

Nome e Cognome:

Appello TAP del 16/2/2023

Scrivere nome, cognome e matricola sul foglio protocollo e sul foglio dei quiz. Avete a disposizione due ore e mezza.

## Quiz (9 punti)

Per ciascuna delle seguenti affermazioni, indicate se è vera o falsa

1. Si consideri il seguente frammento di codice

```
public interface IMine {
    int B();
}

public class SUT { /*...*/
    public void TheMethod(int x){/*...*/}
    public SUT(IMine mine){/*...*/}
}

[TestFixture]
public class SUT_Test {
    [Test]
    public void TheMethodDoesNotThrow([Random(1,10,4)] int what) {
        var mine1 = new Mock<IMine>();
        mine1.Setup(x => x.B()).Returns(what);
        new SUT(mine1.Object).TheMethod(3);
    }

    [Test]
    public void BIsUsed() {
        var mine = new Mock<IMine>();
        mine.Setup(x => x.B()).Returns(1024);
        new SUT(mine.Object).TheMethod(5);
        mine.Verify(s=>s.B(), Times.AtLeastOnce);
    }
}
```

PER USARE IL MOCK VERO E PROPRIO

ESISTE SOLO NEI MOCK

Vero Falso

- ☐ ☒ il metodo TheMethodDoesNotThrow non è staticamente corretto, perché non si può applicare l'attributo Random a un parametro
- ☐ ☒ il tipo di mine1.Object non corrisponde a quanto atteso dal costruttore di SUT, quindi il test TheMethodDoesNotThrow non è staticamente corretto
- ☐ ☒ il test TheMethodDoesNotThrow non è staticamente corretto perché un test deve contenere almeno una asserzione
- ☐ ☒ il test TheMethodDoesNotThrow è staticamente corretto ma totalmente inutile perché qualsiasi implementazione lo passa
- ☒ ☒ mine1.Object nel test TheMethodDoesNotThrow viene usata come un mock
- ☒ ☐ mine1.Object nel test TheMethodDoesNotThrow viene usata come uno stub ma non un mock
- ☒ ☐ mine.Object, all'interno del metodo BIsUsed, è un mock
- ☐ ☒ mine.Object, all'interno del metodo BIsUsed, è uno stub ma non un mock
- ☒ ☐ la classe SUT\_Test è staticamente corretta

CONTROLLA CHE  
RITORNI UN  
VALORE (CHECK  
RESULT)

MENTRE MOCK  
CONTROLLA COME  
FUNZIONA  
(BEHAVIOUR)

2. Sia dato il seguente codice

```
public interface ID {
    public string Name { get; }
    public string M();
}

public class D: ID {
    private static Random _random = new Random();
    private static string RandomName() {
        var names = new[] { "Pio", "Edo", "Ada", "Eva", "Ivo", "Zoe" };
        return names[_random.Next(0, names.Length)];
    }
    public string Name { get; init; }
    public string M() { return "ciao "+Name; }
    public D() { Name = RandomName(); }
}

public class C {
    public void M() { Console.WriteLine(new D().M()); }
    public void M(int n){ for (int i = 0; i < n; i++) M(); }
}

public class C1 {
    private ID _myD;
    public C1(ID d) { _myD = d; }
    public void M() { Console.WriteLine(_myD.M()); }
    public void M(int n) { for (int i = 0; i < n; i++) M(); }
}
```

Vero Falso

- ☐ ☒ la classe D non implementa correttamente l'interfaccia ID perché la property Name ha `init` oltre che `get`
- ☐ ☒ la classe D non rispetta i principi della dependency injection perché il costruttore non ha un parametro di tipo `string`
- ☐ ☒ la classe C1 è la corretta modifica della classe C ottenuta mediante dependency injection
- ☐ ☒ la classe C non è staticamente corretta perché non ha nessun costruttore
- ☐ ☒ le chiamate `new C().M(3);` e `new C1(new D()).M(3);` producono sempre lo stesso risultato
- ☐ ☒ le chiamate `new C().M(3);` e `new C1(new D()).M(3);` non sono mai equivalenti, perché la prima genera errore non esistendo il costruttore senza parametri per C
- ☒ ☐ la classe C1 rispetta i principi della dependency injection (senza prendere in considerazione la sua corrispondenza con la classe C)
- ☐ ☒ la classe C rispetta i principi della dependency injection
- ☒ ☐ per modificare la classe C secondo la dependency injection mantenendone il comportamento serve una factory per oggetti di tipo D

è vera, ma non per il motivo della prof, ma perchè è vera quella prima!!!

La classe C non rispetta i principi della dependency injection perché crea direttamente una nuova istanza di D nel suo metodo M(), invece di accettare un'istanza di ID come parametro.

Per modificare la classe C secondo i principi della dependency injection mantenendone il comportamento, è possibile utilizzare una factory per creare gli oggetti di tipo D

Unless the class is static, classes without constructors are given a public parameterless constructor by the C# compiler in order to enable class instantiation

## Esercizio 1 (8 punti)

Si consideri l'interfaccia `Comparable` con il metodo `CompareTo(object?)` che restituisce un numero intero minore di 0 se l'istanza corrente precede l'argomento della chiamata, 0 se sono uguali e un numero maggiore di 0 se lo segue (rispetto all'ordine usato per il confronto).

Scrivere l'extension-method `EnoughSmaller` che, invocato su `s`, una sequenza (eventualmente nulla) di elementi di un tipo generico `T` che estende/implementa `Comparable`, `threshold`, un elemento di tipo `T`, e un intero `howMany`, restituisce `true` se `s` contiene almeno `howMany` elementi che precedono `threshold`. Nel caso in cui `s` non contenga almeno `howMany` elementi che precedono `threshold`, se `s` è finita restituirà `false`. Altrimenti il suo comportamento non è definito (=potete fare quello che volete). Per esempio, sulla sequenza

source = 

1.5
-----

5.2
-----

-2.1
------

31.5
------

5.4
-----

-5.1
------

6.7
-----

`source.EnoughSmaller(4.3, 4) == false`, `source.EnoughSmaller(4.3, 3) == true`, `source.EnoughSmaller(100.6, 11) == false`

Il metodo dovrà prendere come parametro "this" `s`, la sequenza sorgente, **che può anche essere infinita**, un elemento di tipo `T` `threshold`, e un intero `howMany`.

Il metodo deve sollevare l'eccezione `ArgumentNullException` se `s` è `null` o `threshold` è `null`.

Il metodo deve sollevare l'eccezione `ArgumentOutOfRangeException` se `howMany` non è strettamente positivo.

La dichiarazione del metodo deve tenere conto degli aspetti di nullabilità dei tipi reference nella scelta dei tipi e usare opportuni vincoli per limitarne l'istanziabilità ai soli tipi che estendono/implementano `Comparable`.

## Esercizio 2 (8 punti)

Implementare, usando NUnit, i seguenti test relativi a `EnoughSmaller`, dell'esercizio 1.

1. Input della chiamata sotto test: `s` è una sequenza non vuota di caratteri a vostra scelta, `threshold` è un carattere a vostra scelta, `howMany` è 0.

Output atteso: `ArgumentOutOfRangeException`

2. Input della chiamata sotto test: `s` è una sequenza di 5 stringhe a vostra scelta, `threshold` è una stringa a vostra scelta, `howMany` è 42.

Output atteso: `false`

3. Test parametrico con un parametro intero `n`. Se `n` non è strettamente maggiore di zero il test dovrà risultare *inconclusive*.

Input della chiamata sotto test: `s` è una sequenza infinita di elementi di tipo `double` tutti minori di zero, `threshold` è 7.42, `howMany` è `n`.

Output atteso: `true`.

4. Input della chiamata sotto test: `s` è una sequenza di lunghezza 20 di elementi tutti più piccoli di `threshold`, `howMany` è 7.

Il test dovrà verificare che il metodo `CompareTo(object?)` sia stato invocato 7 volte complessivamente su `threshold` e gli elementi di `s`, cioè che contando il numero di chiamate al metodo su `threshold` e su ciascun elemento di `s` il risultato sia 7.