Programación orientada a objetos. Examen julio 2010.

04 de marzo de 2011.

1. Sean las siguientes clases e interfaces:

```
interface ITransporte { }
interface INautico : ITransporte { }
interface ITerrestre : ITransporte { }
abstract class Barco : INautico { }
class MotoAgua : INautico { }
class Lancha : Barco { }
class Crucero : Barco { }
abstract class Automovil: ITerrestre { }
class Moto : ITerrestre { }
class Auto : Automovil { }
class Camion : Automovil { }

class Programa {
    void Principal()
    {
        /*1*/ ITransporte ma = new MotoAgua();
        /*2*/ Moto m = ma;
        /*3*/ Barco c = new Crucero();
        /*4*/ ITerrestre t = new Automovil();
        /*5*/ INautico n = c;
        /*6*/ Automovil a = new Auto();
        /*7*/ Camion ca = a;
    }
}
```

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 definidas en las líneas **1 y 3**, y los tipos de los objetos creados en las líneas **3 y 6**.

1.3. Realiza los cambios necesarios en el código anterior para que:

a- un **objeto** como el creado en la **línea 1** pueda recibir los siguientes mensajes: aumentarVelocidad(), frenar()

b- la **variable** declarada en la **línea 7** pueda recibir los siguientes mensajes: tieneZorra(), reducirVelocidad()

2. El siguiente código se utiliza para un sistema de gestión de proyectos:

```
class Proyecto
{
    /* completar */
    List<Etapa> Etapas { get; set; }
    void AgregarEtapa(Etapa e) {/* completar */}
    void BorrarEtapa(Etapa e) {/* completar */}
    Boolean ProyectoFinalizado() {/* completar */}
}
class Etapa
{
    /* completar */
    List<Entregable> Entregables { get; set; }
    void AgregarEntregable(Entregable e) {/* completar */}
    Boolean EtapaAtrasada() {/* completar */}
}
class Entregable
{ }
```

El código debe cumplir con las siguientes condiciones:
a) No se pueden agregar etapas a los proyectos que han finalizado.
b) Un proyecto por definición, siempre tiene una fecha de fin definida.
c) No se pueden agregar entregables a etapas que tienen atraso.
d) Una etapa debe tener por lo menos un entregable.
e) Luego que se agrega un entregable, la cantidad de entregables de la etapa aumenta en uno.
f) No se pueden borrar etapas que ya han sido cumplidas.
g) Luego que se borra una etapa, la duración del proyecto se reduce (utiliza cualquier manera para representar la reducción)

2.1. Indica qué condiciones corresponden a precondiciones, postcondiciones e invariantes.
2.2. Modifica el código provisto para cumplir todas las condiciones mencionadas. Declara las precondiciones, postcondiciones e invariantes usando Debug.Assert(bool Condition). Debes completar en los lugares indicados y agregar cosas adicionales si lo necesitas.

3. Expone un ejemplo claro y completo, donde sea más ventajoso utilizar composición y delegación frente a utilizar herencia. No es necesario que se programe la solución siempre que se ejemplifiquen claramente las ventajas.

4. Analiza y responde las siguientes preguntas:
4.1. Programa un código análogo al siguiente (funcional y conceptualmente), sin utilizar interfaces.

```
interface IFruto {
      double Calorias();
}
class Manzana: IFruto {
      private double peso;
      public double Peso { get; }
      public double Calorias(){
            return peso*0,3
      }
}
```

4.2. Dadas las siguientes clases (que representan una naranja), explica las diferencias que implican las dos definiciones dadas.

```
class Naranja {
      double Calorias {}
      virtual void Madurar(){}
      void Caer(){}
}
```

```
abstract class Naranja {
      abstract double Calorias();
      virtual void Madurar(){}
      void Caer(){}
}
```

4.3. Si tuvieras que optar entre un lenguaje que solo posee encadenamiento estático y otro que solo posee encadenamiento dinámico para construir un programa orientado a objetos. ¿Cuál elegirías? ¿Por qué?

5. Sea el siguiente código

```
class Libro {}
class Biblioteca {
    List<Libro> Libros { get; set; }
    void OrdenarAscendente() {
        // Ordena los libros ascendentemente
    }
    void OrdenarRandomicamente() {
        // Ordena los libros randómicamente
    }
}
```

5.1 Critica el código en base al OCP.
5.2 Critica el código en base al DIP
5.3 Critica el código en base al SRP

6. Re-programa el código provisto en el ejercicio 5 para que cumpla con todos los principios evaluados en el mismo (OCP, DIP y SRP).

7. Sea el siguiente código:

```
class Porton {
    public void Abrir() { /* Abre el portón */ }
    public void Cerrar() { /* Cierra el portón */ }
}

class Pestillo
{
    private Porton p;
    private Boolean abierta;
    public Pestillo(Porton p)
    {
        this.p = p;
        this.abierta = false;
    }
    public void Girar() {
        /* Al girar el pestillo, si la puerta está abierta, la
        cierra, y si está cerrada la abre*/
        bool abrir = !(this.abierta); //Lógica actual de apertura
              if (abrir)
                 this.Abrir();
              else
                 this.Cerrar();
    }
    private void Abrir()
    {
        abierta = true;
        p.Abrir();
    }
    private void Cerrar()
    {
        abierta = false;
        p.Cerrar();
    }
}
```

7.1. Se quiere modificar este código para que el pestillo pueda abrir cualquier tipo de puerta (no solo portones). Realiza este cambio e indica qué principio, de los vistos en clase, estás aplicando.

7.2. En el código anterior, la manera de definir si la puerta se puede abrir o cerrar, depende de si la puerta esta cerrada o abierta. Ésta no será siempre la forma de realizarse la acción, por ejemplo, algunos pestillos para poder abrir la puerta deben tener desactivada la seguridad, o para cerrar la puerta deben tener una señal de cierre habilitada; existen otras combinaciones más. Desarrolla entonces, una solución donde cada pestillo pueda definir sus condiciones para abrirse o cerrarse, re-utilizando en la mayor manera posible la estructura general del código provisto por el pestillo original.