Programación orientada a objetos. Segundo parcial 2º semestre 2009.
Jueves 29 de octubre de 2009.

1. Sea el siguiente código:

```
class Certificado {     }
class Votante {
    public void Votar(Lista l, Circuito c) {/**/}
    public void PagarMulta() {/**/}
}
class Lista {
    private String presidente;
}
class Voto {
    private Votante votante;
    private Lista lista;
}
class Circuito {
    private IList votantes = new ArrayList();
    private IList votos = new ArrayList();
    void Cerrar() { }
    void Habilitar(Votante v) {/**/}
    Certificado RegistrarVoto(Votante v, Lista l) {
          /* registra el voto */
    }
    String ContabilizarVotos() {
          /* devuelve el presidente con mas votos */
    }
}
```

Y las siguientes condiciones:
a) Un circuito no puede registrar más votos que la cantidad de votantes.
b) Un votante puede votar una única vez.
c) Al votar, el votante recibe un certificado de voto.
d) El circuito no puede contabilizar los votos hasta no haber cerrado.
e) Una lista siempre tiene un presidente.
f) Un votante solo puede votar si está habilitado en el circuito.
g) Un votante que no votó, puede pagar la multa y al hacerlo recibe un certificado de voto.

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 ¿Qué diferencia existe entre el uso de estas condiciones y el uso de excepciones? ¿Cuándo usarías una y cuándo otras?

2.
2.1 Una clase cualquiera puede heredar de una clase abstracta, de una clase concreta o implementar una interfaz. Programa un ejemplo en C# donde te sería más útil heredar de una clases abstracta que heredar de una concreta o implementar una interfaz. Explica el por qué. Puedes no programar el cuerpo de los métodos o constructores ni declarar variables de instancia, pero el resto del código debe ser correcto.

2.2 Provee un ejemplo de código en donde es más adecuado usar composición y delegación que herencia. Explica por qué consideras que ésto es así.

2.3 Vuelve a programar el ejemplo anterior utilizando herencia, no puedes utilizar composición y delegación para resolver este punto. Si necesitas hacer uso de herencia múltiple, hazlo.

3.
3.1 Explica brevemente que es el acoplamiento. ¿Por qué es indeseable el alto acoplamiento?.
3.2 El siguiente programa presenta un acoplamiento indeseable. El generador de PDF es todo un sistema en si mismo, altamente complejo, pero si quiero reutilizar en otro programa el Sistema de Gestión, debo compilar mi sistema contra todas las clases del Sistema más las clases del Generador de PDF. Propone una solución que me permita separar el `GenDePdf` del `SistemaDeGestion`, permitiendome usar el segundo sin necesidad de incluir el primero.

```
interface IConstructorReporte {
    Reporte Construir();
}
class Reporte { }
class SistemaDeGestion {
    private Hashtable constructores = new Hashtable();
    void ImprimirAPdf(String nombre, GenDePdf gen) {
        Reporte r = constructores[nombre].Construir();
        gen.Generar(r);
    }
}
class GenDePdf {
    public void Generar(Reporte r) {
        /* complejo codigo */
    }
    /* Cantidades industriales de líneas de código,
       violando todos los principios conocidos */
}
/* Muchas clases más de las que depende el Generador de PDF. */
```

4. Dado el siguiente código:

```
interface Interfaz {
    void M1();
}
abstract class ClaseAbstracta : Interfaz {
    public void M1() { }
    public abstract Boolean M2();
}
class ClaseConcreta : ClaseAbstracta {
    public override Boolean M2() { return false; }
    public virtual void M3(Int32 i) { }
    public void M4() { }
}
```

4.1. ¿En base a que se determina el método a ejecutar al usar encadenamiento estático? ¿Y al usar dinámico?
4.2. Dadas la siguientes asignaciones:

```
ClaseConcreta concreta = new ClaseConcreta();
ClaseAbstracta abst = new ClaseConcreta();
Interfaz inter = abst;
ClaseConcreta concretaCast = (ClaseConcreta) abst;
```

Indica el tipo de encadenamiento usado en cada caso:

```
a) inter.M1();
b) abst.M1();
c) abst.M2();
d) concretaCast.M2();
e) concreta.M4();
f) concretaCast.M3(10);
```

5. El siguiente código representa un programa peridístico del '60 (o eso suponemos):

```
class Adivino {
    Boolean SeraPresidente(String nombre) {
      /* tirada de buzios, lectura de hígado de pollo y hojas de té */
    }
    void Comer(){ /* los adivinos también comen */ }
    void Dormir(){ /* los adivinos también duermen */ }
    void Meditar(){ /* Trance profundo */ }
}
class ProgramaPeriodistico {
    private Adivino adivino;
    void ContratarAdivino(Adivino adivino) { this.adivino = adivino; }
    void TransmitirElecciones() {
        while (True) {
            if (adivino.SeraPresidente("Juan")) {
               Transmitir("Juan! El pueblo ha decidido!");
               return;
            } else if (adivino.SeraPresidente("Raul")) {
               Transmitir("Raul! El pueblo ha decidido!");
               return;
            } else {
               adivino.Meditar();
               Transmitir("En seguida volvemos con más información!");
            }
        }
    }
    private void Transmitir(String mensaje) {
        /* transmite el mensaje al mundo */
    }
}
```

Por suerte, ahora los programas periodísticos profesionalizaron su trabajo, y en lugar de usar Adivinos usan Consultoras, las cuales aplican un proceso estádistico, formal, y dentro de la medida, objetivo, para tratar de proyectar resultados.
Critica el código respecto a:
1.ISP
2.DIP

6. Programa nuevamente el código, realizando los cambios necesarios para que se cumplan el ISP y el DIP.

7. Sea el siguiente código:

```
    interface IOperacionBancaria { void Realizar(); }
    class Retiro : IOperacionBancaria {
        public void Realizar() { /* hace un retiro */ }
    }
    class Deposito : IOperacionBancaria {
        public void Realizar() { /* hace un deposito */ }
    }
    interface ICondicion {    Boolean CumpleCondicion(); }
    interface IAlarma {       void EmitirAlarma(); }
```

Te piden que programes una operacion bancaria compleja y segura. Una operacion bancaria compleja y segura, es una operación compuesta de varias operaciones individuales, que antes de realizar estas operaciones, evalua si un conjunto de condiciones se cumplen. Si todas las condiciones se cumplen, realiza las operaciones individuales, en otro caso, emite un conjunto de alarmas.
Por supuesto, la operación programada debe ser lo más reutilizable que puedas.