TAP: Appello del 9/6/2022

Scrivere nome, cognome e matricola sul foglio protocollo e sul foglio con le risposte al quiz. Avete a disposizione due ore e mezza.

Esercizio 1 (9 punti)

Scrivere un metodo statico `InterleavingApply <T>` che, date due sequenze `s1` e `s2` di elementi di tipo `T` e una funzione `f` da `T` in `TRes`, produce una sequenza di elementi di tipo `TRes` contenente l'applicazione di `f` alternativamente agli elementi di `s1` e di `s2`. Ad esempio, se `s1` è la sequenza {"pippo", "archimede", "minnie"}, `s2` è la sequenza {"topolino", "qui", "basettoni"} e `f` è la funzione che restituisce il primo carattere, il risultato sarà la sequenza {'p', 't', 'a', 'q', 'm','b'}.

Si noti che gli argomenti `s1` e `s2` di `InterleavingApply` e quindi il suo risultato possono rappresentare sequenze infinite.

Il metodo dovrà sollevare

- `FunctionApplicationException` se una chiamata di `f` solleva eccezione, *propagando l'errore*;
- `DifferentLengthException` se `s1` e `s2` sono entrambe finite ma di lunghezza diversa, inizializzando opportunamente la proprietà `FirstIsLonger` di `DifferentLengthException` per indicare se quella più lunga è la prima (`s1`) o la seconda (`s2`).

Le eccezioni da utilizzare sono definite come segue

```
[Serializable]
public class DifferentLengthException : Exception {
  public bool FirstIsLonger { get; }
  public DifferentLengthException(bool firstIsLonger=true) {
    FirstIsLonger = firstIsLonger; }
  public DifferentLengthException(string message, bool firstIsLonger=true) :
    base(message) { FirstIsLonger = firstIsLonger; }
  public DifferentLengthException(string message, Exception inner,
                                  bool firstIsLonger=true) :
    base(message, inner) { FirstIsLonger = firstIsLonger; }
}
[Serializable]
public class FunctionApplicationException : Exception {
  public FunctionApplicationException() { }
  public FunctionApplicationException(string message) : base(message) { }
  public FunctionApplicationException(string message, Exception inner) :
    base(message, inner) { }
}
```

Esercizio 2 (7 punti)

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

1. Input della chiamata sotto test: `s1` è la sequenza {8, 11, 35}, `s2` è la sequenza {100, 34, 23} e `f` è la funzione che calcola la divisione intera del suo input per 7.
   Output atteso: la sequenza {1, 14, 1, 4, 5, 3}.
2. Test parametrico con due parametri, `a1` e `a2`, di tipo array di carattere. Se i due parametri hanno la stessa lunghezza il test dovrà risultare *inconclusive*.
   Input della chiamata sotto test: `a1`, `a2` e la funzione che dato un carattere restituisce vero se questo è una cifra
   Output atteso: una eccezione di tipo `DifferentLengthException` avente la proprietà `FirstIsLonger` vera se e solo se `a1` ha più elementi di `a2`.
3. Input della chiamata sotto test: `s1` è la sequenza infinita che restituisce ciclicamente gli elementi {7, 5, 17}, `s2` è la sequenza infinita che restituisce ciclicamente gli elementi {2, 18, 42, 128, 512} e `f` è la funzione che dato un intero restituisce vero se e solo se questo è pari.
   Output atteso: una sequenza i cui primi 100 elementi sono {false, true, false, true, ..., false, true} (i primi 100 elementi della sequenza infinita che restituisce ciclicamente gli elementi {false, true})

Esercizio 3 (9 punti)

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

1. Si consideri il seguente frammento di un progetto che usa l'Entity Framework

```
public class A {
  public int AId { get; set; }
  public int BId { get; set; }
  public List<B> MyBs { get; set; };
}
[Index(nameof(X),nameof(Y),IsUnique = true)]
public class B {
  public int BId { get; set; }
  public int X { get; set; }
  public int Y { get; set; }
  public List<A> MyAs { get; set; };
}
public class C {
  public int CId { get; set; }
  public int AId { get; set; }
  public A A { get; set; }
}
public class MyDbContext: DbContext {
  public DbSet<A> As { get; set; }
  protected override void OnConfiguring(DbContextOptionsBuilder options) {...};
}
```

Vero	Falso	
☐	☒	l'EF non può identificare entità in questo frammento perché nessuna classe estende la classe base `Entity`, radice per l'EF di tutte le entità
☒	☐	la classe `A` è un'entità, ovvero ha una tabella corrispondente sul DB
☒	☐	la classe `B` è un'entità, ovvero ha una tabella corrispondente sul DB
☒	☐	la classe `C` è un'entità, ovvero ha una tabella corrispondente sul DB
☐	☒	per rappresentare questo frammento, l'EF genera meno di 3 tabelle sul DB → ne genera 3
☒	☐	per rappresentare questo frammento, l'EF genera esattamente 3 tabelle sul DB
☐	☒	per rappresentare questo frammento, l'EF genera più di 3 tabelle sul DB
☒	☐	`MyDbContext` non è staticamente corretto perché non ha property di tipo `DbSet<B>` e di tipo `DbSet<C>`
☐	☒	l'EF non può identificare entità in questo frammento perché nessuna classe è annotata con l'attribute `EntityAttribute`
☒	☐	la definizione della classe `MyDbContext` è indispensabile per determinare quali sono le entità
☐	☒	la property `BId` nella classe `A` è una proprietà di navigazione verso `B`
☒	☐	le property `MyBs` in `A` e `MyAs` in `B` rappresentano i due lati di una relazione molti-a-molti per l'EF
☐	☒	le property `MyBs` in `A` e `MyAs` in `B` rappresentano due relazioni uno-a-molti distinte per l'EF
☒	☐	il codice `var b1= new B() { X = 1, Y = 1 }; var b2= new B() { X = 1, Y = 1 };` solleva eccezione per violazione di indice *unique*
☒	☐	nella tabella associata a `B` ci possono essere più righe che differiscono solo per il valore della chiave primaria
☐	☒	l'EF interpreta la property `AId` nella classe `C` come la chiave della proprietà di navigazione `A`
☐	☒	l'EF interpreta la property `BId` nella classe `A` come la chiave della proprietà di navigazione `MyBs`
☐	☒	le classi `A`, `B` e `C` sono dinamicamente scorrette perché non hanno un costruttore

2. Si consideri il seguente frammento di codice

```
public static class C {
  public static int F(int x, int y) {
    if(x>y) return x;
    if (y > x) return y;
    throw new ArgumentException(message:"", paramName:nameof(x));
   }
  public static int G(int x, int y) {
    if (x > y) return x;
    if (y > x) return y;
    throw new ArgumentException(message: nameof(x));
  }
}
[TestFixture]
public class Test {
  [Test]
  public void TF1() {
    Assert.That(()=>C.F(3,3),Throws.TypeOf<ArgumentException>()
                                    .With.Property("ParamName").EqualTo("x"));
  }
  [Test]
  public void TF2() {
    Assert.That(() => C.F(3, 3), Throws.InstanceOf<Exception>());
  }
  [Test]
  public void TF3() {
    Assert.That(() => C.F(3, 3), Throws.TypeOf<Exception>());
  }
  [Test]
  public void TG1([Random(10,50,3)] int x, [Random(51, 100, 3)] int y) {
    Assert.That(C.G(x, y), Is.EqualTo(y));
  }
  [Test]
  public void TG2() {
    Assert.That(() => C.G(3, 3), Throws.TypeOf<ArgumentException>()
                                    .With.Property("ParamName").EqualTo("x"));
  }
  [Test]
  public void TG3() {
    var result = C.G(3, 3);
    Assert.That(result, Throws.InstanceOf<ArgumentException>());
  }
 }
```

Vero	Falso	
☒	☐	Il test TF1 ha successo
☐	☒	Il test TF1 non è corretto perché usa metodi inesistenti
☒	☐	Il test TF2 ha successo
☐	☒	Il test TF3 ha successo
☒	☐	Il test TG1 ha successo
☒	☐	Il test TG1 è un test parametrico che viene tradotto da NUnit in 9 test base
☐	☒	Il test TG1 non è corretto, perché l'attributo Random va usato all'interno dell'attributo TestCase e non sui parametri
☐	☒	Il test TG2 ha successo
☐	☒	Il test TG3 ha successo → manca la lambda