**Ambientes virtuais de Execução – 2º Teste de Época Normal – 1 de fevereiro de 2019**
**2018/2019 Semestre de Inverno - Duração 2h30**

**Número:** _____ **Nome:** _____

Nas questões 1 a 3, marque cada alternativa como verdadeira (V) ou falsa (F). Uma alternativa assinalada corretamente conta 0,5 valores, incorretamente desconta 0,25 valores ao total da respectiva questão.

1)

```
class R { public int i; }
struct V { public int i; }
... public static void Main() {
   R r = new R();   R r2 = new R();
   V v = new V();   V v2 = new V();
   Console.WriteLine(v.Equals(v));
   Console.WriteLine(v.Equals(v2));
   Console.WriteLine(object.ReferenceEquals(v, v));
   Console.WriteLine(r.Equals(r));
   Console.WriteLine(r.Equals(r2));
}
```

Relativamente ao código acima...

a) ___ ... são realizadas duas operações de *box* e o resultado é: `true`, `true`, `true`, `true`, `false`.

b) ___ ... são realizadas cinco operações de *box* resultantes das chamadas a `WriteLine(object)`.

c) ___ ... são realizadas quatro operações de *box* e o resultado é: `true`, `true`, `false`, `true`, `false`.

d) ___ ... é possível eliminar algumas operações de *box* ocorridas em `Main` sem alterar o código de `Main` e sem alterar a categoria de `R` e `V`, mas fazendo outras alterações ao código apresentado.

2) Com as classes do espaço de nomes `Reflection.Emit`...

a) ___ ... é possível definir um *assembly* dinâmico que invoca métodos de outro *assembly* mas só se este último não for compilado a partir de uma linguagem fonte .Net que utilize ponteiros (ex: C .Net).

b) ___ ... não é possível gerar código *não verificável* pois o programa emitirá erros de execução ao gerar o tipo.

c) ___ ... não é possível gerar múltiplos *assemblies* dinâmicos no mesmo programa, mas apenas um.

d) ___ ... é possível definir um tipo que deriva de um tipo definido noutro *assembly* dinâmico.

3)

```
class A {
  public virtual void M() {
    Console.WriteLine("A");
} }
```

```
class B : A {
  public override void M() {
    Console.WriteLine("B");
} }
```

```
class C : B {
   public void M() {
      Console.WriteLine("C");
} }
```

Considerando a seguinte definição, a execução de **`Printer.Print(new C())`**; ...

```
class Printer {
  public static void Print(C c) {
    ((A)c).M();
    ((B)c).M();
    c.M();
  }
}
```

a) ___ ... imprime: C, C, C

b) ___ ... imprime: B, B, C

Caso seja feita a alteração seguinte:

```
interface I { void M(); }
class A : I { ... }
class Printer { public static void Print(I c) { ... }
```

a execução de **`Printer.Print(new C())`**; ...

c) ___ ... imprime: B, B, B

d) ___ ... imprime: A, B, C

4. [2,5] Escreva em IL o código do construtor e do método **New** da classe A.

```
delegate int Func(object p);
interface IFixture {
   object New();
}
class A : IFixture {
   public Func Handlers { get; set; }
   private int max;
   public A(int max) {
      this.max = max;
      Handlers = M;
   } ...
   private static int M(object obj) {
      Console.WriteLine("M: {0}", obj);
      return 10;
   }
   public object New() {
      return Handlers(max);
   }
}
```

5. [2,5] Acrescente à interface `IEnumerable<T>` suporte para a operação ***lazy* genérica `UnzipMany`**, que recebe uma sequência de sequências de `T`, que contém sequências misturadas (i.e., a **primeira sequência** contém o **primeiro** elemento de cada sequência original, a **segunda sequência** contém o **segundo** elemento de cada sequência original, etc.), e produz uma nova sequência de sequências de `T` (i.e. `IEnumerable<IEnumerable<T>>`) obtida separando as sequências fonte. As sequências deverão ser *lazy*. O troço de código seguinte ilustra o comportamento pretendido:

```
List<IEnumerable<object>> tuples = new List<IEnumerable<object>>();
tuples.Add(new object[] { "a", 1, "A" });
tuples.Add(new object[] { "b", 2, "B" });
tuples.Add(new object[] { "c", 3, "C" });
IEnumerable<IEnumerable<object>> original = tuples.UnzipMany();
foreach (IEnumerable<object> s in original) {
   Console.WriteLine(String.Join(",", s));
}
```

```
Standard Output:

a,b,c
1,2,3
A,B,C
```

6. [9] Considere o seguinte exemplo de utilização de `TestSuite` com o *output* apresentado:

```
TestSuite tester = new TestSuite(new ConsoleReport());
tester.Add(new TestAdd());
tester.Add(new TestLength());
tester.Run();
```

*Output:*

*OK TestAdd*
*FAILED TestLength: Expected 7 but actual is 4*

```
class TestAdd : UnitTest {
  public override string Name {
    get { return "TestAdd"; }
  }
  public override void Test() {
    AssertEquals(7, 3 + 4);
} }
```

```
class TestLength : UnitTest {
  public override string Name {
    get { return "TestLength"; }
  }
  public override void Test() {
    AssertEquals(7, "ISEL".Length);
} }
```

Fazem parte desta solução as seguintes classes:

```
public class TestSuite {
  List<UnitTest> tests;
  IReport report;
  public TestSuite(IReport consoleReport) {
    this.tests = new List<UnitTest>();
    this.report = consoleReport;
  }
  public void Add(UnitTest ut) {
    tests.Add(ut);
  }
  public void Run() {
    foreach (UnitTest ut in tests) {
      try {
        ut.Test();
        report.Ok(ut);
      }
      catch (AssertException e) {
        report.Fail(ut, e);
      }
    }
}
```

```
abstract class UnitTest {
  public abstract string Name { get;  }
  public abstract void Test();
  public static void AssertEquals<T>(T expected, T actual) {
    if (!expected.Equals(actual))
      throw new AssertException(
        "Expected " + expected + " but actual is " +
                                                  actual);
  }
```

```
public class ConsoleReport : IReport {
  public void Fail(UnitTest ut, AssertException e) {
    Console.WriteLine("FAILED " + ut.Name + ": " +
                                                e.Message);
  }
  public void Ok(UnitTest ut) {
    Console.WriteLine("OK " + ut.Name);
  }
}
```

```
class AssertException : Exception {
  public AssertException(string message) : base(message) { }
}
```

Em cada uma das alíneas seguintes i**mplemente todos os tipos auxiliares necessários.**

a) [3] **Modificando APENAS** o método `void Add(UnitTest ut)` da classe `TestSuite` e sem **adicionar** novos campos, dê suporte para que um teste unitário possa indicar o tipo de excepção esperada. Exemplo:

```
[Expected(typeof(DivideByZeroException))]
class TestDiv : UnitTest {
  public override string Name { get{return "TestDiv"; }}
  public override void Test() {
    int zero = 0;
    int res = 7 / zero;
  }
}
[Expected(typeof(FormatException))]
class TestDivFail : UnitTest {
  public override string Name { get{return "TestDivFail";}}
  public override void Test() { int res = 7 / 3; }
}
```

```
TestSuite tester = new TestSuite(...);
tester.Add(new TestDiv());
tester.Add(new TestDivFail());
tester.Run();
```

*Output:*

*OK TestDiv*
*FAILED TestDivFail: Expected FormatException exception not thrown!*

b) [2] **SEM adicionar** novos campos e **nem modificar** nenhum dos métodos da classe `TestSuite` implemente o método `void Add(string name, Action handler)` na classe `TestSuite` que adiciona um teste unitário para o delegate `handler` recebido por parâmetro. Exemplo:

```
TestSuite tester = new TestSuite(new ConsoleReport());
tester.Add("TestAdd",
    () => UnitTest.AssertEquals(11, 8 + 3));
tester.Add("TestMul",
    () => UnitTest.AssertEquals(444, 11 * 4));
tester.Add("TestSub",
    () => UnitTest.AssertEquals(7, 11 - 4));
tester.Run();
```

*Output:*

*OK TestAdd*
*FAILED TestMul: Expected 444 but actual is 44*
*OK TestSub*

c) [4] **SEM adicionar** novos campos e **nem modificar** nenhum dos métodos da classe `TestSuite` implemente o método `void Add(Type klass)` na classe `TestSuite` que adiciona um teste unitário para cada método público (estático ou de instância) de `klass`, sem parâmetros e anotado com o *custom attribute* `Test`.
Exemplo:

```
class TestCalculator {
  [Test] public static void TestAdd() {
    UnitTest.AssertEquals(11, 8 + 3);
  }
  [Test] public void TestMul() {
    UnitTest.AssertEquals(444, 11 * 4);
  }
  [Test] public void TestSub() {
    UnitTest.AssertEquals(7, 11 - 4);
  }
}
```

```
TestSuite tester = new TestSuite(new ConsoleReport());
tester.Add(typeof(TestCalculator));
tester.Run();
```

*Output:*

*OK TestAdd*
*FAILED TestMul: Expected 444 but actual is 44*
*OK TestSub*