Programación orientada a objetos. Segundo parcial 2º semestre 2007.

Martes 30 de octubre de 2007.

Al igual que las demás instancias de evaluación este examen será calificado de acuerdo a la escala de letras vigente: D, R, B, MB, BMB, S o D, R, B, MB, S según la generación de cada alumno. Por eso las preguntas no tienen puntos.

Las respuestas deben aparecer en el orden en que están formuladas las preguntas.
Escriban sólo del anverso de la hoja.
Numeren las hojas.
Gracias y mucha suerte.

1.

```
class UrnaDeVotos {
    private List<Int32> votantes = new List<Int32>();
    private Dictionary<String, Int32> votos = new Dictionary<String, Int32>();
    public UrnaDeVotos(String[] candidatos) {
        foreach (String candidato in candidatos) {
            votos[candidato] = 0;
        }
    }
    public void Votar(Int32 cedula, String candidato) {
        votantes.Add(cedula);
        votos[candidato] = votos[candidato] + 1;
    }
    public Boolean YaVoto(Int32 cedula) {
        return votantes.Contains(cedula);
    }
    public Boolean EsCandidato(String candidato) {
        return votos.ContainsKey(candidato);
    }
    public Int32 CantidadVotos(String candidato) {
        if (EsCandidato(candidato)) {
            return votos[candidato];
        }
        else {
            return 0;
        }
    }
    public Int32 CantidadVotantes() {
        return votantes.Count;
    }
}
```

Y las siguientes condiciones:

a) Una persona solo puede votar una vez (no se aceptan dos votos del mismo número de cédula)
b) Una persona solo puede votar por un candidato existente.
c) Cuando se vota por un candidato, la cantidad de votos del candidato aumenta en uno.

d) Cuando se vota por un candidato, la cantidad de votantes aumenta en uno.
e) Los votos de cada candidato deben ser siempre mayor o igual que cero.

1.1. Identifica que condiciones corresponden a precondiciones, postcondiciones o invariantes.
1.2. Utilizando Debug.Assert(), agrega el código necesario asegurar que las precondiciones, postcondiciones e invariantes se cumplen.

2.

Sea la siguiente interfaz:

```
interface IUrnaVotos {
    void Abrir();
    void Cerrar();

    void Votar(Int32 cedula, String candidato);
    void Voto(Int32 cedula);
    Boolean EsCandidato(String candidato);
    Int32 Votos(String candidato);
    Int32 Votantes();
}
```

Utilizando la clase del ejercicio anterior, programa una clase que implemente la interfaz utilizando:

2.1. Composicion y delegacion
2.2. Herencia

Lo único que tienes que agregar frente a la anterior es que para votar la urna debe estar abierta.

3.

```
interface IColeccionOrdenada<T> where T:IComparable {
  void Add(T elemento);
}
abstract class Lista<T> {
   //...
   public abstract void Add(T elemento);
}

// Una lista que utiliza un array para guardar elementos de la lista
class ArrayList<T>: Lista<T> {
    //...
   public override void Add(T elemento) {
         //....
    }
}

// Una lista que utiliza una lista doblemente encadenados para guardar elementos de la lista
class LinkedList<T>: Lista<T> {
    //...
    public override void Add(T elemento) {
```

```
            //...
        }
    }
```

Te piden que proveas una lista ordenada que utilice un array para guardar elementos y otra lista ordenada que utilice una lista encadenada para guardar elementos.
3.1 Para hacerlo, ¿usarías herencia, o composición y delegación? ¿Por qué?
3.2 Explica claramente cómo realizarías tu implementación; no es necesario que la programes.

4.

```
    interface IServidor {
        void Iniciar();
        void Detener();
        void ProcesarEmail(Email mail);
        void ProcesarSMS(SMS mensaje);
    }
    class Servidor : IServidor {
        public void Iniciar() {
            //inicia el servidor, configurandolo y realizando tareas necesarias para proveer los servicios
        }
        public void Detener() {
            //detiene el servidor, liberando recursos y cerrando conexiones existentes
        }
        public void ProcesarEmail(Email mail) {
            //codigo
        }
        public void ProcesarSMS(SMS mensaje) {
            //codigo
        }
    }
    class ClienteDeEmail {
        private IServidor servidor;
        public ClienteDeEmail(IServidor servidor) {
            this.servidor = servidor;
        }
        public void EnviarMail(String destino, String texto) {
            servidor.ProcesarEmail(new Email(destino, texto));
        }
        public void Iniciar() {
            // inicia el cliente de email, mostrando una interfaz grafica, etc.
        }
        public void Detener() {
            // detiene el cliente de email, liberando recursos y cerrando la interfaz
        }
    }
    public class Email {
        public Email(String destinto, String mensaje) { }
    }
    public class SMS {
    }
```

El código anterior no cumple el principio de segregación de interfaces.
4.1. ¿Por qué? Programa el código nuevamente explicando que cambios harías para que lo cumpla.
4.2. Compara ambas versiones respecto a acomplamiento y cohesion. ¿Cuál está más acoplada? ¿Cuál es más cohesiva? ¿Cómo lo sabes?

5.
El siguiente código está escrito en un lenguaje nuevo:

```
clase Clase1:
      metodo metodo1():
            imprimir "Clase1-metodo1"
      metodo metodo2():
            imprimir "Clase1-metodo2"

clase Clase2 hereda Clase1:
      metodo metodo1():
            imprimir "Clase2-metodo1"

clase Clase3 hereda Clase2:
      metodo metodo2():
            imprimir "Clase3-metodo2"

clase Programa:
      metodo main():
            Clase2 c2 = new Clase3()
            c2.metodo1()
            c2.metodo2()
            Clase1 c1 = c2
            c1.metodo1()
```

5.1 ¿Qué se imprime si el lenguaje siempre utiliza encadenamiento dinámico?
5.2 ¿Qué se imprime si el lenguaje siempre utiliza encadenamiento estático?

6.
Sea el siguiente código:

```
class Rueda {
        public void Girar() {
            // gira!
        }
}
class Eje {
        private List<Rueda> ruedas = new List<Rueda>();
        public List<Rueda> Ruedas { get { return ruedas; } }
}
class Vehiculo {
        private List<Eje> ejes = new List<Eje>();
        public List<Eje> Ejes { get { return ejes; } }
        public void Moverse() {
            foreach (Eje eje in ejes) {
                foreach(Rueda rueda in eje.Ruedas) {
                    rueda.Girar();
                }
            }
```

```
        }
    }
    class Program {
        public static void Main(String[] args) {
            Vehiculo monociclo = new Vehiculo();
            monociclo.Ejes.Add(new Eje());
            monociclo.Ejes[0].Ruedas.Add(new Rueda());

            while (true) {
                monociclo.Moverse();
            }
        }
    }
```

El código anterior viola el principio de Demeter.
6.1. Identifica tres casos donde se viole el principio y explica por qué se viola en estos casos.
6.2. Programa lo necesario para que el principio no se viole.
6.3. Compara brevemente las dos soluciones respecto a acoplamiento. ¿Cuál está más acoplada? ¿Cómo lo sabes?

7.
El principio de inversión de dependencias (DIP), entre otras virtudes, aumenta el grado de reutilización de una clase. Explica brevemente por qué ésto sucede mediante un ejemplo donde apliques el DIP.

8.

```
    class Usuario {
        private String nombre;
        public String Nombre { get { return nombre; } }
        public Usuario(String nombre) { this.nombre = nombre; }
    }
    class EliminadorDuplicados {
        private List<Usuario> usuarios;
        public EliminadorDuplicados(List<Usuario> usuarios) {
            this.usuarios = usuarios;
        }
        public void EliminarDuplicados() {
            for (int i = 0; i < usuarios.Count; i++) {
                int j = i + 1;
                while (j < usuarios.Count) {
                    if (usuarios[i].Nombre.Equals(usuarios[j].Nombre)) {
                        usuarios.RemoveAt(j);
                    } else {
                        j = j + 1;
                    }
                }
            }
        }
    }
```

Utilizando herencia, programa una clase que permita eliminar duplicados de una lista de cadenas (String).