Exercise 5

Software Development 2019

Department of Computer Science
University of Copenhagen

Rasmus Hvid Schmidt <lcb852@alumni.ku.dk>,Mads Lind Larsen
<pjz396@alumni.ku.dk>

Version 1.2: February 21th

Due: March 8th

This assignment is individual. Remember to read the entire assignment text before beginning to code

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. 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 take an integer as input and return that same number as a string, with the following exceptions: (a) for multiples of 3, return "Fizz"; (b) for multiples of 5, return "Buzz"; and (c) for multiples of *both* 3 and 5, return "FizzBuzz".

1.1. Create a new solution, name it Exercise5.

1.2. Create a new class library project called `FizzBuzz`.

1.3. Create a static class `FizzBuzz.Buzzer`, with a public static method named `Buzz` that takes an integer as input and returns a string as specified above. You could make a console application to see it in action, however this is not required for the hand-in.

Hint: Use modulo (`%`).

2 Hello, NUnit

Manual testing (using either `Console.WriteLine`, or writing a small testing class) is quick and painless, but it is not a very robust way to test. NUnit, is an industrial-strength testing framework that comes with much more functionality than using the `Console.WriteLine`, and even offers IDE support.

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

2.2. Your new library should have a file called `Test.cs`, which is going to contain our tests.

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.

2.3. Implement the following "smoke tests": A test `Test3YieldsFizz`, asserting that `Buzzer.Buzz` yields the string "Fizz" for the number 3. Furthermore implement 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.

2.4. 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.

3 Comparison Of Abstract Data Types

In the object oriented programming paradigm we're modelling the real world using abstract data types. In this part of the assignment, we will look at comparing abstract data types.

An abstract data type might not be naturally comparable, which is why you as the developer need to explicitly define how they're compared. (How would you for example compare one DIKUmon to another DIKUmon from A2 to each other? By level? Health? Strength? Name? Or a combination?) In the .NET framework the IComparable interface is implemented to allow comparison between abstract data types. Various methods in the .NET framework rely on this interface to be implemented e.g. Array.Sort and ArrayList.Sort.

Classes that implement the IComparable 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`.

In this part of the assignment we're going to declare a Car class, which will implement the IComparable interface. We want to be able to sort a list of Car instances, based on the inferred value of the Car. This value will be calculated based on some of the fields declared in the Car class.

3.1. Add a project to your Exercise5 solution.

3.2. Declare a Car class in the new project.

3.3. Create fields Make, Horsepower, Color, KilometersDriven and KmPerLiter.

3.4. Implement the IComparable interface on the Car class.

3.4.1. Create the CompareTo method.

3.4.2. Based on the fields Horsepower, KilometersDriven and KmPerLiter create a calculation of the value of the Car as follows.

i. Implement a method that based on the fields calculates an inferred value for the Car.

ii. InferredValue = a * Horsepower - b * KilometersDriven + c * KmPerLiter. You decide the paramater values for a, b and c.

3.4.3. Override the ToString method on Car to print Make and the InferredValue.

3.5. Instantiate ten different Car.

3.6. Put them in a list, print the array instances, sort the list using yourList.Sort and print the array instances again.

4 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`

4.1 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.

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

4.1. Unzip the archive.

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

4.3. 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.

4.4. Run the TUI project to see it work.

5 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!

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

Call it `Tests`.

5.2. Remove the automatically created `TestCase` method.

5.3. Add 3 test methods:

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

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

5.3.3. `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.

5.4. 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.

5.5. Now it is time to run the unit tests. Go to the "Tests" panel, and run your tests. 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.

5.6. Create a few more tests to reassure yourself that the implementation works.

6 Modifications

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

6.1 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:

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

6.1. Find a solution for the problem and implement it instead of the shown line above.

6.2 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.

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

6.3 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.

6.3. 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.

The class should implement the interface `IComparable`, and so implement the method `CompareTo`, as mentioned above.

`CompareTo` should increment a private instance variable for the number of comparisons (initialized to 0 in the constructor). Other than that it should refer to the implementation of `CompareTo`, already available for `int`.

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

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

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

6.5. 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`.

6.4 Comparing with Linear Search

Linear search is when we traverse at most the entire array

6.6. Declare a public static method `Library.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.

6.7. Assert that Linear search and Binary search return the same index when searching through the same array. If they do not, why do you think that is? Fix your implementation of Binary search such that it works.

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

7 Submission

You should submit *one* ZIP archive containing the relevant folders, make sure to include the entire solutions and not just individual .cs files etc.

7.1 Clean up your code

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

- Remove commented out code.
- Make sure that your files and folders have the correct names.
- 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.