Programación orientada a objetos. Examen julio 2007.

Lunes 16 de julio de 2007

Sea el siguiente programa:

```
using System;
using System.Collections.Generic;

public interface Disco
{
   String Nombre { get; }
}

public class CD : Disco
{
   private String nombre;
   public String Nombre { get { return nombre; } }

   private String artista;
   public String Artista { get { return artista; } }

   public CD(String nombre, String artista)
   {
      this.nombre = nombre;
      this.artista = artista;
   }
   public Boolean MusicaFavorita()
   {
      return Artista.Equals("Buitres");
   }
}

public class DVD : Disco
{
   private String nombre;
   public String Nombre { get { return nombre; } }

   private String[] actores;
   public String[] Actores { get { return actores; } }

   public DVD(String nombre, params String[] actores)
   {
      this.nombre = nombre;
      this.actores = actores;
   }
```

```
    public Boolean PeliculaFavorita()
    {
        foreach (String actor in actores)
        {
            if (actor.Equals("Uma Thurman"))
            {
                return true;
            }
        }
        return false;
    }
}

class Program
{
    static void Main()
    {
        List<Disco> discos = new List<Disco>();
        discos.Add(new CD("Grandes Éxitos", "Grupo Casino"));
        discos.Add(new CD("Abraxas", "Carlos Santana"));
        discos.Add(new CD("BDDL1", "Buitres"));
        discos.Add(new CD("Mientras", "Buitres"));
        discos.Add(new CD("Aunque Cueste Ver El Sol", "NTVG"));
        discos.Add(new DVD("Pulp Fiction", "Uma Thurman", "John Travolta"));
        discos.Add(new DVD("Kill Bill", "Uma Thurman", "David Carradine"));
        foreach (Disco disco in discos)
        {
            ReproducirFavorito(disco);
        }
        Console.ReadKey();
    }

    static void ReproducirFavorito(Disco disco)
    {
        if (disco.GetType().Equals(typeof(CD)))
        {
            if (((CD)disco).MusicaFavorita())
            {
                Console.WriteLine("Reproduciendo {0}", disco.Nombre);
            }
        }
        else
        {
            if (((DVD)disco).PeliculaFavorita())
            {
                Console.WriteLine("Reproduciendo {0}", disco.Nombre);
            }
        }
    }
}
```

1.1 Este programa reproduce un disco si es favorito, pero tiene el problema que cuando se espera un objeto de tipo Disco pero puede aparecer un objeto de un subtipo CD o DVD. ¿Qué principio viola? Justifiquen la respuesta de forma breve y concreta.

1.2 Hagan la menor cantidad de cambios posibles para que el programa no viole ese principio. No tienen que escribir todo el código de nuevo, siempre y cuando quede bien claro qué es lo que cambian.

Sea la interfaz IComparable:

```
/// <summary>
/// Define un método de comparación generalizado que permite a los tipos
/// que la implementen crear un método de compración específico para ese
/// tipo.
/// </summary>
public interface IComparable
{
   /// <summary>
   /// Compara el receptor con otro objeto del mismo tipo.
   /// </summary>
   /// <param name="obj">Un objeto a comparar con el receptor.</param>
   /// <returns>Un entero de 32 bits con signo indicando el orden
   /// relativo de los objetos comprados. El valor retornado tiene estos
   /// significados: cuando es menor que cero, el receptor es menor que
   /// el argumento; cuando es cero, el receptor es igual que el
   /// argumento; cuando es mayor que cero, el receptor es mayor que el
   /// argumento</returns>
   int CompareTo(object obj);
}
```

Sea el siguiente programa que busca un elemento en un vector.

```
using System;

class Finder
{
   private IComparable[] data;
   public IComparable[] Data { get { return data; } }

   public Finder(IComparable[] data)
   {
      this.data = data;
   }

   public Int32 Find(IComparable item)
   {
      return Find(0, data.Length - 1, item);
   }

   public Int32 Find(Int32 from, Int32 to, IComparable item)
   {
      if (from > to)
      {
         return -1;
      }
      Int32 middle = from + (to - from) / 2;
      if (data[middle].Equals(item))
      {
         return middle;
      }
      else if (data[middle].CompareTo(item) > 0)
      {
         return Find(from, middle - 1, item);
      }
      else
      {
         return Find(middle + 1, to, item);
      }
   }
```

```
}

class Program
{
   private static String ArrayToString(IComparable[] data)
   {
      String result = "";
      for (Int32 i = 0; i <= data.Length - 1; i++)
      {
         result = result + " " + data.GetValue(i);
      }
      return result.Trim();
   }

   static void Main()
   {
      //IComparable[] data = { "a", "b", "c", "d", "e" };
      //IComparable item = "a";
      IComparable[] data = { 1, 2, 3, 4, 5 };
      IComparable item = 3;

      Finder finder = new Finder(data);
      Console.WriteLine("Buscar \"{0}\" en \"{1}\": ", item,
         ArrayToString(data));
      Console.WriteLine("Está en la posición \"{0}\"", finder.Find(item));
   }
}
```

2. La clase Finder busca un elemento en un vector -que tiene que estar ordenado en forma ascendente, dicho sea de paso, pero omitan ese detalle por el momento-. Modifiquen el programa anterior, creando una clase Finder con genéricos, sabiendo que en la declaración de un tipo genérico en C# se puede obligar a que el tipo parámetro implemente una interfaz, poniendo `class Clase<T> where T: Interfaz`. No olviden modificar también el código de la clase Program si fuera necesario.

Sea el siguiente programa:

```
using System;
using System.Collections.Generic;

public class Bag<T> where T: IComparable
{
   private T[] elements;

   public Int32 Count { get { return elements.Length; } }

   private void InternalAdd(T item)
   {
      T[] temp = new T[elements.Length + 1];

      Int32 i = 0;
      while ((i < elements.Length) && (elements[i].CompareTo(item) < 0))
      {
         temp[i] = elements[i];
         i++;
      }
      temp[i] = item;
```

```
            i++;
            while (i < temp.Length)
            {
                temp[i] = elements[i - 1];
                i++;
            }

            elements = temp;
        }

        public void Add(T item)
        {
            // ¡Your code here!
        }

        public override string ToString()
        {
            String result = "";
            foreach (T item in elements)
            {
                result = result + " " + item.ToString();
            }
            return result.Trim();
        }
    }

    class Program
    {
        static void Main()
        {
            Bag<Int32> bag = new Bag<Int32>();
            bag.Add(1);
            bag.Add(2);
            bag.Add(4);
            bag.Add(5);
            bag.Add(3);
            bag.Add(1);
            Console.WriteLine(bag);
        }
    }
```

La clase Bag es una colección que no incluye duplicados, es decir, al intentar agregar un elemento que ya existe, el elemento no se agrega.

3. Implementen el método `public void Add(T item)` usando composición y delegación con la clase Finder anterior. Noten que el método `void InternalAdd(T item)` agrega un elemento al vector `elements` en el orden correcto. Tengan en cuenta que al agregar el primer elemento `elements` es `null`.

4. C# no tiene herencia múltiple. Para resolver la pregunta anterior, ¿serviría para algo que tuviera? Justifiquen la respuesta en forma breve y concreta.

5. Agreguen a las clases Finder y Bag las precondiciones, poscondiciones e invariantes necesarias para especificar que:

   - El vector de la clase Finder tiene que estar ordenado en forma ascendente para poder buscar.
   - La clase Bag es una colección que no incluye duplicados, es decir, al intentar agregar un elemento que ya existe, el elemento no se agrega.

Conviertan esta especificación en lenguaje natural en precondiciones, poscondiciones o invariantes, según corresponda, usando Debug.Assert(Boolean compare). No es necesario que escriban las clases nuevamente, siempre y cuando indiquen claramente donde agregarían el código.

6. ¿Está bien que un objeto tenga **estado** y no **comportamiento**? ¿Y que tenga **comportamiento** y no **estado**? Consideren la clase Actor que aparece a continuación. Los actores tienen la responsabilidad de conocer su nombre ¿Hay algo mal en esta clase Actor tal como está programada? Si lo hay escriban la versión corregida si fuera necesario.

```
public class Actor
{
   private String nombre;
   public String Nombre { get { return nombre; } set { nombre = value }}
}
```

7. ¿Puede un objeto tener un tipo que tenga más operaciones que el tipo definido por la clase de ese objeto? Justifiquen la respuesta. Programen un pequeño ejemplo en C# si fuera posible.

8. ¿Puede tener más de un tipo un objeto de una clase que no implementa una interfaz? Justifiquen la respuesta. Programen un pequeño ejemplo en C# si fuera posible.

Sea el siguiente programa:

```
public class Ancestro
{
   public void Método()
   {
      Console.WriteLine("Yo soy el ancestro");
   }
}

public class Sucesor : Ancestro
{
   public void Método()
   {
      Console.WriteLine("El sucesor soy yo");
   }
}

public class Program
{
   public static void Main()
   {
      Ancestro b = new Sucesor();
      b.Método();
   }
}
```

9.1 ¿Qué aparece impreso en la consola si el lenguaje usa encadenamiento estático?

9.2 ¿Qué aparece impreso en la consola si el lenguaje usa encadenamiento dinámico?

9.3 ¿Qué cambio podrían hacer en el método Main de la clase Program para que apareciera impreso "Yo soy el ancestro" tanto si el lenguaje usara encadenamiento dinámico como estático? Escriban nuevamente la clase Program.

9.4 ¿Qué cambio podrían hacer en el método Main de la clase Program para que apareciera impreso "El sucesor soy yo" tanto si el lenguaje usara encadenamiento dinámico como estático? Escriban nuevamente la clase Program.