# *Programación orientada a objetos. Examen julio 2008.*

*Lunes 14 de julio de 2008.*

1.1 Un objeto puede tener más de un tipo y objetos de distintas clases pueden tener tipos en común. Escriban un ejemplo breve de código C# donde aparezca cada uno de los siguientes casos:

- Un objeto con más de un tipo. Indica los tipos del objeto.
- Dos objetos de distinta clase con tipos en común, declarando los tipos mediante interfaces.
- Dos objetos de distinta clase con tipos en común, declarando los tipos mediante clases.

1.2 Sean las siguientes clases e interfaces:

```
interface Reproducible {
   void Reproducir();
}
abstract class Disco : Reproducible {
   public String GetTitulo() { /* implementacion */ }
   public abstract Int32 GetDiametro();
   public abstract void Reproducir();
}
class Cd : Disco {
   public override void Reproducir() { /* implementacion */ }
   public IList GetCanciones() { /* implementacion */ }
   public void PasarASiguienteCancion() { /* implementacion */ }
   public override Int32 GetDiametro() { return 42; }
}
class Cassette : Reproducible {
   public IList GetCanciones() { /* implementacion */ }
   public void Reproducir() { /* implementacion */ }
}
```

¿Es posible definir una variable que de tipo Reproducible que referencie a un objeto que implemente el método void PasarASiguienteCancion()? Justifica. En caso negativo, ¿qué cambio podrías realizar para poder hacerlo? En caso afirmativo, provee un ejemplo donde ésto suceda.

1.3 ¿Qué mensajes puedes enviar al objeto referenciado por la variable **cd** desde el siguiente fragmento de código? ¿Cómo lo sabes?

```
Disco cd = new Cd()
```

Sea el siguiente código,

```
class LineaFactura {
   private readonly String item;
   private readonly Double valor;
    /* COMPLETAR código */
}
class Factura {
   private readonly IList lineas = new ArrayList();
   public virtual Boolean Contains(LineaFactura linea) { /* COMPLETAR código */ }
   public virtual void AgregarLinea(LineaFactura linea) { /* COMPLETAR código */ }
   public virtual Double Total() { /* COMPLETAR código */ }
}
```

2.1 Completa el código faltante para asegurarte que Double Total() retorna la suma de los valores de cada linea de factura agregada.

2.2 Para cada una de las siguientes condiciones, indica si es una precondición, postcondición o invariante, y agrégala en el código programado en la parte 2.1 usando Debug.Assert.

a) El ítem no debe ser nunca nulo y el valor siempre mayor igual que cero.
b) No se puede agregar más de una vez la misma línea a la factura.
c) Al agregar la linea, la factura contiene la linea, y el total aumenta tanto como el valor de la línea de factura.
d) El total corresponde siempre a la suma de todos los valores de todas las lineas de la factura.

*Puedes responder la pregunta 2.1 y 2.2 juntas si te resulta más cómodo.*

2.3 ¿Existen diferencias entre usar una afirmación (por ejemplo, mediante Debug.Assert(Boolean)) y lanzar una excepción? Si las hay, ¿cuándo usarías una y cuándo la otra?. Esquematiza un breve ejemplo en C# que muestre ambas situaciones.

3.1 ¿En base a qué se determina el método a encadenar cuando se usa encadenamiento dinámico? ¿Y cuándo se usa estático?
3.2 En los siguientes casos, indica qué tipo de encadenamiento -estático o dinámico- se utiliza y qué método se ejecuta al correr el programa.

```
/*1*/Reproducible a = new Cd();
/*2*/Disco b = new Cd();
/*3*/Cd c = new Cd();
/*4*/a.Reproducir();
/*5*/b.GetTitulo();
/*6*/c.GetDiametro();
```

Sean las siguientes clases:

```
public class CompactDisc {
   public void PlayAll() { /* implementación de largo absurdo */ }
}
public class VideoDisc {
   public void PlayMovie() { /* código de ilegible complejidad */ }
}
```

Los métodos CompactDisc.PlayAll () y VideoDisc.PlayMovie () son semánticamente equivalentes pero tienen nombres diferentes porque han sido programados en distinto momento y por diferentes personas.

Un fragmento del programa que usa estas clases aparece a continuación:

```
IList s = new ArrayList();
s.Add(new CompactDisc());
/* Otros discos compactos son creados y agregados aquí… */
s.Add(new VideoDisc());
/* Otros videodiscos son creados y agregados aquí… */
foreach (Object o in s) {
   if (o is CompactDisc) {
      ((CompactDisc)o).playAll();
   } else {
      ((VideoDisc)o).playMovie();
   }
}
```

4.1 ¿Qué principio o principios se violan en el código anterior? ¿Por qué?
4.2 Escribe una versión del código que funcione correctamente y no viole estos principios. Pueden hacer todas las modificaciones que consideren necesarias.

5.1 Un objeto se manda un mensaje a sí mismo. El método que se ejecuta como consecuencia, ¿tiene que estar implementado en la clase de ese objeto? Justifiquen la respuesta en forma breve y concreta. Muestren la respuesta en un programa de ejemplo si fuera posible.
5.2 Una clase cualquiera puede heredar de una clase abstracta o implementar una interfaz. Desde el punto de vista de los **tipos** ¿qué similitudes y diferencias existen en ambos casos?

Sean las siguientes clases:

```
public class Cuenta {
   private Double saldo = 0;
   public Double Saldo { get { return saldo;} set {saldo = value;}}
}
public class Banco {
   private IList clientes = new ArrayList();
   public IList Clientes { get { return clientes;} set {clientes = value;}}
```

```
}
public class Cliente {
   private Cuenta cuenta;
   public Cuenta Cuenta { get { return cuenta; }}
   private String nombre;
   public String Nombre { get { return nombre; }}
   public Cliente(String nombre, Cuenta cuenta) {
      this.nombre = nombre;
      this.cuenta = cuenta;
   }
}
public class Cajero {
   private Banco b = new Banco();
   /* otro codigo altamente secreto */
   public void Debitar(String nombreCliente, Double monto) {
      foreach (Cliente c in b.Clientes)
         if (c.Nombre.Equals(nombreCliente)) {
            Cuenta cuenta = c.Cuenta;
            if (monto < cuenta.Saldo)
               cuenta.Saldo = cuenta.Saldo - monto;
         }
   }
}
```

6.1 Crítica el código de void Debitar(String nombreCliente, Double monto) en base a criterios de cohesión y acoplamiento. Justifica tus críticas.
6.2 Programa el código nuevamente disminuyendo el acoplamiento.

7.
Sean las siguientes clases:

```
public class Sensor {
   private Double presion;
   public virtual Double GetPresion() {
            return presion;
   }
   public virtual void SetPresion(Double value) {
            presion = value;
   }
   public virtual void Configurar() {
      /* código gigantesco, tan grande que cut&paste da out of memory */
   }
}
public class Beeper {
   public void Beep() {
      /* hace "bep bep bep beeep beeep beeep bep bep bep" */
   }
}
```

La clase Sensor representa un sensor de presión. El método Double GetPresion() retorna la presión actual. Cuando se mide una nueva presión, el hardware mágicamente llama a void SetPresion(Double value) con la nueva presión para actualizar el valor del sensor de presión.

La clase Beeper tiene la responsabilidad de emitir un "*beep*" o alarma. En este caso, el *beep* consiste en una señal de SOS cuando se ejecuta el método void Beep().

Debes programar una clase usando **composición y delegación** llamada SensorBeeper que debe tener por lo menos los mismos métodos que la clase Sensor. Los sensores SensorBeeper deben emitir un "*beep*"o alarma cuando la presión asignada al sensor supera un cierto valor.

Debes programar tu clase de forma que sea posible cambiar **en tiempo de ejecución** la forma en que se hace "beep" y debes explicar por qué ésto es posible. No tienes que implementar otras posibles formas de hacer "beep", solo dejar tu programa abierto a que pueda hacerlo de otras formas en el futuro.