiliary views or to create not too big views by the view methods, which is not easy to control by random generators.

TODO: I can 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.