Programación orientada a objetos. Examen diciembre 2008.

Martes 16 de diciembre de 2008.

Sean las siguientes clases e interfaces:

```
interface Comida { }
interface VehiculoAereo {
   void Volar();
}
abstract class Animal: Comida {
   /* implementación completa */
   private Comida almuerzo;
   void Comer(Comida almuerzo) {
      this.almuerzo = almuerzo;
   }
   Comida UltimoAlmuerzo { get { return almuerzo; } }
}
class Reno : Animal, VehiculoAereo {
   /* implementación completa */
}
class Trineo: VehiculoAereo {
   /* implementación completa */
   List<Reno> Renos {  get {  /* implementación completa */ } }
}
class Humano : Animal {
   /* implementación completa */
   void Pensar() { }
}
class Program {
   static void Main(string[] args) {
      /*1*/ Comida rudolph = new Reno("Rudolph");
      /*2*/ Reno blitzen = new Animal();
      /*3*/ VehiculoAereo comet = new Reno("Comet");
      /*4*/ Reno dazer = comet;
      /*5*/ Animal papaNoel = blitzen;
      /*6*/ Humano papaNoel2 = papaNoel;
      /*7*/ rudolph = new Humano();
   }
}
```

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 la variable **dazer** y los tipos del objeto creado en la línea **7**.

1.3 Sea el siguiente código:

```
Reno rudolph = new Reno("Rudolph");
Animal papaNoel = new Humano();
papaNoel = rudolph;
rudolph.Comer(papaNoel);
papaNoel.Comer(rudolph);
```

¿El código es correcto? En caso negativo, justifica. En caso positivo, ¿Cuál es la clase del objeto retornado por **papaNoel.UltimoAlmuerzo** al final del código? ¿Cuál es la clase del objeto retornado por **rudolph.UltimoAlmuerzo** al final del código?.

El siguiente código se utiliza para entregar regalos el día de reyes.

```
class ReyMago: Humano {}
class Camello {
   /* completar */
   ReyMago Jinete { set { /* completar */ } }
   void CargarRegalo(Regalo r) { /* completar */ }
   Regalo EntregarRegalo() { /* completar */ }
   void Comer(Comida c) { /* completar */ }
}
class Regalo {
   /* completar */
}
```

La clase **Camello** debe cumplir las siguientes condiciones:

a) No puede entregar dos regalos seguidos, sin haber comido en el medio.
b) No puede comer dos veces seguidas, sin haber entregado un regalo en el medio.
c) El camello no puede entregar un regalo sin un jinete.
d) Luego de entregar un regalo, el camello no lleva más ese regalo.
e) Luego de cargar un camello con un regalo, el camello lleva ese regalo.
f) No se puede entregar un regalo si el camello no tiene más regalos disponibles.

2.1 Indica que condiciones corresponden a precondiciones, postcondiciones e invariantes.
2.2 Modifica y programa correctamente la clase **Camello** para cumplir todas las condiciones mencionadas. Declara las precondiciones, postcondiciones e invariantes usando **Debug.Assert(bool Condition)**.

Sea el siguiente código

```
class AccesoNoAutorizado : Exception { }
class ClienteEmail {
   void Enviar(String destino, String mensaje) { /* codigo difícil */ }
   List<String> ChequearCorreo() { /* codigo aun más difícil */ }
   /* ... otras operaciones adicionales requeridas para gestión de un cliente de correo */
}
class MonitorSeguridad {
   private ClienteEmail cliente = new ClienteEmail();
   private PoliticaSeguridad politica = new PoliticaSeguridad();
   PoliticaSeguridad Politica { get { return politica; } }
   void AutorizarOp(String usuario, String operacion) {
      if ( ! politica.OpsAutorizadas[usuario].Contains(operacion)) {
         cliente.Enviar("admin@server", "ALERTA!");
         throw new AccesoNoAutorizado();
      }
   }
}
class PoliticaSeguridad {
   public IDictionary<String, List<String>> OpsAutorizadas = new Dictionary<String,List<String>>();
}
```

*Para quienes cursaron en el 2006 o anterior (**INDICAR EN ESTE CASO**):*

3.1 Considerando que las formas de notificar alertas pueden cambiar, critica el código en base a criterios de reusabilidad, mantenimiento y distribución del código.

Para quienes cursaron en el 2007 o posterior:

Critica el código en base a:
3.1 Patrón Experto
3.2 DIP
3.3 ISP

4.1 Corrije el código del ejercicio 3 en base a las críticas realizadas.

5.1 Compara el uso de herencia frente a composición delegación como mecanismo de reutilización de código.

5.2 Provee un ejemplo donde es más adecuado utilizar composición y delegación que herencia, y justifica por qué esto es así. **Basa tu justificación en la comparación realizada en el punto 5.1** (en otras palabras, una justificación basada únicamente en decir que no *es-un* no es aceptable).

Sea el siguiente código:

```
class Pelota { }
class Zapato {
   void Patear(Pelota pelota) {
   }
}
class Pie {
   Zapato zapato = new Zapato();
}
class Jugador {
   Pie derecho = new Pie();
   Pie izquierdo = new Pie();
}
class Progam {
   static void Main(string[] args) {
      Jugador j = new Jugador();
      Pelota p = new Pelota();
      /* dominar la pelota */
      for (Int32 i = 0; i < 10000; i++) {
         if (i % 2 == 0) {
            j.derecho.zapato.patear(p);
         } else {
            j.izquierdo.zapato.patear(p);
         }
      }
   }
}
```

6.1 ¿Con qué clases está acoplada la clase Progam?
6.2 Reduce el acoplamiento de la clase Progam programando los cambios necesarios.

Sea el siguiente código

```
interface Ingrediente {
}
class Ensalada {
   /* completar */
   IList<Ingrediente> Ingredientes { get { ingredientes; } }
}
```

7 Una ensalada de lechuga y tomate puede combinarse con una de salamín y queso para crear una ensalada de lechuga, tomate, salamín y queso.

Utilizando las clases anteriores, programa un código que permita representar ensaladas como combinaciones de otras ensaladas, de una ensalada y otros ingredientes, de varias ensaladas y un ingrediente, etc. Puedes modificar todo lo que necesites. Provee métodos en ensalada que permitan crear fácilmente estas combinaciones.