Programación orientada a objetos. Examen febrero 2007.

Martes 6 de febrero de 2007.

Sean las siguientes clases y programa:

```
using System;

interface ChangeItem
{
    Int32 Version { get; }
    void CheckIn();
    void CheckOut();
}

interface ConfigItem
{
    void Branch(String toBranch);
    void Merge(String fromBranch);
}

interface BuildItem
{
    String Tag { get; }
    void Build(String newTag);
}

class File : ChangeItem, ConfigItem
{
    private Int32 version;
    public Int32 Version { get { return version; } }

    public void CheckIn()
    {
        /* Código no mostrado */
    }

    public void CheckOut()
    {
        /* Código no mostrado */
    }

    public void Branch(String toBranch)/* Código no mostrado */
    {
        /* Código no mostrado */
    }

    public void Merge(String fromBranch)
    {
        /* Código no mostrado */
    }

    public Boolean CompareTo(File file)
    {
        /* Código no mostrado */
        return true;
    }
}
```

```
class Source : File
{
}

class Library : File, BuildItem
{
    private String tag;
    public String Tag { get { return tag; } }

    public Library(String initialTag)
    {
        tag = initialTag;
        /* Resto del código no mostrado */
    }

    public void Build(String newTag)
    {
        tag = newTag;
        /* Resto del código no mostrado */
    }
}

class Program
{
    static void Main(string[] args)
    {
        /*1*/ File a = new File();
        /*2*/ Library b = new Library("RC1");
        /*3*/ Library c = new Library("RC1");
        /*4*/ BuildItem d = new Library("RC2");
        /*5*/ ChangeItem e = new Source();
        /*6*/ Source f = new File();
        /*7*/ ChangeItem g = a;
        /*8*/ ConfigItem h = a;
    }
}
```

1.1 Indiquen las líneas de código incorrectas.

1.2 ¿Los objetos contenidos o referenciados en qué variables son iguales? Consideren solo los objetos definidos en las líneas correctas. Contesten completando una tabla como la siguiente:

El objeto contenido o referenciado en la variable…	Es igual al objeto contenido o referenciado en la variable…

1.3 ¿Qué variables contienen o hacen referencia al mismo objeto. Consideren solo los objetos definidos en las líneas correctas. Contesten completando una tabla como la siguiente:

El objeto contenido o referenciado en la variable…	Es el mismo objeto que el contenido o referenciado en la variable…

1.4 ¿Cuáles tipos tiene cada uno de los objetos contenidos o referenciados en las variables definidas en las líneas correctas? Consideren solo los tipos definidos en esta pregunta. Contesten completando una tabla como la siguiente:

El objeto contenido o referenciado en la variable…	Tiene el o los tipos…

1.5 ¿Qué mensajes pueden ser enviados a los objetos contenidos o referenciados en las variables a y e? ¿Cómo lo saben? Tengan en cuenta sólo los métodos para contestar esta pregunta, es decir, no consideren las propiedades. Escriban un programa en el que envían a los objetos contenidos o referenciados en esas variables todos los mensajes posibles.

1.6 ¿Los objetos contenidos o referenciados en qué variables tienen el mismo tipo? Consideren todas las variables definidas en las líneas correctas y solo los tipos definidos en esta pregunta. Contesten completando una tabla como la siguiente:

El tipo…	Lo tienen los objetos contenidos o referenciados en las variables

Sean las siguientes interfaces. Las interfaces están definidas en `System.Collections` y están copiadas aquí con la documentación del .NET Framework para vuestra comodidad. La interfaz `IEnumerable` ya la conocen.

```
/// <summary>
/// Exposes the enumerator, which supports a simple iteration over
/// a non-generic collection.
/// </summary>
interface IEnumerable
{
    /// <summary>
    /// Returns an enumerator that iterates through a collection.
    /// </summary>
    /// <returns>An IEnumerator object that can be used to iterate
    /// through the collection.</returns>
    IEnumerator GetEnumerator();
}

/// <summary>
/// Defines size, enumerators, and synchronization methods for all
/// nongeneric collections.
/// </summary>
interface ICollection : IEnumerable
{
    /// <summary>
    /// Copies the elements of the ICollection to an Array,
    /// starting at a particular Array index.
    /// </summary>
    /// <param name="array">The one-dimensional Array that is the
    /// destination of the elements copied from ICollection. The
    /// Array must have zero-based indexing. </param>
    /// <param name="index">The zero-based index in array at which
    /// copying begins.</param>
    void CopyTo(Array array, int index);

    /// <summary>
    /// Gets the number of elements contained in the ICollection.
    /// </summary>
    Int32 Count { get; }

    /// <summary>
    /// Gets a value indicating whether access to the ICollection
    /// is synchronized (thread safe).
    /// </summary>
    Boolean IsSynchronized { get; }
```

```
        /// <summary>
        /// Gets an object that can be used to synchronize access to
        /// the ICollection.
        /// </summary>
        Object SyncRoot { get; }
    }
```

Un `Stack` es una colección de objetos en la que puedo agregar elementos, pero solo puedo acceder y remover el último objeto agregado; a este último se le denomina la "boca" del `Stack`:

```
using System;
using System.Collections;

class Stack
{
    private ArrayList elements = new ArrayList();

    private Int32 limit;
    public Int32 Limit { get { return limit;  } }

    public Stack(Int32 limit)
    {
        this.limit = limit;
    }

    /// <summary>
    /// Agrega un nuevo elemento al stack.
    /// </summary>
    /// <param name="element">El nuevo elemento a agregar.</param>
    public void Push(Object element)
    {
        /* Completen el código de este método. */
    }

    /// <summary>
    /// Remueve un elemento del stack.
    /// </summary>
    /// <returns>El último elemento agregado.</returns>
    public Object Pop()
    {
        /* Completen el código de este método. */
    }

    /// <summary>
    /// Provee acceso al último elemento agregado.
    /// </summary>
    /// <returns>El último elemento agregado.</returns>
    public Object Peek()
    {
        /* Completen el código de este método. */
    }
}

class Program
{
    static void Main()
    {
        Stack b = new Stack(2);
        Console.WriteLine("Count: {0}", b.Count);
        String abc = "abc";
        b.Push(abc);
        Console.WriteLine("Push: '{0}', Count: {1}",
```

```
                b.Peek(), b.Count);
            b.Push("xyz");
            Console.WriteLine("Push: '{0}', Count: {1}",
                b.Peek(), b.Count);
            Console.WriteLine("Pop: '{0}', Count: {1}",
                b.Pop(), b.Count);
            Console.WriteLine("Pop: '{0}', Count: {1}",
                b.Pop(), b.Count);
        }
    }
```

2.1 Sabiendo que las colecciones son objetos del tipo `ICollection` hagan las modificaciones necesarias en la clase `Stack` para que las instancias de esta clase tengan el tipo `ICollection`. Tengan en cuenta que las instancias de la clase `ArrayList` también son del tipo `ICollection`.

2.2 Completen los métodos `void Push(Object)`, `Object Pop()` y `Object Peek()`.

2.3 Agreguen usando `Debug.Assert(Boolean)` lo que haga falta para especificar las siguientes afirmaciones:

- La cantidad de elementos en el `Stack` es siempre menor que el límite.
- La cantidad de elementos en el `Stack` es siempre mayor o igual que cero.
- Al agregar un elemento, la cantidad de elementos aumenta en uno.
- Al agregar un elemento, el `Stack` contiene el elemento; ese elemento queda en la "boca" del `Stack`, es decir, es accedido usando el método `Object Peek()`.
- Para remover un elemento, el `Stack` debe contener al menos un elemento.
- Al remover un elemento, el `Stack` ya no contiene el elemento en la "boca".
- Al remover un elemento, la cantidad de elementos en el `Stack` disminuye en uno.

No es necesario que escriban el programa nuevamente, mientras quede bien claro dónde harían los cambios.

2.4 Cambien, agreguen o quiten una línea del código anterior, para que se viole una precondición. Escriban la línea, si corresponde, e indiquen claramente cuál cambia o dónde la agregan o la quitan, en qué método, de qué clase.

2.5 ¿La línea de código de la pregunta anterior podría estar en la clase `Stack`? ¿Y en la clase `Program`? En cada caso indiquen "sí" o "no" y justifiquen la respuesta.

2.6 Cambien, agreguen o quiten una línea del código anterior, para que se viole una poscondición. Escriban la línea, si corresponde, e indiquen claramente cuál cambia o dónde la agregan o la quitan, en qué método, de qué clase.

2.7 ¿La línea de código de la pregunta anterior podría estar en la clase `Stack`? ¿Y en la clase `Program`? En cada caso indiquen solamente "sí" o "no".

3. Expliquen en forma clara y precisa cómo funciona el encadenamiento estático y el encadenamiento dinámico. ¿El concepto es aplicable a los lenguajes orientados a objetos en los que **no** se declara el tipo de las variables y argumentos? Muestren las diferencias entre encadenamiento estático y dinámico programando un pequeño ejemplo en C#.

4. Una clase cualquiera puede heredar de una clase abstracta o implementar una interfaz. Desde el punto de vista de los tipos ¿qué similitudes y diferencias existen en ambos casos? Programen un pequeño ejemplo en C#, usando las clases provistas anteriormente y modificándolas según sea necesario, para mostrar al menos una similitud y una diferencia.

Sean las siguientes clases:

```
public class Conductor
{
    public void Conducir()
```

```
        {
            /*…*/
        }
    }

    public class Trabajador
    {
        public void Trabajar()
        {
            /*…*/
        }
    }


    public class Programa
    {
        public static void Main()
        {
            ConductorProfesional o = new ConductorProfesional();
            o.Conducir();
            o.Trabajar();
            o.LlevarPasajeros();
        }
    }
```

5. Escriban la clase `ConductorProfesional` que falta, de forma tal que escriban lo menos posible, pero que el código todavía compile, asumiendo que el lenguaje tiene o no herencia simple o múltiple, según las siguientes opciones:

	Tiene herencia simple	Tiene herencia múltiple
5.1	Sí	Sí
5.2	Sí	No
5.3	No	Sí
5.4	No	No

   Si dos opciones tienen la misma respuesta, no escriban nuevamente la clase, sólo indíquenlo.

6. ¿Existen diferencias entre el uso de una afirmación (por ejemplo, mediante `Debug.Assert(Boolean)`) y lanzar una excepción? Si las hay, ¿cuándo usaría una y cuándo la otra?. Programe un breve ejemplo en C# que muestre ambas situaciones.

7. ¿Puede haber **encadenamiento estático** cuando el tipo de una variable está declarado en una interfaz? Justifiquen la respuesta en forma breve y concreta. Muestren la respuesta en un programa de ejemplo si fuera posible; pueden usar las clases que aparecen anteriormente.

Sean las siguientes clases:

```
public class Edit
{
    private string text;
    public string Text
    {
        get { return text; }
        set { SetText(value); }
    }

    protected virtual void SetText(string value)
    {
        text = value;
    }
}
```

```
public class Beeper
{
    public void Beep()
    {
        //...
    }
}
```

La clase `Edit` representa un control de edición de un *framework* de interfaz de usuario. La propiedad `Text` representa el texto mostrado en el control. Cuando el usuario escribe en el control el *framework* usa la propiedad `Text` para cambiar el contenido del control. También el programador puede usar la propiedad `Text` para cambiar el contenido del control por código.

La clase `Beeper` tiene la responsabilidad de emitir un sonido "beep".

8. Terminen de programar la siguiente clase de tres formas: la primera usando herencia simple y composición y delegación, la segunda usando herencia múltiple -asuman por un instante que C# soportara herencia múltiple- y la tercera usando solo composición y delegación. La clase `BeeperEdit` debe tener por lo menos los mismos métodos que la clase `Edit`. Los controles `BeeperEdit` deben emitir un sonido "beep" cuando se modifica el texto del control.

```
public class BeeperEdit
{
    // Programen la clase
}
```

9. Si todos los constructores de una clase son privados ¿puede existir en un programa al menos una instancia de esa clase? Si no se puede justifiquen brevemente el porqué. Si se puede hagan todas las modificaciones que sean necesarias en el código siguiente para mostrarlo en un pequeño programa:

```
using System;

public class ClassWithPrivateConstructor
{
    private ClassWithPrivateConstructor()
    {
    }
}

class Program
{
    static void Main(string[] args)
    {
    }
}
```