***Programación orientada a objetos. Examen diciembre 2009.***

*Martes 15 diciembre de 2009.*

1. Sea el siguiente código:

```
interface ISimbolismo {
    void Representar();
}
interface IOrgano {
    void Funcionar();
}
abstract class Cruz : ISimbolismo {
    /* implementacion completa y correcta */
    public Int16 LargoTramos { get { /* .... */ } }
    abstract public void AsignarAnguloEntreTramos(Int16 angulo);
}
interface ICorazon: IOrgano, ISimbolismo {
    void Amar();
    void Odiar();
}
class CruzRoja: Cruz {
    /* implementacion completa y correcta */
}
interface IGatoNegro : ISimbolismo {
    void Maullar();
}
class Alma : ICorazon, IGatoNegro {
    /* implementacion completa y correcta */
    void Volar() { /* .... */ }
}
class Program {
    void Main() {
        /*1*/ IOrgano a = new Alma();
        /*2*/ IGatoNegro g = a;
        /*3*/ ISimbolismo s = new Cruz();
        /*4*/ s = a;
        /*5*/ Cruz cr = new CruzRoja();
        /*6*/ CruzRoja c = s;
        /*7*/ s = cr;
        /*8*/ ICorazon h = new ICorazon();
    }
}
```

1.1 Indica las líneas de código incorrectas. Si una asignación es incorrecta, de todas formas asume que la variable fue definida.
1.2 Indica los tipos de las variables **c y h,** y el tipo del objeto creados en la **línea 5**.
1.3 ¿Qué mensajes puedes enviar a un objeto referenciado por una variable **definida como en la linea 1**? ¿Cómo lo sabes?

2. Sea el siguiente código:

```
class Persona { }
class Piso { }
class Elevador {
    IList<Persona> personas = new List<Persona>();
    IList<Persona> Personas { get { return personas; } }
    void Abrir() { }
    void Cerrar() { }
    void Subir(Persona p) { }
    void PararEn(Piso piso) { }
}
```

Y las siguientes condiciones:

a) Solo puede subir una persona al elevador cuando la puerta está abierta.
b) Cuando el elevador para en un piso, bajan todas las personas que iban a ese piso.
c) Cuando el elevador para en un piso, se abre automáticamente la puerta.
d) Cuando se baja a una persona del elevador, esta queda en el piso donde el elevador está parado.
e) Una persona puede subir al elevador solo si está en el piso donde el elevador está parado.
f) Una persona no puede estar al mismo tiempo en un piso y en un elevador.
g) Solo se puede parar en un piso si la puerta estaba cerrada.

2.1. Identifica qué condiciones corresponden a precondiciones, postcondiciones o invariantes.
2.2. Programa nuevamente las clases para que cumplan estas condiciones, modificándolas pero respetando su estructura actual. Recuerda que lo que no está pedido explícitamente en las condiciones puedes implementarlo como gustes.
2.3. Utilizando Debug.Assert(), agrega el código necesario para verificar esas precondiciones, postcondiciones e invariantes. Puedes hacer esta parte junto con la 2.2.

3.
3.1. Se dice que al usar herencia usamos "reutilización estática", mientras que al usar composición y delegación usamos "reutilización dinámica". Explica claramente por qué se dice ésto. Puedes ayudarte de un ejemplo.

3.2. Sea el siguiente código:

```
interface ITanque {
    void Llenar(int litros);
}

/* un horno industrial */
class Horno {
    public virtual void GetTanque() {
      return new TanqueHorno(this);
    }
}

/* un tanque de combustible para hornos */
class TanqueHorno: ITanque {
    private readonly Horno horno;
    TanqueHorno(Horno horno) {
        this.horno = horno;
    }

    public virtual void Llenar(int litros) {
        /* complejo y peligroso proceso de llenado que
             utiliza, para alguna cosa rara, al horno */
    }
}
```

Los clientes reportan que los hornos se les desbordan porque los tanques no cortan la toma de combustible cuando superan un cierto límite de litros.

Te piden que **programes** un **Horno nuevo** cuyo Tanque no acepte más combustible una vez alcanzado ese límite. **No puedes cambiar el código provisto**, debes reutilizarlo de alguna forma. **Pista:** *La mejor solución utiliza herencia y composición y delegación.*

4.
Hay Aviones, Barcos y Helicópteros. También hay Aeropuertos, Puertos y Helipuertos. Pero para tu programa todos estos puertos son muy parecidos: solo permiten Estacionar Aviones, Barcos o Helicópteros, respectivamente. Si Estacionar se implementa **agregando el vehículo a una lista**, provee un **programa** que modele este problema, usando el mecanismo de reutilización de código más apropiado.

5.
El siguiente código, escrito por los antiguos Griegos para un ataque a Troya, ha sido rescatado por arqueólogos:

```
class Griego {
  void Filosofar() { /* solo se que no se nada */ };
  void Atacar()  { /* ¡Esto es Esparta! */ }
}

class Caballo {

  IList<Griego> griegos = new List<Griego>();

  void AgregarGriego(Griego g) { griegos.Add(g); }

  void LLegarA(Troya troya) {
      while (!LlegoLaNoche()) { /* do nop */ }
      Console.WriteLine("¡Sorpresa!");
      foreach(Griego g in griegos) { g.Atacar(); }
  }

  protected virtual bool LlegoLaNoche(){/*el reloj solar da sombra?*/}
}
```

Las grandes potencias quieren adecuarlo para respetar principios modernos de POO y poder usarlo en escenarios distintos (sin griegos, de día, etc.).

5.1 **Indica** que principios de ISP, DIP, Experto, LSP, OCP **se violan** y **justifica por qué** crees que esto es así. Si el principio no se viola no tienes por que justificar nada.

6. **Programa** nuevamente el código anterior para cumplir los principios violados.

7. El siguiente programa monitorea los valores de acciones que cotizan en la bolsa:

```
class Accion {
    String nombre;
    String Nombre { get { return nombre; } }
    String valor;
    String Valor { get { return valor; } }
}

class Monitor {
    void Monitorear() {
        while (True) {
            Accion accion = SiguienteAccion();
            Console.WriteLine("La accion {0} vale {1}",
                accion.nombre, accion.valor);
        }
    }
    private Accion SiguienteAccion() {
        /* complejo código para obtener el siguiente
           valor disponible */
    }
}
```

Te piden que modifiques el programa para que en el futuro puedas enviar un email, guardar un registro en disco, y mandar un SMS, cuando el valor de una acción esté por arriba o por abajo de un límite, tenga un crecimiento exponencial, o decaiga cuadráticamente. Lo más probable es que en el futuro tengas que monitorear otras cosas o tomar otras acciones.

Programa nuevamente la clase, agregando lo que necesites (otras clases, interfaces, etc.), para que puedas reutilizar Monitor para hacer estas cosas.