Programación orientada a objetos. Segundo parcial 2007.

Sábado 16 de junio de 2007

1.1 Cuando estoy programando un método en una clase y no sé si alguna subclase de esa clase necesitará sobreescribir ese método ¿lo declaro para que use encadenamiento estático o dinámico? Justifiquen la respuesta en forma breve y concreta.

1.2 ¿Cómo hago en C# para programar esa situación?

2.1 En un lenguaje orientado a objetos hipotético en el que no hay encadenamiento dinámico, es decir, solo hay encadenamiento estático ¿puede haber interfaces? Justifiquen la respuesta en forma breve y concreta.

2.2 En ese lenguaje hipotético ¿puede haber clases abstractas? Justifiquen la respuesta en forma breve y concreta.

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

class Mano
{
    public void Agarrar(Object objeto)
   {
       Console.WriteLine("La mano agarra un objeto tipo {0}",
            objeto.GetType());
    }
}

class Brazo
{
    private Mano mano;
    public Mano Mano { get { return mano; } set { mano = value; } }
}

class Robot
{
    private IList<Brazo> brazos = new List<Brazo>();
    public IList<Brazo> Brazos { get { return brazos; }
      set { brazos = value; } }
}

class Program
{
    static void Main(string[] args)
    {
        Robot robot = new Robot();
        robot.Brazos.Add(new Brazo());
        robot.Brazos[0].Mano = new Mano();

        Object objeto = new Object();
        robot.Brazos[0].Mano.Agarrar(objeto);
    }
}
```

3.1 La ley de Demeter resuelve el problema de asignar responsabilidades para evitar conocer la estructura interna de un objeto. La solución según esta ley es que un objeto sólo puede enviar mensajes en cinco situaciones. ¿Cuáles son?

3.2 ¿Qué consecuencias negativas puede tener aplicar la Ley de Demeter? Contesten en forma breve y concreta.

3.3 En el programa anterior hay varias oportunidades en las que se viola la Ley de Demeter. Identifíquenlas copiando la línea en la que ocurre la violación e indicando qué es lo que está mal.

3.4 Intentando hacer la menor cantidad de cambios posibles en el programa anterior y manteniendo las clases y su semántica, modifíquenlo para mostrar al menos un caso en el que se cumple cada una de las situaciones contempladas en la ley de Demeter. Identifiquen claramente la línea y la situación.

4. Sea el siguiente programa:

```
using System;

class Keyboard
{
    public Char Read()
    {
        // ...
    }
}

class Printer
{
    public void Write(Char c)
    {
        // ...
    }
}

class Copier
{
    private static Char Eof = Char.Parse("0x26"); // Fin de archivo

    Keyboard keyboard;
    Printer printer;
    public Copier(Keyboard keyboard, Printer printer)
    {
        this.keyboard = keyboard;
        this.printer = printer;
    }
    public void Copy()
    {
        Char c;
        c = keyboard.Read();
        while (c != Eof)
        {
            printer.Write(c);
            c = keyboard.Read();
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Keyboard keyboard = new Keyboard();
        Printer printer = new Printer();
        Copier copier = new Copier(keyboard, printer);
        copier.Copy();
    }
}
```

El programa anterior tiene el problema de que tanto la entrada como la salida son dependientes del dispositivo. Por ejemplo, la clase `Copier` no funciona para copiar al disco duro, usando una instancia de `Disk` en lugar de una instancia de `Printer`, a pesar de que tienen la misma interfaz; ni funciona leyendo de un archivo o una conexión de red, usando instancias de `File` y `Socket` respectivamente, a pesar de que también tienen la misma interfaz:

```
class Disk
{
    public void Write(Char c)
    {
        // ...
    }
}

class File
{
    public Char Read()
    {
        // ...
    }
}

class Socket
{
    public Char Read()
    {
        // ...
    }
}
```

Resuelvan el problema aplicando el principio de inversión de dependencia. Escriban todo el código nuevamente.

5. Sea el siguiente programa que ordena un vector de enteros.

```
using System;

class IntegerSorter
{
    private Int32[] array;
    public Int32[] Array { get { return array; } }

    public IntegerSorter(Int32[] array)
    {
        this.array = array;
    }
```

```
        private void Swap(Int32 i, Int32 j)
        {
            Int32 temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }

        public void Sort()
        {
            int i;
            int j;
            int temp;

            for (i = (array.Length - 1); i >= 0; i--)
            {
                for (j = 1; j <= i; j++)
                {
                    if (array[j - 1] > array[j])
                    {
                        Swap(j - 1, j);
                    }
                }
            }
        }
    }

    class Program
    {
        private static void Print(String message, Int32[] array)
        {
            Console.Write("{0}: ", message);
            for (Int32 i = 0; i <= array.Length - 1; i++)
            {
                Console.Write("{0} ", array[i]);
            }
            Console.WriteLine();
        }

        static void Main(string[] args)
        {
            Int32[] array = { 9, 7, 6, 2, 3, 4, 5, 0, 1, 8 };

            IntegerSorter sorter = new IntegerSorter(array);
            Print("Elementos a ordenar: ", sorter.Array);
            sorter.Sort();
            Print("Elementos ordenados: ", sorter.Array);
        }
    }
```

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);
    }
```

La clase `IntegerSorter` del programa anterior ordena un vector de `Int32`. Quiero tener una clase `Sorter` capaz de ordenar un vector de cualquier tipo de datos que se pueda ordenar, es decir, el operador `<=`, o algo equivalente, debe estar definido para el tipo de datos de los elementos del vector. Sabiendo que el operador `<=` no está definido para todos los tipos de datos y 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`, modifiquen el programa anterior, creando una clase `Sorter` con genéricos, que pueda ordenar cualquier un vector de cualquier tipo de datos cuyos elementos implementen la interfaz `IComparable`. No olviden modificar también el código de la clase `Program` si fuera necesario.

Importante: no deben agregar el espacio de nombre `System.Collections.Generic` a la lista de espacios de nombres utilizados.

6. La clase `IntegerSorter` del programa de la pregunta anterior ordena un vector de `Int32`. Si necesito ordenar además un vector de `String` debería crear otra clase parecida, digamos `StringSorter` con un método `public void Sort()` y repetir prácticamente todo el código de la clase `IntegerSorter`. Para evitar tener código repetido sabemos que se puede usar herencia. Implementen nuevamente la clase `IntegerSorter` y creen la clase `StringSorter` usando herencia. No pueden usar genéricos en esta oportunidad.

7. Dicen que la composición y delegación es tan efectiva para la reutilización como la herencia. Programen nuevamente las clases `IntegerSorter` y `StringSorter` del ejercicio anterior usando composición y delegación en lugar de herencia.

8. La clase `IntegerSorter` de la pregunta 5 no está especificada; por lo tanto no podemos decir si es correcta o no. Una especificación en lenguaje natural podría ser:

   - El vector a ordenar está desordenado, es decir, existe por lo menos un par de elementos en el vector donde el primer elemento del par no es menor que el segundo.
   - El vector queda ordenado luego de ordenarlo, es decir, para todo par de elementos en el vector, el primer elemento del vector es menor o igual que el segundo.

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