Programación Orientada a Objetos. Segundo Parcial 1er semestre 2009.
Viernes 12 de junio de 2009

1. Sea el siguiente código:

```
class LigaUniversitaria {
    private Hashtable cuadros = new Hashtable();
    private Hashtable jugadores = new Hashtable();
    /* Registra un cuadro */
    public void Registrar(Cuadro cuadro) { }
    /* Desregistra un cuadro */
    public void Borrar(String nombre) { }
}
class Cuadro {
    private String nombre;
    private String serie;
    private IList jugadores = new ArrayList();
    /* Agrega un jugador al Cuadro */
    public void Agregar(Jugador jugador) { }
    /* Saca a un jugador del Cuadro */
    public void Borrar(String nombre) { }
    public void Jugar() { }
}
class Jugador {
    private String nombre;
    private Cuadro cuadro;
}
```

Y las siguientes condiciones:

a) No se puede registrar un cuadro con un nombre ya registrado
b) Cuando se borra un jugador, éste no pertenece más al cuadro
c) Cuando se registra un cuadro, el cuadro ingresa en la serie H
d) Un cuadro debe tener 11 o más jugadores antes de poder registrarse.
e) Un cuadro no puede jugar con menos de 11 jugadores.
f) Un jugador no puede estar registrado en más de un cuadro

1.1. Identifica qué condiciones corresponden a precondiciones, postcondiciones o invariantes.
1.2. Programa nuevamente las clases para que cumplan estas condiciones, modificándolas pero respetando su estructura actual.
1.3. Utilizando Debug.Assert(), agrega el código necesario para verificar esas precondiciones, postcondiciones e invariantes. Puedes hacer esta parte junto con la 1.2.
1.4 ¿Quién es el culpable de que se viole una precondición? ¿Y una postcondición?

2. Sea el siguiente código:

```
public class Clinica {
    private Double montoConsulta;
    public Double MontoConsulta {
         get { return montoConsulta;}
          set { montoConsulta = value;}
     }
}
public class Hospital {
    private IList pacientes = new ArrayList();
    public IList Pacientes {
         get { return pacientes;}
         set {pacientes = value;}
    }
}
public class Paciente{
    private Clinica clinica;
    public Clinica Clinica { get{ return clinica; } }
    private String ci;
    public String Ci { get { return ci; } }
}
public class Consulta {
    private Hospital h = new Hospital();
    public void CalcularImporte(String ci) {
         foreach (Paciente p in h.Pacientes) {
             if (p.Ci.Equals(ci)) {
                 Clinica cli = p.Clinica;
                 Double importe = cli.MontoConsulta;
                if (importe > 0) {
                    Imprimir(p, importe);
                }
            }
        }
    }
    private void Imprimir(Paciente p, Double total) {
         /* complejo codigo para imprimir un ticket */
    }
}
```

2.1 Explica qué es el acoplamiento y critica el código void CalcularImporte(String ci) en base a su acoplamiento.
2.2 Explica **claramente** que cambios realizarías para mejorar el código en base a las críticas hechas en el punto anterior. No tienes porque volver a programar el código.

3. Sea el siguiente código:

```
public class Base {
     public void Print() {Console.WriteLine("Base");}
}
public class Deriv: Base {
     public void Print() {Console.WriteLine("Deriv");}
}
Base obj;           /1/
obj = new Base();   /2/
obj.Print();        /3/
obj = new Deriv();  /4/
obj.Print()         /5/
```

3.1 Indica cuál es la salida de las líneas 3 y 5.
3.2 ¿En base a qué se está determinando el método a ejecutar en este caso?
3.3 Si usaras un tipo de encadenamiento distinto, ¿cuál sería la salida? ¿cómo se determinaría el método a ejecutar?.

4.

```
class Raton {
   private Int32 x;
   public Int32 X { get {return x; } }
   private Int32 y;
   public Int32 Y { get { return y; } }
}
class Menu {
   private Raton r;
   public void ProcAccion() {
      if (r.X > 100 && r.X < 200 && r.Y < 400) {
         ProcAccion1();
      } else if (r.X > 200 && r.Y > 400) {
         ProcAccion2();
      }
   }
   private void ProcAccion1() { /* implementacion */}
   private void ProcAccion2() { /* implementacion */}
}
```

El código anterior viola el DIP. Explica por qué y prográmalo nuevamente para que lo cumpla.

5.

```
class Persona {/* complejo codigo que pasa el test de Turing */}
interface IRol { void Actuar(); }
class Estudiante: IRol {
   public void Actuar() { /* hace lo propio de un estudiante */ }
}
class Profesor: IRol {
   public void Actuar() { /* hace lo propio de un profesor */ }
}
```

5.1 Modela en tu programa personas que son estudiantes o profesores.

Discute si utilizarías herencia o composición y delegación y propone una implementación modificando tanto como necesites las clases anteriores. Puedes escribir la implementación o describir la idea general.

Asume que puedes utilizar herencia múltiple si así lo necesitas y que en el futuro pueden aparecer otros requerimientos (estudiantes que son también profesores, o nuevos roles).

6.

```
class Autorizador {
    /* nombre usuario -> lista de operaciones permitidas */
    private Hashtable usuarios = new Hashtable();
    public Boolean Autorizar(String usuario, String operacion) {
       return ((IList)usuarios[usuario]).Contains(operacion);
    }
 }
```

Tu jefe te pide que desarrolles una solución para que sea posible mostrar un mensaje por consola cuando un usuario es autorizado para realizar una operación. Horas más tarde te pide que que envíes éstos datos por email. Horas más tarde que lo guardes en disco. Al otro día, que hagas la tres juntas.

6.1 Programa una solución para atender los requerimientos de tu jefe sin tener que modificar constantemente Autorizador. Puedes modificar Autorizador tanto como necesites, pero no deberías tener que modificarla ante a cada nuevo requerimiento.