**Міністерство освіти і науки України**

**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**

**Факультет інформатики та обчислювальної техніки**

**Кафедра обчислювальної техніки**

**Лабораторна робота №3**

з дисципліни

«Компоненти програмної інженерії. Якість та тестування програмного забезпечення»

на тему

«Unit тестування з використанням методів White Box Testing»

Виконав:

Перевірив:

студент групи ІП-93

Бабарикін Ігор Владиславович

Домінський Валентин Олексійович

номер залікової книжки: 9311

номер у списку: 9

Київ 2021

Зміст

[Мета: 3](#_Toc70859593)

[Завдання: 3](#_Toc70859594)

[Хід роботи: 3](#_Toc70859595)

[Початок роботи: 3](#_Toc70859596)

[Приватні поля 4](#_Toc70859597)

[Init 4](#_Toc70859598)

[GetHash 6](#_Toc70859599)

[HashSha2 8](#_Toc70859600)

[Adler32CheckSum 9](#_Toc70859601)

[Тестування: 11](#_Toc70859602)

[Init 11](#_Toc70859603)

[Результати тестування 11](#_Toc70859604)

[Сирцеві коди: 12](#_Toc70859605)

[TestFileWorkingUtils (тести) 12](#_Toc70859606)

[Висновки: 12](#_Toc70859607)

[Джерела: 12](#_Toc70859608)

Мета:

Написати Unit тести з використанням методів White Box Testing

Завдання:

|  |  |  |
| --- | --- | --- |
| N п/п | 9311 mod 6 | Library |
| 1 | 0, 2, 3 | IIG.BinaryFlag |
| 2 | 1, 4, 5 | IIG.PasswordHashingUtils |

Варіант = 9311 mod 6 = 5, отже провести тестування IIG.PasswordHashingUtils

Отже, мій вибір для лабораторної:

* .NET 5
* Бібліотека для тестування xUnit
* Бібліотека IIG.PasswordHashingUtils

Хід роботи:

Початок роботи:

Я створив проект “Lab3” на .NET 5, додав xUnit Test Project “TestPasswordHashingUtils” та бібліотеку IIG.PasswordHashingUtils:



Оскільки у даній лабораторній Ми використовуємо BBT, то можемо переглянути її вміст та зробити відповідні графи для подальшого використовування їх при написанні тестів.

Давайте пройдемося по коду та спробуємо його проаналізувати:

Приватні поля



Як видно з даного шматку, Ми маємо два приватних статичних поля з певними значеннями за замовченням:

* \_modAdler представляє з себе модуль найбільшого простого числа, меншого, ніж 216 , який буде використовуватися при хешуванні паролю
* \_salt використовується для створення певного хешу Нашого паролю

Init



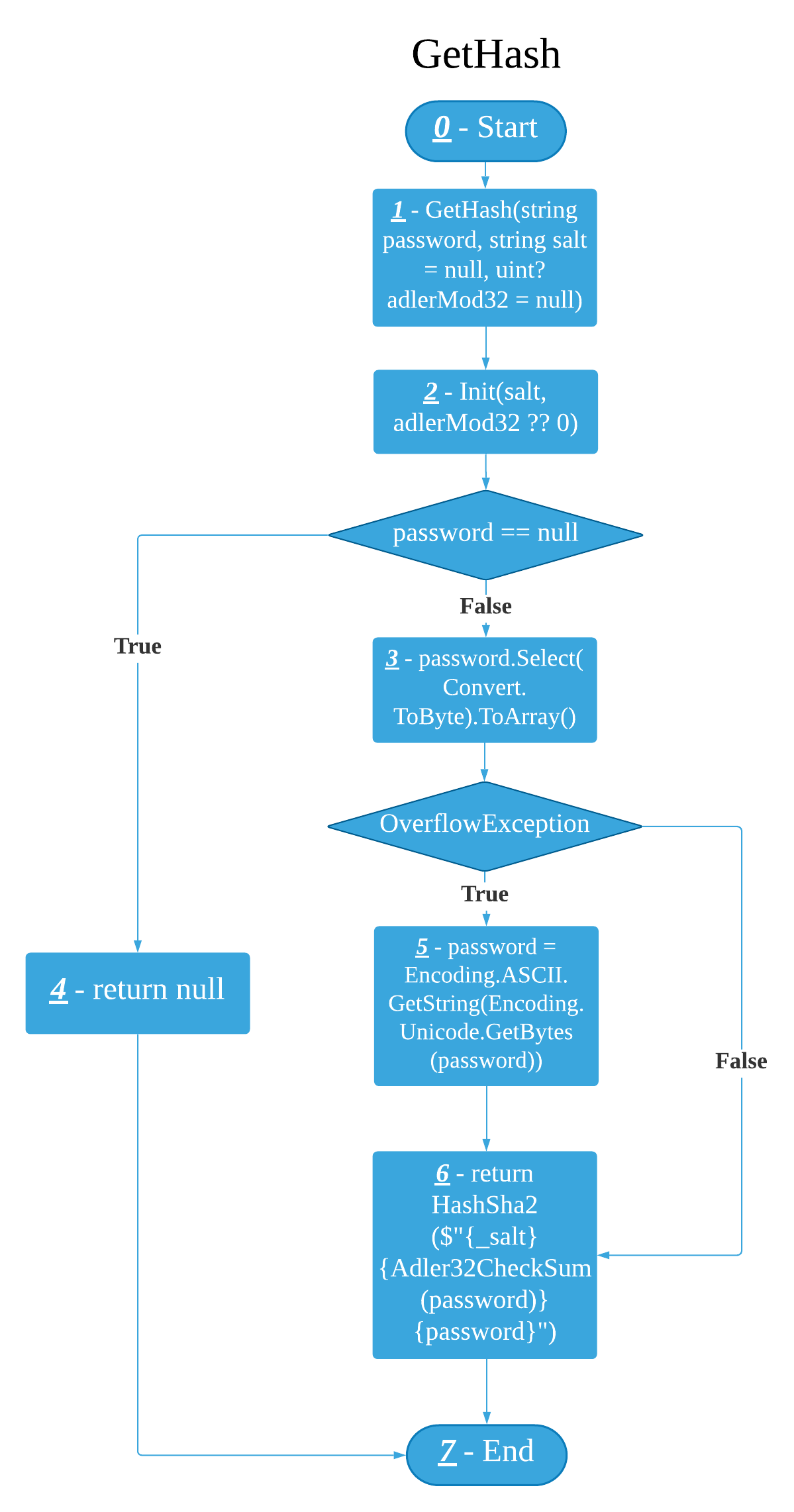
Дана функція перезаписує значення приватних полів, які були вище, з певними перевірками на нові, які приходять як параметри.



GetHash



Цей метод повертає певний хеш для паролю, який Ми передали. Сюди ж можна вписати salt та adlerMod, якщо Нам потрібні якісь конкретні значення.



Далі вже йдуть функції, які є приватними, отже тестувати їх напряму – неможливо

HashSha2



У цьому шматку коду приймається певна стрічка, яка приймає участь у створенні SHA2 хешу.



Adler32CheckSum



І останній метод Adler32CheckSum. Як видно з XML-коментаря дана функція приймає текст, індекс та довжину Adler32CheckSum, а повертає Adler32CheckSum у вигляді стрічки.



Отже Ми можемо дійти до висновку, що напряму тестувати можна лише дві функції: Init та GetHash.

**Тестування**:

Init

Execution Route 0\_1\_6

Почати тестування Я вирішив з методу Init. Але сталася проблема: Я не можу перевірити чи змінилися приватні поля без іншого методу – GetHash. Тобто, якщо повертається один й той же хеш до та після виклику Init, то Ми можемо дійти до висновку, що \_modAdler та \_salt залишилися тими самими. Давайте спробуємо так і зробити:





І тест дійсно пройшов! Тепер зробимо те саме, але з іншими значенням, які підуть по цьому ж маршруту:



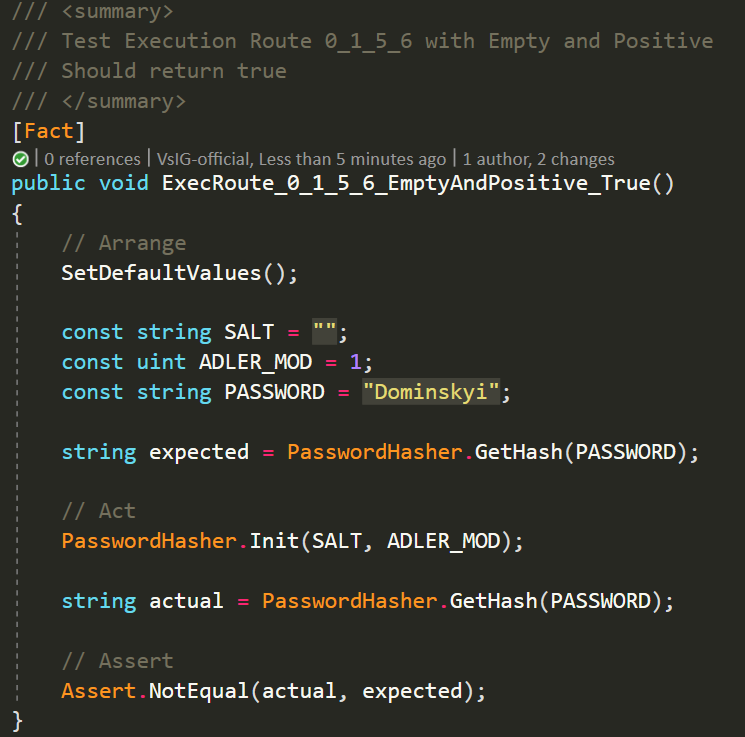
Також були спроби передати від’ємне значення AdlerMod, але оскільки у бібліотеці використовується uint, який не може приймати негативні значення, то Visual Studio видавало помилку.

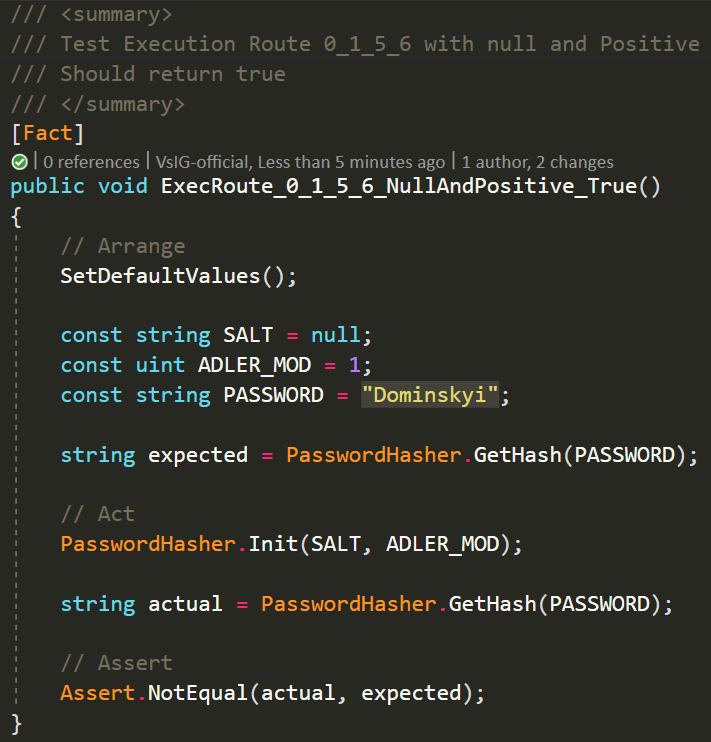
Execution Route 0\_1\_5\_6

А тут Я зрозумів одну річ… усі тести будуть використовувати спільні параметри PasswordHasher’у. Тобто, якщо у тесті, який запускається раніше, йде зміна, наприклад, \_modAdler32, то вона буде помітна в УСІХ подальших тестах, навіть якщо вони знаходяться у різних класах. Через це Я доволі довгий час не міг зрозуміти, чому падає тест. Вирішилось це за допомогою додаткової функції SetDefaultValues, яка викликається на початку кожного тесту.



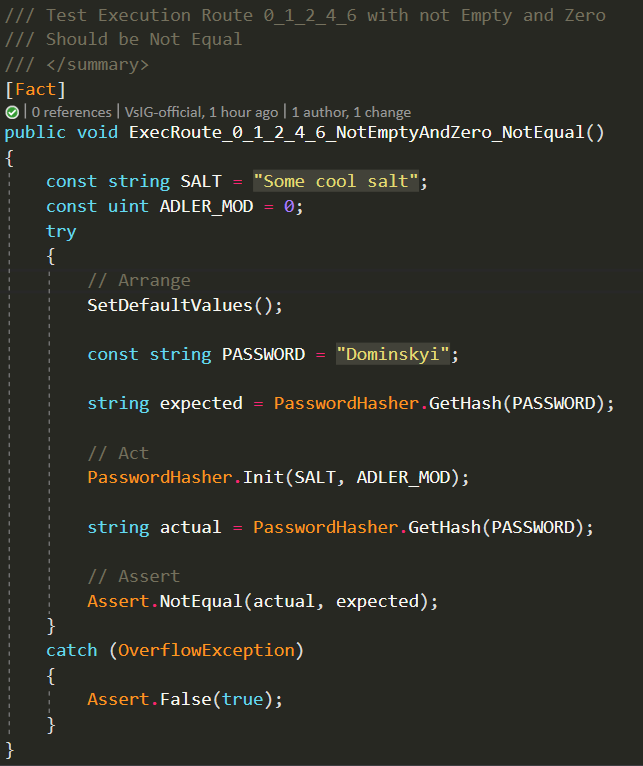
Але давайте перейдемо до самих тестів даного шляху.





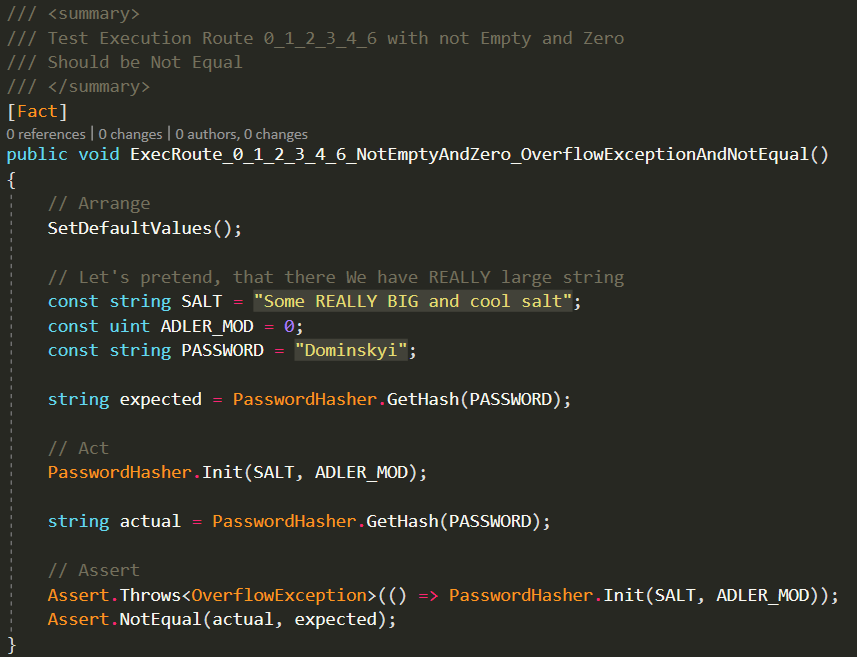
Якщо глянути на діаграму, то можна зрозуміти, що \_adlerMod32 повинен мінятися, оскільки він > 0 (через це й зміниться сам хеш), але не сіль.

Execution Route 0\_1\_2\_4\_6



Даний шлях повинен мати вже сіль, яка дійсно є валідним текстом, на відміну від минулих прикладів, де тестувалися шляхи з передачею порожньої стрічки або взагалі null, але у той же час AdlerMod не повинен мінятися.

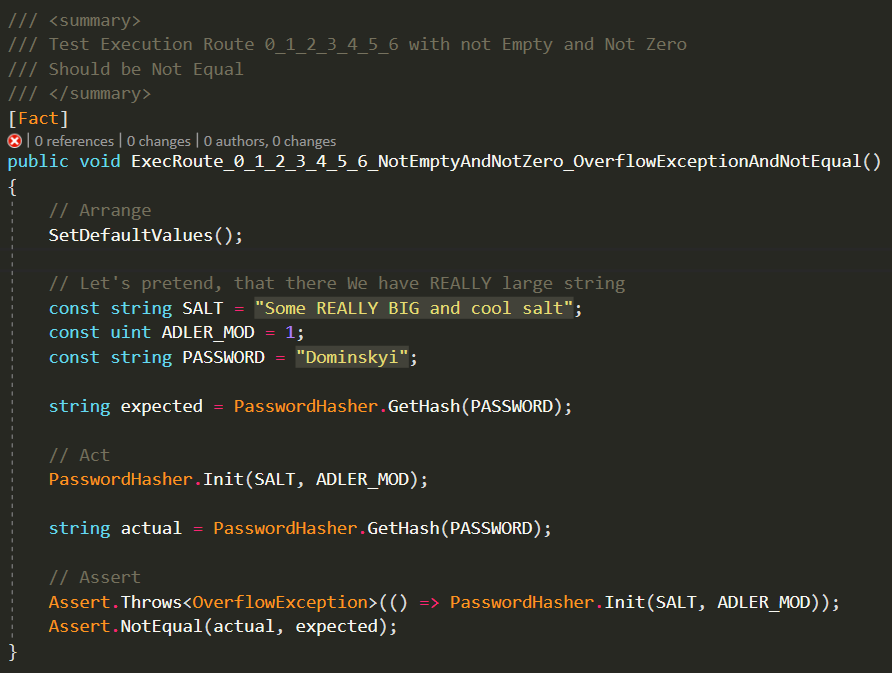
Execution Route 0\_1\_2\_3\_4\_6



А тут Я витратив багато часу на написання тесту і все одно не зміг зробити нічого дійсно підходящого. Щоб даний шлях пройти, треба щоб виникнуло OverflowException - виняток, який видається, якщо виконання арифметичної операції, операції приведення до типу або перетворення в контексті, який перевіряється, призводить до переповнення. Оскільки єдина змінна, якою Ми можемо управляти, аби виникнув цей exception – це string salt, то Я намагався записати в неї максимальну можливу к-сть знаків, на кшталт «string salt = new string('A', 2147483647)» або «string salt = new string('A', int.MaxValue)», але при таких значеннях проект не запускався в онлайн редакторах, а на Моїй локальній машині видавало OutOfMemoryException.

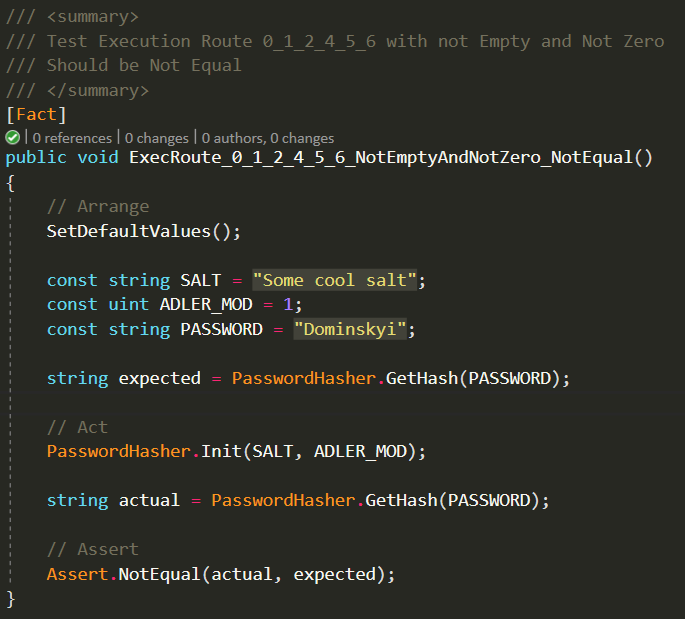
Assert.Throws<OverflowException> перевіряє, чи дійсно дана функція видає exception при певних параметрах і, якщо це так, то тест проходить.

Execution Route 0\_1\_2\_3\_4\_5\_6



Цей шлях дуже схожий на попередній, але adlerMod тут уже виставлений > 0, тому й проходимо через усі можливі операції.

Execution Route 0\_1\_2\_4\_5\_6

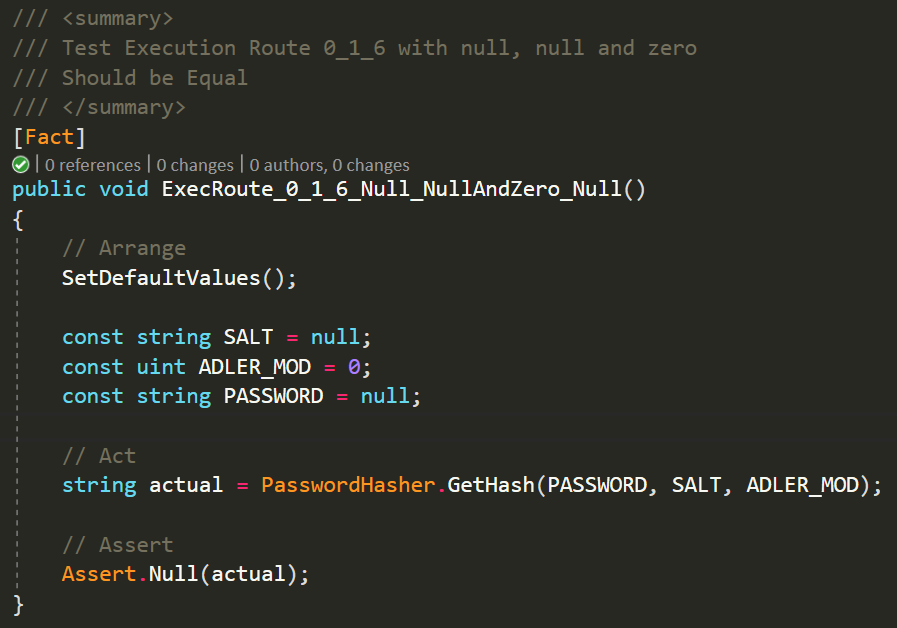


І останній можливий маршрут, який пов’язаний з методом Init. У даному випадку Ми проходимо через усі стадії без усіляких виключень, при цьому маючи дуже надійний хеш для Нашого паролю.

GetHash

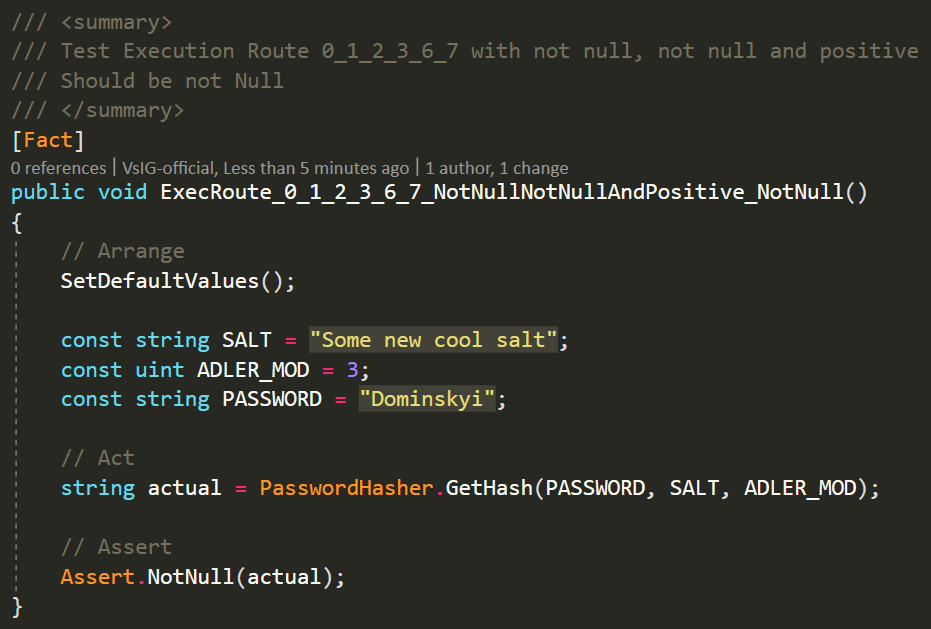
У даній секції Я не буду зачіпати метод Init, оскільки його тестування вже було зроблено

Execution Route 0\_1\_2\_4\_7



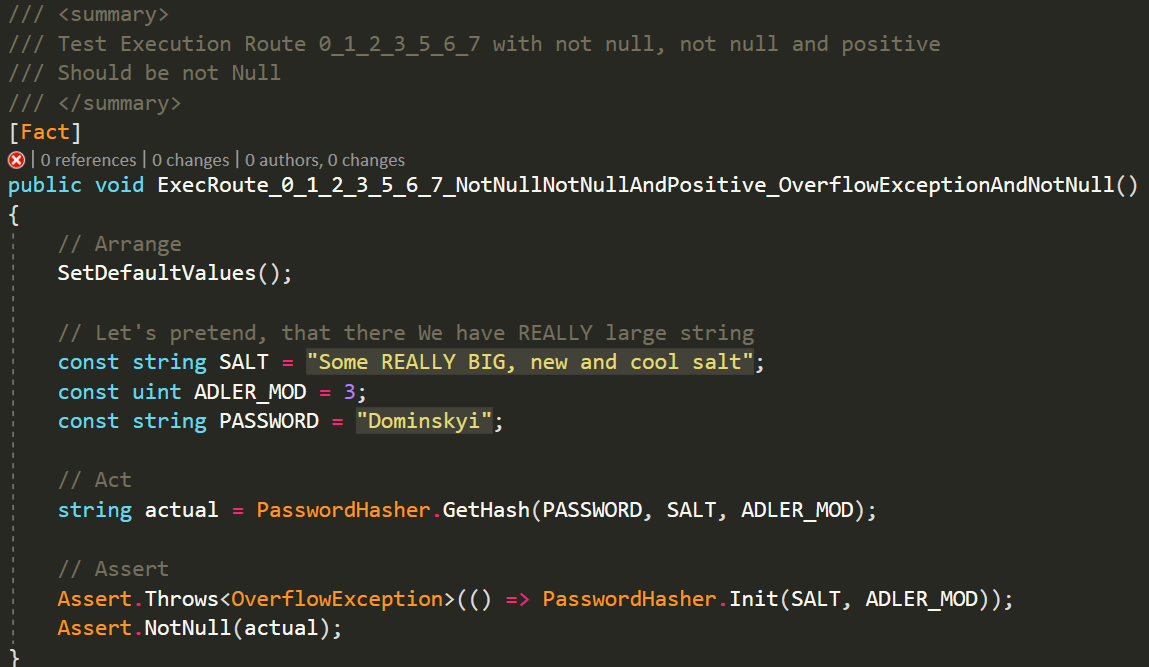
Перший обраний Мною шлях – найпростіший – перевірка паролю на null, та, якщо він дійсно таким є, return того ж null.

Execution Route 0\_1\_2\_3\_6\_7



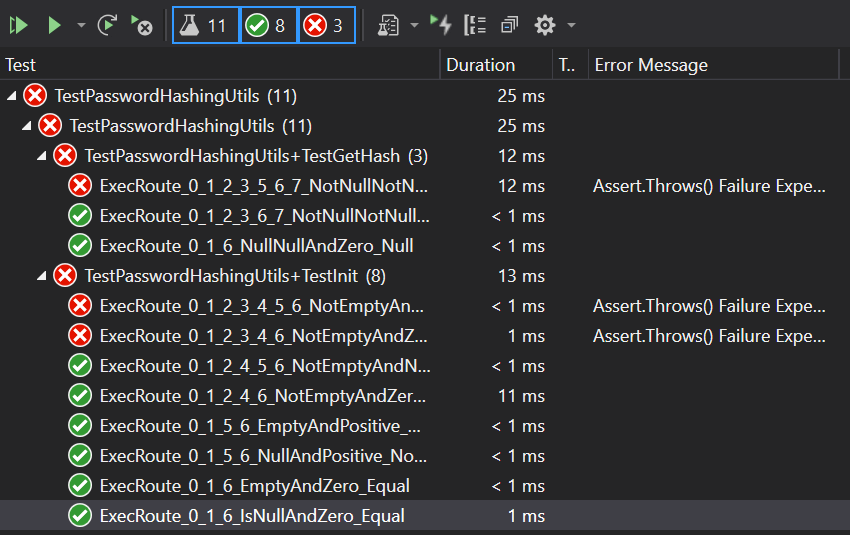
Це, по суті, найкращий для користувача шлях, адже код повністю відпрацьовує своє, не викликаючи жодних виключень.

Execution Route 0\_1\_2\_3\_5\_6\_7



І останній тест – майже копія минулого, з однією відмінністю – у Нас тут знову OverflowException, який трішки змінює алгоритм.

Результати тестування

Як видно з даних результатів, не пройшли успішно лише тести з OverflowException. Саме вони й псують статистику охоплення коду тестами.

Сирцеві коди:

TestPasswordHashingUtils (тести)

**using** System**;**

**using** Xunit**;**

**using** IIG**.**PasswordHashingUtils**;**

**namespace** TestPasswordHashingUtils

**{**

/// <summary>

/// Class for testing Password Hashing Utils

/// </summary>

**public** class TestPasswordHashingUtils

**{**

/// <summary>

/// Sets the default values for PasswordHasher

/// </summary>

**public** static void SetDefaultValues**()**

**{**

const string DEFAULT\_SALT **=** "put your soul(or salt) here"**;**

const uint DEFAULT\_ADLER\_MOD **=** 65521**;**

PasswordHasher**.**Init**(**DEFAULT\_SALT**,** DEFAULT\_ADLER\_MOD**);**

**}**

/\*

Naming:

1. Execution Route being tested.

2. The scenario under which it's being tested.

3. The expected behavior when the scenario is invoked.

\*/

#region ExecutionRoutes

#region Init

/// <summary>

/// Class for testing Init method

/// </summary>

**public** class TestInit

**{**

#region 0\_1\_6

/// <summary>

/// Test Execution Route 0\_1\_6 with null and zero

/// Should be Equal

/// </summary>

**[**Fact**]**

**public** void ExecRoute\_0\_1\_6\_IsNullAndZero\_Equal**()**

**{**

// Arrange

SetDefaultValues**();**

const string SALT **=** **null;**

const uint ADLER\_MOD **=** 0**;**

const string PASSWORD **=** "Dominskyi"**;**

string expected **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Act

PasswordHasher**.**Init**(**SALT**,** ADLER\_MOD**);**

string actual **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Assert

Assert**.**Equal**(**actual**,** expected**);**

**}**

/// <summary>

/// Test Execution Route 0\_1\_6 with Empty and zero

/// Should be Equal

/// </summary>

**[**Fact**]**

**public** void ExecRoute\_0\_1\_6\_EmptyAndZero\_Equal**()**

**{**

// Arrange

SetDefaultValues**();**

const string SALT **=** ""**;**

const uint ADLER\_MOD **=** 0**;**

const string PASSWORD **=** "Dominskyi"**;**

string expected **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Act

PasswordHasher**.**Init**(**SALT**,** ADLER\_MOD**);**

string actual **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Assert

Assert**.**Equal**(**actual**,** expected**);**

**}**

#endregion 0\_1\_6

#region 0\_1\_5\_6

/// <summary>

/// Test Execution Route 0\_1\_5\_6 with Empty and Positive

/// Should be not Equal

/// </summary>

**[**Fact**]**

**public** void ExecRoute\_0\_1\_5\_6\_EmptyAndPositive\_NotEqual**()**

**{**

// Arrange

SetDefaultValues**();**

const string SALT **=** ""**;**

const uint ADLER\_MOD **=** 1**;**

const string PASSWORD **=** "Dominskyi"**;**

string expected **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Act

PasswordHasher**.**Init**(**SALT**,** ADLER\_MOD**);**

string actual **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Assert

Assert**.**NotEqual**(**actual**,** expected**);**

**}**

/// <summary>

/// Test Execution Route 0\_1\_5\_6 with null and Positive

/// Should be not Equal

/// </summary>

**[**Fact**]**

**public** void ExecRoute\_0\_1\_5\_6\_NullAndPositive\_NotEqual**()**

**{**

// Arrange

SetDefaultValues**();**

const string SALT **=** **null;**

const uint ADLER\_MOD **=** 1**;**

const string PASSWORD **=** "Dominskyi"**;**

string expected **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Act

PasswordHasher**.**Init**(**SALT**,** ADLER\_MOD**);**

string actual **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Assert

Assert**.**NotEqual**(**actual**,** expected**);**

**}**

#endregion 0\_1\_5\_6

#region 0\_1\_2\_4\_6

/// <summary>

/// Test Execution Route 0\_1\_2\_4\_6 with not Empty and Zero

/// Should be Not Equal

/// </summary>

**[**Fact**]**

**public** void ExecRoute\_0\_1\_2\_4\_6\_NotEmptyAndZero\_NotEqual**()**

**{**

SetDefaultValues**();**

const string SALT **=** "Some cool salt"**;**

const uint ADLER\_MOD **=** 0**;**

**try**

**{**

// Arrange

const string PASSWORD **=** "Dominskyi"**;**

string expected **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Act

PasswordHasher**.**Init**(**SALT**,** ADLER\_MOD**);**

string actual **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Assert

Assert**.**NotEqual**(**actual**,** expected**);**

**}**

**catch** **(**OverflowException**)**

**{**

Assert**.**False**(true);**

**}**

**}**

#endregion 0\_1\_2\_4\_6

#region 0\_1\_2\_3\_4\_6

/// <summary>

/// Test Execution Route 0\_1\_2\_3\_4\_6 with not Empty and Zero

/// Should be Not Equal

/// </summary>

**[**Fact**]**

**public** void ExecRoute\_0\_1\_2\_3\_4\_6\_NotEmptyAndZero\_OverflowExceptionAndNotEqual**()**

**{**

// Arrange

SetDefaultValues**();**

// Let's pretend, that there We have REALLY large string

const string SALT **=** "Some REALLY BIG and cool salt"**;**

const uint ADLER\_MOD **=** 0**;**

const string PASSWORD **=** "Dominskyi"**;**

string expected **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Act

PasswordHasher**.**Init**(**SALT**,** ADLER\_MOD**);**

string actual **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Assert

Assert**.**Throws**<**OverflowException**>(()** **=>** PasswordHasher**.**Init**(**SALT**,** ADLER\_MOD**));**

Assert**.**NotEqual**(**actual**,** expected**);**

**}**

#endregion 0\_1\_2\_3\_4\_6

#region 0\_1\_2\_3\_4\_5\_6

/// <summary>

/// Test Execution Route 0\_1\_2\_3\_4\_5\_6 with not Empty and Not Zero

/// Should be Not Equal

/// </summary>

**[**Fact**]**

**public** void ExecRoute\_0\_1\_2\_3\_4\_5\_6\_NotEmptyAndNotZero\_OverflowExceptionAndNotEqual**()**

**{**

// Arrange

SetDefaultValues**();**

// Let's pretend, that there We have REALLY large string

const string SALT **=** "Some REALLY BIG and cool salt"**;**

const uint ADLER\_MOD **=** 1**;**

const string PASSWORD **=** "Dominskyi"**;**

string expected **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Act

PasswordHasher**.**Init**(**SALT**,** ADLER\_MOD**);**

string actual **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Assert

Assert**.**Throws**<**OverflowException**>(()** **=>** PasswordHasher**.**Init**(**SALT**,** ADLER\_MOD**));**

Assert**.**NotEqual**(**actual**,** expected**);**

**}**

#endregion 0\_1\_2\_3\_4\_5\_6

#region 0\_1\_2\_4\_5\_6

/// <summary>

/// Test Execution Route 0\_1\_2\_4\_5\_6 with not Empty and Not Zero

/// Should be Not Equal

/// </summary>

**[**Fact**]**

**public** void ExecRoute\_0\_1\_2\_4\_5\_6\_NotEmptyAndNotZero\_NotEqual**()**

**{**

// Arrange

SetDefaultValues**();**

const string SALT **=** "Some cool salt"**;**

const uint ADLER\_MOD **=** 1**;**

const string PASSWORD **=** "Dominskyi"**;**

string expected **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Act

PasswordHasher**.**Init**(**SALT**,** ADLER\_MOD**);**

string actual **=** PasswordHasher**.**GetHash**(**PASSWORD**);**

// Assert

Assert**.**NotEqual**(**actual**,** expected**);**

**}**

#endregion 0\_1\_2\_4\_5\_6

**}**

#endregion Init

#region GetHash

/// <summary>

/// Class for testing GetHash method

/// </summary>

**public** class TestGetHash

**{**

#region 0\_1\_6

/// <summary>

/// Test Execution Route 0\_1\_6 with null, null and zero

/// Should be Null

/// </summary>

**[**Fact**]**

**public** void ExecRoute\_0\_1\_6\_NullNullAndZero\_Null**()**

**{**

// Arrange

SetDefaultValues**();**

const string SALT **=** **null;**

const uint ADLER\_MOD **=** 0**;**

const string PASSWORD **=** **null;**

// Act

string actual **=** PasswordHasher**.**GetHash**(**PASSWORD**,** SALT**,** ADLER\_MOD**);**

// Assert

Assert**.**Null**(**actual**);**

**}**

#endregion 0\_1\_6

#region 0\_1\_2\_3\_6\_7

/// <summary>

/// Test Execution Route 0\_1\_2\_3\_6\_7 with not null, not null and positive

/// Should be not Null

/// </summary>

**[**Fact**]**

**public** void ExecRoute\_0\_1\_2\_3\_6\_7\_NotNullNotNullAndPositive\_NotNull**()**

**{**

// Arrange

SetDefaultValues**();**

const string SALT **=** "Some new cool salt"**;**

const uint ADLER\_MOD **=** 3**;**

const string PASSWORD **=** "Dominskyi"**;**

// Act

string actual **=** PasswordHasher**.**GetHash**(**PASSWORD**,** SALT**,** ADLER\_MOD**);**

// Assert

Assert**.**NotNull**(**actual**);**

**}**

#endregion 0\_1\_2\_3\_6\_7

#region 0\_1\_2\_3\_5\_6\_7

/// <summary>

/// Test Execution Route 0\_1\_2\_3\_5\_6\_7 with not null, not null and positive

/// Should be not Null

/// </summary>

**[**Fact**]**

**public** void ExecRoute\_0\_1\_2\_3\_5\_6\_7\_NotNullNotNullAndPositive\_OverflowExceptionAndNotNull**()**

**{**

// Arrange

SetDefaultValues**();**

// Let's pretend, that there We have REALLY large string

const string SALT **=** "Some REALLY BIG, new and cool salt"**;**

const uint ADLER\_MOD **=** 3**;**

const string PASSWORD **=** "Dominskyi"**;**

// Act

string actual **=** PasswordHasher**.**GetHash**(**PASSWORD**,** SALT**,** ADLER\_MOD**);**

// Assert

Assert**.**Throws**<**OverflowException**>(()** **=>** PasswordHasher**.**Init**(**SALT**,** ADLER\_MOD**));**

Assert**.**NotNull**(**actual**);**

**}**

#endregion 0\_1\_2\_3\_5\_6\_7

**}**

#endregion GetHash

#endregion ExecutionRoutes

**}**

**}**

Висновки:

Виконавши цю лабораторну роботу я познайомився з **White Box тестуванням** у цілому, а також використав таку техніку, як **Тестування потоку виконання** - при тестуванні даним видом тест-кейси створюються таким чином, щоб перевірити правильність виконання максимально можливої кількість шляхів виконання ПЗ

Джерела:

* Github - <https://github.com/VsIG-official/Components-Of-Software-Engineering>
* Program.cs - <https://github.com/VsIG-official/Components-Of-Software-Engineering/blob/master/Labs/Lab2/Lab2/Program.cs>
* TestFileWorkingUtils.cs - <https://github.com/VsIG-official/Components-Of-Software-Engineering/blob/master/Labs/Lab2/TestFileWorkingUtils/TestFileWorkingUtils.cs>
* Офіційна документація - [https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices#characteristics-of-a-good-unit-test](https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices%23characteristics-of-a-good-unit-test)
* Лекція по темі Black Box Testing - [https://docs.google.com/presentation/d/1zDBgNUV73ja\_yShIY-\_WpTGkktol7DOE/edit#slide=id.p18](https://docs.google.com/presentation/d/1zDBgNUV73ja_yShIY-_WpTGkktol7DOE/edit%23slide=id.p18%20)