Exercises: Unit Testing

You can check your solutions in **Judge system**: https://judge.softuni.bg/Contests/3162/Unit-Testing

Use the **provided skeleton** for each of the exercises. For each task you need to **add** the correct **project reference**.

Before submitting your solution, make sure that the **using of the project** that has been tested in an **inactive state**.

```
using NUnit.Framework;
//using P01_Database;
using System;
```

1. Unit Testing the Collection<T> Class

You are given a C# class **`Collection<T>`**, which implements a generic collection, holding an indexed sequence of elements:

```
public class Collection<T>
{
    public int Capacity { get { … } }
    public int Count { get { … } }
    public Collection(params T[] items) { … }
    public void Add(T item) { … }
    public void AddRange(params T[] items) { … }
    public T this[int index] { … }
    public void InsertAt(int index, T item) { … }
    public void Exchange(int index1, int index2) { … }
    public T RemoveAt(int index) { … }
    public void Clear() { … }
    public override string ToString() { … }
}
```

Create new **NUnit Project** and name it **`Collections.Tests`**, add class **`CollectionTests.cs`** and **add project reference**. Write **unit tests** for the class **`Collection<T>`** in the project **`Collections`**. Ensure that the **code coverage** is high and that all interesting cases are covered: test **all public methods**, test the **auto growing** of the underlying array, test with valid and invalid ranges, and try to cover all other **special cases**.

Hints

- Test **all public methods**.
- Think about all **different scenarios**, which can happen, e. g. insert at the start, insert at the end, insert at the middle, insert with auto-grow, insert at invalid position, insert into empty collection, etc.
- Ensure you test each method with **valid and invalid data**.
- Implement a **performance test** with 1 million items, with **timeout**.

You may implement the following unit tests:

© SoftUni – https://softuni.org. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.

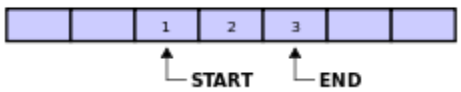
CollectionsTests (31)	145 ms	Test_Collections_InsertAtEnd	< 1 ms
Test_Collections_1MillionItems	95 ms	Test_Collections_InsertAtInvalidIndex	2 ms
Test_Collections_Add	< 1 ms	Test_Collections_InsertAtMiddle	< 1 ms
Test_Collections_AddRange	1 ms	Test_Collections_InsertAtStart	< 1 ms
Test_Collections_AddRangeWithGrow	1 ms	Test_Collections_InsertAtWithGrow	< 1 ms
Test_Collections_AddWithGrow	< 1 ms	Test_Collections_RemoveAll	11 ms
Test_Collections_Clear	< 1 ms	Test_Collections_RemoveAtEnd	< 1 ms
Test_Collections_ConstructorMultipleItems	< 1 ms	Test_Collections_RemoveAtInvalidIndex	3 ms
Test_Collections_ConstructorSingleItem	< 1 ms	Test_Collections_RemoveAtMiddle	< 1 ms
Test_Collections_CountAndCapacity	6 ms	Test_Collections_RemoveAtStart	< 1 ms
Test_Collections_EmptyConstructor	< 1 ms	Test_Collections_SetByIndex	< 1 ms
Test_Collections_ExchangeFirstLast	< 1 ms	Test_Collections_SetByInvalidIndex	2 ms
Test_Collections_ExchangeInvalidIndexes	15 ms	Test_Collections_ToStringCollectionOfCollections	2 ms
Test_Collections_ExchangeMiddle	< 1 ms	Test_Collections_ToStringEmpty	< 1 ms
Test_Collections_GetByIndex	< 1 ms	Test_Collections_ToStringMultiple	< 1 ms
Test_Collections_GetByInvalidIndex	7 ms	Test_Collections_ToStringSingle	< 1 ms

2. Unit Testing the CircularQueue<T> Class

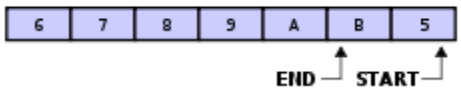
You are given a C# class **`CircularQueue<T>`**, which implements the data structure **circular queue** (learn more at https://en.wikipedia.org/wiki/Circular_buffer):

```
public class CircularQueue<T>
{
    public CircularQueue(int capacity) { … }
    public int Count { get { … } }
    public int Capacity { get { … } }
    public int StartIndex { get { … } }
    public int EndIndex { get { … } }
    public void Enqueue(T element) { … }
    public T Dequeue() { … }
    public T Peek() { … }
    public T[] ToArray() { … }
    public override string ToString() { … }
}
```

The circular queue internally holds its elements in **array**, where the **start** and the **end** of the queue are **array indexes**. In the beginning, the **end index** goes after the **start index**:



After several enqueue / deque operations are executed, the queue elements move right and can **cross the array border** and continue from its leftmost element. In this case the **end index** goes before the **start index**:



The **border crossing** in the circular queue implementation is a little bit tricky and should be handled with care.

© SoftUni – https://softuni.org. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.
Follow us:

Write Unit Tests

In **`Collections.Tests`** project add class **`CircularQueueTests.cs` and project reference.** Your task is to write **unit tests** for the class **`CircularQueue<T>`**. Ensure that the **code coverage** is high and that all interesting cases are covered: test **all public methods**, test the **queue growing**, test the **range crossing** (when the end index comes before the start index) and any other special cases. Test the **private methods** indirectly, but especially designed invocations of the public methods.

Hints

Create a Visual Studio solution, holding two projects:

- **`CircularQueue`** – it will hold the **`CircularQueue<T>`** class, which should be unit tested
- **`CircularQueue.Tests`** – it will hold the **test classes**, which cover the circular queue functionality

You may implement the following **test cases**:

```
class CircularQueueTests
{
    public void Test_CircularQueue_ConstructorDefault() { … }
    public void Test_CircularQueue_ConstructorWithCapacity() { … }
    public void Test_CircularQueue_Enqueue() { … }
    public void Test_CircularQueue_EnqueueWithGrow() { … }
    public void Test_CircularQueue_Dequeue() { … }
    public void Test_CircularQueue_DequeueEmpty() { … }
    public void Test_CircularQueue_EnqueueDequeue_WithRangeCross() { … }
    public void Test_CircularQueue_Peek() { … }
    public void Test_CircularQueue_PeekEmpty() { … }
    public void Test_CircularQueue_ToArray() { … }
    public void Test_CircularQueue_ToArray_WithRangeCross() { … }
    public void Test_CircularQueue_ToString() { … }
    public void Test_CircularQueue_MultipleOperations() { … }
    public void Test_CircularQueue_1MillionItems() { … }
}
```

Testing the Constructor

As start, we can **test the default constructor** of the **`CircularQueue<T>`** class:

```
[Test]
public void Test_CircularQueue_ConstructorDefault()
{
    var queue = new CircularQueue<int>();
    Assert.That(queue.ToString() == "[]");
    Assert.That(queue.Count == 0);
    Assert.That(queue.Capacity > 0);
    Assert.That(queue.StartIndex == 0);
    Assert.That(queue.EndIndex == 0);
}
```

Testing the **constructor with fixed initial capacity** is very similar.

Testing Enqueue / Dequeue / Peek

Next, we can **test the enqueue method**: create a queue, enqueue few elements and check whether the elements in the queue are as expected:

© SoftUni – https://softuni.org. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.

```
[Test]
public void Test_CircularQueue_Enqueue()
{
    var queue = new CircularQueue<int>();
    queue.Enqueue(10);
    queue.Enqueue(20);
    queue.Enqueue(30);
    Assert.That(queue.ToString() == "[10, 20, 30]");
    Assert.That(queue.Count == 3);
    Assert.That(queue.Capacity >= queue.Count);
}
```

Test the **enqueue with auto-grow**: enqueue enough elements for overflow the capacity. This will cause auto-grow.

Test the **dequeue** operation: with **non-empty** and with **empty queue**.

Test the **peek** operation: with **non-empty** and with **empty queue**.

Testing ToArray() and ToString()

Test also the **`ToArray()`** and **`ToString()`** operations, which are quite similar. Just enqueue some data, invoke **`ToArray()`** / **`ToString()`** and assert the result is as expected:

```
[Test]
public void Test_CircularQueue_ToArray()
{
    var queue = new CircularQueue<int>();
    queue.Enqueue(10);
    queue.Enqueue(20);
    queue.Enqueue(30);

    var array = queue.ToArray();

    CollectionAssert.AreEqual(new int[] { 10, 20, 30 }, array);
}
```

Testing the Bounds Crossing

In the circular queue the most interesting moment is when **the end index crosses the bounds** of the internal buffer and comes behind the start index. This situation should be carefully tested. This is an example of test, which causes **range crossing** of the internals buffer bounds:

© SoftUni – https://softuni.org. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.
Follow us:

```
[Test]
public void Test_CircularQueue_EnqueueDequeue_WithRangeCross()
{
    var queue = new CircularQueue<int>(5);
    queue.Enqueue(10);
    queue.Enqueue(20);
    queue.Enqueue(30);
    var firstElement = queue.Dequeue();
    Assert.That(firstElement, Is.EqualTo(10));
    var secondElement = queue.Dequeue();
    Assert.That(secondElement, Is.EqualTo(20));
    queue.Enqueue(40);
    queue.Enqueue(50);
    queue.Enqueue(60);
    Assert.That(queue.ToString() == "[30, 40, 50, 60]");
    Assert.That(queue.Count == 4);
    Assert.That(queue.Capacity == 5);
    Assert.That(queue.StartIndex > queue.EndIndex);
}
```

Combined Test

To test the **`CircularQueue<T>`** class under heavy load with multiple enqueue / dequeue / peek / to string operations and multiple asserts, we can design a special test, which will repeat the following **300 times**:

- Enqueue 2 new elements
- Peek the first element
- Dequeue the first element
- Check the count of elements
- Check the elements, using **`ToArray()`** / **`ToString()`**

The above will ensure that the underlying circular buffer will auto-grow and overflow the borders multiple times. Such test will ensure with high confidence that the class behaves correctly.

This is an example how we can implement such a **complex test with multiple operations and assertions**:

```
[Test]
public void Test_CircularQueue_MultipleOperations()
{
    const int initialCapacity = 2;
    const int iterationsCount = 300;
    var queue = new CircularQueue<int>(initialCapacity);
    int addedCount = 0;
    int removedCount = 0;
    int counter = 0;
    for (int i = 0; i < iterationsCount; i++)...
    Assert.That(queue.Capacity > initialCapacity);
}
```

These are the **operations** and **assertions** we can perform at each iteration:

- Check the **queue size**:

  ```
  Assert.That(queue.Count == addedCount - removedCount);
  ```

- Enqueue **2 new elements**, **peek** and **remove 1 element**. This will guarantee that the circular buffer will move to the right and will cross the bounds of its underlying array at some moment:

© SoftUni – https://softuni.org. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.
Follow us:

```
queue.Enqueue(++counter);
addedCount++;
Assert.That(queue.Count == addedCount - removedCount);

queue.Enqueue(++counter);
addedCount++;
Assert.That(queue.Count == addedCount - removedCount);

var firstElement = queue.Peek();
Assert.That(firstElement == removedCount + 1);

var removedElement = queue.Dequeue();
removedCount++;
Assert.That(removedElement == removedCount);
Assert.That(queue.Count == addedCount - removedCount);
```

- Check the queue content after each iteration:

```
var expectedElements = Enumerable.Range(
    removedCount + 1, addedCount - removedCount).ToArray();
var expectedStr = "[" + string.Join(", ", expectedElements) + "]";

var queueAsArray = queue.ToArray();
var queueAsString = queue.ToString();

CollectionAssert.AreEqual(expectedElements, queueAsArray);
Assert.AreEqual(expectedStr, queueAsString);

Assert.That(queue.Capacity >= queue.Count);
```

Performance Test

Finally, we could implement a simple performance test, which **adds 1 million items** in the queue and **removes 0.5 million items** from it. It could look like this:

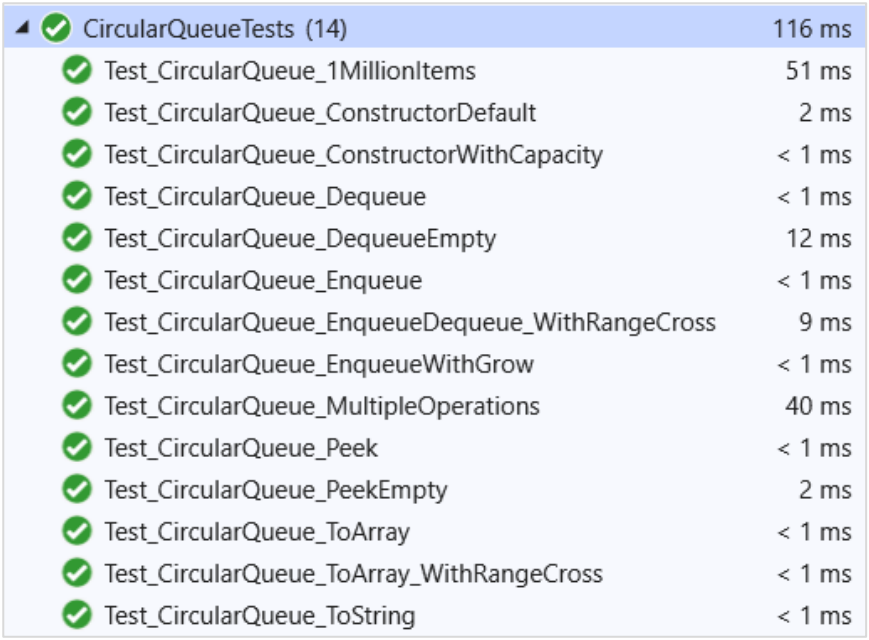
```
[Test]
[Timeout(500)]
public void Test_CircularQueue_1MillionItems()
{
    const int iterationsCount = 1000000;
    var queue = new CircularQueue<int>();
    int addedCount = 0;
    int removedCount = 0;
    int counter = 0;
    for (int i = 0; i < iterationsCount / 2; i++)
    {
        queue.Enqueue(++counter);
        addedCount++;

        queue.Enqueue(++counter);
        addedCount++;

        queue.Dequeue();
        removedCount++;
    }
    Assert.That(queue.Count == addedCount - removedCount);
}
```


© SoftUni – https://softuni.org. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.
Follow us:

This is an example how your unit tests may look like after successful execution:



3. Database

You are provided with a simple class - **Database**. It should **store integers**. **The initial integers should be set by constructor**. They are stored **in array**. **Database** have a functionality to **add**, **remove** and **fetch all stored items**. Your task is to **test the class**. In other words **write tests**, so you are sure its methods are working as intended.

Constraints

- Storing array's **capacity** must be **exactly 16 integers**
  - If the size of the array is not 16 integers long, **InvalidOperationException** is thrown.
- **Add** operation, should **add an element at the next free cell** (just like a stack)
  - If there are 16 elements in the Database and try to add 17th, **InvalidOperationException** is thrown.
- **Remove** operation, should support only removing an element **at the last index** (just like a stack)
  - If you try to remove element from empty Database, **InvalidOperationException** is thrown.
- **Constructors** should take integers only, and store them in **array**.
- **Fetch method** should return the elements as **array**.

Hint

Do not forget to **test the constructor(s)**. They are methods too!

4. Extended Database

You already have a class - **Database**. We have **modified it** and added some **more functionality** to it. It supports **adding, removing and finding People**. In other words, it **stores People**. There are two types of finding methods - first: **FindById (long id)** and the second one: **FindByUsername (string username)**. As you may guess, each person has its own **unique id**, and **unique username**. Your task is to test **the provided project**.

Constraints

Database should have methods:

- Add
  - If there are already users with this username, **InvalidOperationException** is thrown.

© SoftUni – https://softuni.org. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.
Follow us:

    - If there are already users with this id, **InvalidOperationException** is thrown.
- Remove
- FindByUsername
    - If no user is present by this username, **InvalidOperationException** is thrown.
    - If username parameter is null, **ArgumentNullException** is thrown.
    - Arguments are all **CaseSensitive**.
- FindById
    - If no user is present by this id, **InvalidOperationException** is thrown.
    - If negative ids are found, **ArgumentOutOfRangeException** is thrown.

Hint

Do not forget to test the constructor(s). They are methods too! Also keep in mind that all the functionality from the previous task still exists and you need to test it again!

5. Car Manager

You are provided with a simple project **containing only one class** - "**Car**". The provided class is simple - its **main point is to represent some of the functionality of a car**. **Each car contains information** about its **Make**, **Model**, **Fuel Consumption**, **Fuel Amount** and **Fuel Capacity**. Also **each car can add some fuel to its tank by refueling** and **can travel distance by driving**. **In order to be driven**, **our car needs to have enough fuel**. **Everything in the provided skeleton is working perfectly fine** and **you mustn't change it**.

In the skeleton you are provided **Test Project** named "**CarManager.Tests**". There you **should place all the unit tests** you write. The **Test Project** have only **one class** inside:

- "**CarTests**" - here you should place **all code** testing the "**Car**" and **it's functionality**.

Your job now is to **write unit tests on the provided project** and **it's functionality**. You should test exactly **every part** of code inside the "**Car**" class:

- You should test **all the constructors**.
- You should test **all properties** (**getters** and **setters**).
- You should test **all the methods** and **validations inside the class**.

Before you submit your solution to Judge, you should **remove all the references and namespaces referencing the other project**. You should **upload only** the "**CarManager.Tests**" project **holding the class with your tests**. **Remove** the "**bin**" and "**obj**" folders **before** submission.

Constraints

- **Everything in the provided skeleton is working perfectly fine**.
- **You mustn't change anything in the project structure**.
- **You can test both constructors together**.
- **You shouldn't test the auto properties**.
- **Any part of validation should be tested**.
- **There is no limit on the tests you will write but keep your attention on the main functionality**.

6. Fighting Arena

You are provided with a project named "**FightingArena**" containing **two classes** - "**Warrior**" and "**Arena**". **Your task** here is simple - **you need to write tests** on the project **covering the whole functionality**. But before start writing

© SoftUni – https://softuni.org. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.

tests, you need to **get know** with the project's **structure** and **bussiness logic**. Each **Arena** has a **collection of Warriors** enrolled for the fights. On the **Arena**, **Warriors** should be able to **Enroll for the fights** and **fight each other**. Each Warrior has **unique name**, **damage** and **HP**. **Warriors** can **attack** other **Warriors**. Of course there is some kind of validations:

- **Name** cannot be **null**, **empty** or **whitespace**.
- **Damage** cannot be **zero or negative**.
- **HP** cannot be **negative**.
- **Warrior** cannot **attack** if his **HP** are **below 30**.
- **Warrior** cannot **attack Warriors** which **HP** are **below 30**.
- **Warrior** cannot **attack stronger enemies**.

On the **Arena** there should be performed **some validations** too:

- **Already enrolled Warriors** should not be able to **enroll again**.
- **There cannot be fight** if **one of the Warriors** is not **enrolled** for the fights.

In the skeleton you are provided **Test Project** named "**FightingArena.Tests**". There you **should place all the unit tests** you write. The **Test Project** have **two classes** inside:

- "**WarriorTests**" - here you should place **all code** testing the "**Warrior**" and **it's functionality**.
- "**ArenaTests**" - here you should place **all code** testing the "**Arena**" and **it's functionality**.

Your job now is to **write unit tests on the provided project** and **it's functionality**. You should test exactly **every part** of code inside the "**Warrior**" and "**Arena**" classes:

- You should test **all the constructors**.
- You should test **all properties** (**getters** and **setters**).
- You should test **all the methods** and **validations inside the class**.

Before you submit your solution to Judge, you should **remove all the references and namespaces referencing the other project**. You should **upload only** the "**FightingArena.Tests**" project **holding the two classes with your tests**. **Remove** the "**bin**" and "**obj**" folders **before** submission.

Constraints

- **Everything in the provided skeleton is working perfectly fine**.
- **You mustn't change anything in the project structure**.
- **You shouldn't test the auto properties**.
- **Any part of validation should be tested**.
- **There is no limit on the tests you will write but keep your attention on the main functionality**.

SoftUni
© SoftUni – https://softuni.org. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.
Follow us: