Exercise Set 1: Unit Testing

Software Development 2017

Department of Computer Science
University of Copenhagen

Kristian Fogh Nissen <kristianfoghnissen@gmail.com>
Alexander Christensen <jpb483@alumni.ku.dk>
Oleks Shturmov <oleks@oleks.info>

Version 1;
Due: Friday, February 17, 15:00

When developing software, many programmers tend to focus primarily on the beauty and performance of their code. This is important, but it is equally, if not more important, to pay attention to the correctness of the code. Without proper, reproducible tests it is difficult to have confidence that the software works as intended. This concern is reinforced as the software changes and grows.

"Unit testing" refers to the act of testing of individual "units" of source code. A "unit" is a small, testable part of an application. It is best to design your application in terms of such units, indeed to enable unit testing.

NB! There is a fresh version of the DIKU policy for MonoDevelop / Xamarin Studio. There are no changes to the style guide, but the policy now is more conforming to the style guide. There is also a new version of `su17.sty`. Please use this latest versions.

This exercise set is heavily inspired by Chapter 7, Beautiful Tests, in Beautiful Code, by Andy Oram and Greg Wilson, O'Reilly Media, 2007.

1 `FizzBuzz`

We begin by setting up a simple program called `FizzBuzz`. This program is supposed to print the numbers from 1 to 100, one per line, with the following exceptions: (a) for multiples of 3, print "Fizz"; (b) for multiples of 5, print "Buzz"; and (c) for multiples of *both* 3 and 5, print "FizzBuzz".

1.1. Create a new solution and console project `FizzBuzz`.

1.2. Replace the contents of the `Main` method with this:

```
for (int i = 1; i <= 100; i++)
{
    Console.WriteLine ("{0}", i);
}
```

This should print the numbers from 1 to 100 when run.

1.3. Add a library project to the solution called `Library`.

1.4. Create a static class `Library.Buzzer`, with a public static method named `Buzz` that takes an integer as input and returns a string. The method should return a string as specified above.

Hint: Use modulo (`%`).

1.5. Modify your `Main` method to use the library method above.

1.6. Run `FizzBuzz` to see it in action.

1.7. In your report, discuss which inputs it would be good to test `FizzBuzz` with, and why.

1.8. Implement these tests manually using the `Tester` class from Exercise Set 0 (just copy it into your `FizzBuzz` project). Use it as described there.

1.9. Conclude your report with an assessment of how robust you think your implementation is, based on the outcome of your test results.

2 Hello, NUnit

Manual testing (using either `Console.WriteLine`, or something like the `Tester` class) is quick and painless, but it is not a very robust way to test. NUnit, is an industrial-strength testing framework, that initially works a bit like our `Tester` class, but comes with much more functionality, and even IDE support.

2.1. Add a new NUnit Library Project to your solution: call it `Tests`.

2.2. Your new library should have a file called `Test.cs`, which should have a method called `TestCase ()`. In this method write:

```
Assert.AreEqual (7, 7);
```

2.3. On the right side of your IDE you should have a pad called Unit Test. Click it. This gives you an overview of all the tests in your solution. Click "Run All" to run all the tests.

2.4. Try changing the assert call to the following:

```
Assert.AreEqual (6, 7);
```

Try running the tests again. This is what a failing test looks like. The code you submit should not have any failing tests. You can safely remove the `TestCase` method — it is not part of the submission.

2.5. In your report, discuss what does `Assert.AreEqual` do, in general.

2.6. We turn to testing the `Buzzer.Buzz` method. Right-click on references for the `Tests` project, and add the `Library` project as a reference.

Pro-tip: You should seek to name your test-cases so that it is easy to identify what went wrong when it does. Most `Assert`-style methods are overloaded to also take in a message to show when the test fails. This can be used to further clarify what is going wrong.

2.7. Implement the following "smoke tests": A test `Test3YieldsFizz`, asserting that `Buzzer.Buzz` yields the string "Fizz" for the number 3. The tests `Test5YieldsBuzz`, `Test15YieldsFizzBuzz`, and `Test43Yields43`, having similar semantics.

You are welcome to add other smoke tests you find interesting.

Explanation: These test some "units" of functionality in `Buzzer.Buzz`. Keeping them in separate test cases allows us to quickly single out which part of the functionality fails, when it does.

2.8. Implement the following "border case tests": A test `Test1`, testing that `Buzzer.Buzz` yields the correct value for the number 1. Similarly, write a test `Test100`.

You are welcome to add other border case tests you find interesting.

2.9. The FizzBuzz program was originally defined to only work with the numbers 1 through to 100. The type of `Buzzer.Buzz` however, makes no such restrictions: the method happily accepts values outside the range $[1; 100]$. What happens in your implementation? Discuss possible ways to disambiguate this ambiguity in the exercise set. You are welcome to write further tests for values outside the range $[1; 100]$.

3 Binary Search

To further study the art of testing we consider the *binary search* algorithm. This algorithm is an amazingly fast searching algorithm. Given a sorted, indexed collection of comparable elements, and a target element, it searches the collection to find the index of the target element. If the target element is not in the collection, it returns a sentinel index (typically, -1), indicating failure.

There is an important ambiguity in the above description. It is your task to spot this ambiguity, if not by sheer analysis, then by extensive testing and debugging of the below implementation.

If you need your memory refreshed on how *binary search* works, Wikipedia has an excellent article on the subject.[1]

[1] https://en.wikipedia.org/wiki/Binary_search_algorithm

3.1 Sample Implementation

Alongside the assignment text, we provide a ZIP archive, `BinarySearch.zip`, containing a sample implementation of the binary search algorithm for arrays of `IComparable` objects. The implementation is incomplete. It is your task to finish it.

Note: `IComparable` is an interface[2]. Classes that implement this interface must implement the following method:

`void CompareTo(object other)`
: Compares `this` object to the `other`.

  Returns less than 0, if `this` is less than the `other`; 0 if they are equal; and greater than 0, if `this` is greater than the `other`.

Armed with the ZIP archive, proceed to the following tasks:

3.1. Unzip the archive.

3.2. Find the solution file `BinarySearch.sln`, and open it in your IDE.

3.3. You will find two projects in the solution: A text-user interface (TUI), and a library project.

The library hosts:

- an implementation of binary search in `Search.cs`;
- a method to show the contents of an array in `Show.cs`; and
- a random `IComparable` (`int`) array generator in `Generator.cs`.

The `Main` method in the TUI project uses all of these library features to conduct a handful `Console.WriteLine`-style tests of the binary search implementation.

3.4. Run the TUI project.

3.5. In your report, discuss what the `Main` method does in more detail.

4 Testing Binary Search with NUnit

As hinted in the previous exercise, littering a program with print statements is not a very good way to perform tests. This is where NUnit comes into place!

4.1. Add a new NUnit Library Project to the `BinarySearch` solution.

Call it `Tests`.

4.2. Remove the automatically create `TestCase` method.

4.3. Add 3 test methods:

[2] See also `http://docs.go-mono.com/?link=T:System.IComparable`.

4.3.1. `TestTooLow`, testing that the binary search yields -1 for values less than any given value in the array.

4.3.2. `TestTooHigh`, testing that the binary search yields -1 for values greater than any given value in the array.

4.3.3. `TestElement`, testing that the binary search yields a value other than -1 for any value in the array.

Hint: There exists an `Assert.AreEqual` and an `Assert.AreNotEqual`.

Hint: Before we can run the tests we need a reference to the `Library`. Find the solution navigation panel, right click on the "References" folder under the `Tests` project, and click "Edit References". Add a reference to the `Library` project.

4.4. Create a new member function `TestEmptyArray` (don't forget the `[Test ()]` attribute). Create an empty array. Create a for loop that iterates over the integers $[-100, 100]$. For each loop iteration, assert that a call to *binary search* with the given integer returns -1.

4.5. Now it is time to run the unit tests. Go to the "Unit Tests" panel, and click "Run All". For any errors encountered, make the necessary modifications to the implementation, or tests. All your tests should pass, but don't craft your tests to fit the implementation.

4.6. Create a couple of more test methods in a similar fashion until your are confident that the implementation works as intended.

4.7. Explain in the report what changes you made to the code, how you accommodated these changes in your test file, and argue about the coverage of these tests.

5 Modifications

The following sections ask you to consider a number of explicit problems with our implementation, which might have illuded your attention thus far.

5.1 Arithmetic Overflow

A central part of doing solid unit testing is to test that the code performs well under extreme circumstances. This is also called "border case testing". The first thing we will optimize within the *binary search* algorithm is taking into account arithmetic overflow.

When the handed out implementation is run against a sufficiently large array, the following line will generate an arithmetic overflow:

```
int mid = (high + low) / 2;
```

5.8. Explain the error that might occur in your report.

5.9. Find a solution for the problem and implement it instead of the shown line above. Discuss in the report what you had changed, and why.

5.2 Out-of-Bounds

When searching through a *sorted* array, it can be checked whether or not the target element is in the array without running through the algorithm.

5.10. Add code to conduct these tests before a binary search is commenced. The test-cases `TestTooLow` and `TestTooHigh` from above should still pass.

5.11. Discuss in the report what you have changed, and why.

5.3 Running Time

You might recall from your DMA course that *binary search* has a "logarithmic" running time: By continuously halving the array when searching through it, the algorithm will perform at most $\lceil 1 + log_2 n \rceil$ look-ups for an array of size n. Thus, *binary search* has an asymptotic running time of $O(log_2 n)$.

It is a good idea to verify this using a test. This is where the use of the `IComparable` type becomes in handy.

5.12. Declare a new class, `Library.ComparisonCountedInt`.

The class should have one constructor, taking in an `int` value. This value is to remain constant throughout the life-time of the object: this is just a specialized box around the `int` type.

The class should implement the interface `IComparable`, and so implement the method `CompareTo`, as mentioned above. This should increment an internal counter for the number of comparisons (initialized to 0 in the constructor), but otherwise refer to the implementation of `CompareTo`, already available for `int`.

The class should also expose a read-only property `ComparisonCount`, exposing the count on the number of comparisons performed on the object.

5.13. Declare a public static method in `Library.ComparisonCountedInt`:

`int CountComparisons(ComparisonCountedInt[] array)`
Sums over the comparison counts in the given array.

5.14. Declare at least one test-case with the prefix `TestRunningTime`, checking that the algorithm does not overstep the logarithmic bound.

Hints:

- You can use the provided `Generator` class to generate a sorted array of a given size and magnitude. Initialize an instance of this class in a `[SetUp()]` method of your NUnit test-class.
  You will need to extend the class with a method to generate an array of the proper type (i.e., `ComparisonCountedInt[]`).

- Use `((int)Math.Ceiling(Math.Log (n, 2.0))` to get the base 2 logarithm of `n`.
- Use `Assert.LessOrEqual` or `Assert.GreaterOrEqual`.

5.15. Discuss your implementation and test results in the report.

5.4 Comparing with Linear Search

Linear search is when we traverse at most the entire array

5.16. Declare a public static method `Search.Linear`:

`int Linear(IComparable[] array, IComparable target)`
Performs a linear search of array for the least index of the target element. Returns -1 if no such index exists.

5.17. Declare at least one test-case with the prefix `TestBinaryVsLinear`, checking that `binary search` performs fewer comparisons than *linear search*, on average.

5.18. Discuss your implementation and test results in the report.

6 Submission

You should submit *two* ZIP archives and a PDF document:

- One ZIP archive `FizzBuzz.zip`, containing a folder `FizzBuzz`, containing your `FizzBuzz` solution.
- Another ZIP archive `BinarySearch.zip`, formatted similarly[3].
- A short PDF report documenting your solution.

The following subsections provide further guidance on how to prepare your code and report.

6.1 Report

Write a report. Your report should at least:

- Give an overview of (the extent of) your solution.
- Discuss the non-trivial parts of your implementation.
- Discuss your design decisions, if any.
- Disambiguate any ambiguities in the exercise set.
- If you deviate from the exercise text, tell us where, and why.

[3]Note, how explicit we are about the structure of your ZIP archives. Please do not submit "ZIP bombs" — ZIP archives that extract to a mess of files into the working directory.

6.2 Refactoring

To keep your TA happy, and receive more valuable feedback, you should:

- Make sure the code compiles without errors and warnings.
- Make the code comprehensible (perform adequate renaming, separate long methods into several methods, add comments where appropriate).
- Make sure the code follows our style guide.