PROGRAMACIÓN ORIENTADA A OBJETOS **12/02/2008**

NOMBRE: ______________________________ **Titulación:** ____________

ESTADO DE LAS PRÁCTICAS: ______________________________

1. Para la implementación de la clase `Figura`, se ha confiado en el uso de un atributo `forma` para distinguir entre distintos tipos de objetos:

```
public class Figura {
      String forma;
      double escala;

      public Figura(String forma, double escala) {
         this.forma = forma;
         this.escala = escala;
      }
      public double getArea() {
            if (forma.equals("cuadrado")) {
                  return escala * escala;
            }
            else if (forma.equals("circulo")) {
                  return Math.PI * escala * escala;
            }
            else { // forma.equals("triangulo"), un triángulo equilátero
                  return escala * (escala * Math.sqrt(3) / 4);
            }
      }
}
```

(**1 pto**) Explica si la implementación de la clase `Figura` se ajusta o no a los ***principios de diseño modular***. Piensa qué pasaría si hacemos `new Figura("rombo", 10.0)` o `new Figura ("cudado",10.0);`. En el caso de que viole alguno de los principios de diseño debes indicar cuál e implementar la solución más adecuada para representar las figuras.

2. (**0'5 ptos**) La clase C# `PeriodoTiempo` incluye un único atributo que almacena el tiempo en segundos. Completa la implementación de la clase `PeriodoTiempo`, sin añadir ningún atributo más, de manera que el código de la derecha sea correcto (NOTA: una hora son 3600 segundos).

```
class PeriodoTiempo
{
    private double segundos;

    //Completar …

}
```

```
class Program
{
    static void Main()
    {
       PeriodoTiempo t = new PeriodoTiempo();
       t.Horas = 24;
       System.Console.WriteLine("Tiempo en horas: " +
                                          t.Horas);
    }
}
```

3. a) (**0'5 ptos**) ¿Existe algún mecanismo en Eiffel equivalente a la declaración de ***clases amigas*** de C++? Justifica la respuesta.

b) (**1 pto**) Escribe el código Java equivalente al siguiente código Eiffel:

```
class A
  creation make1, make2
  feature{ }
      at2: expanded B
  feature{ALL}
      at1: INTEGER
      make1 is do end
      make2(v:INTEGER) is do
            at1:=v
      end
end --A
```

4. La clase `LineaTelefono` incluye un método `marcar` que se encarga de establecer la comunicación vía `modem` con el número de teléfono que se le pasa como parámetro. En el caso de que la llamada del modem fracase, se reintentará un número limitado de veces (`MAX_FALLOS`):

```
class LineaTelefono feature
   …
   marcar(numero:NumeroTelefono) is
     local
           fallos : INTEGER
           exito: BOOLEAN
     require
           numero.esValido
     do
         modem.llamar(numero)  --La llamada puede fracasar
         exito := true
     ensure
           modem.estaConectado;
     rescue
         if numero.esValido AND NOT exito then
                fallos := fallos +1
                if (fallos < MAX_FALLOS) then
                        esperar; -- espera un tiempo aleatorio
                        retry;
                end
         end
   end -- marcar
  end -- LineaTelefono
```

a) (**1 pto**) Explica la implementación del método `marcar` de la clase `LineaTelefono` (cláusulas **`require`**, **`ensure`** y **`rescue`**).

b) (**1 pto**) Implementa el código Java equivalente para el método `marcar`. En el caso de que utilices excepciones debes indicar y justificar el tipo de cada una de ellas (Runtime o comprobada).

5. Dado el siguiente código Eiffel con ***herencia repetida***:

```
deferred class Interprete feature
      nombre : STRING
      interpretar is deferred end
end
class Guitarrista inherit Interprete feature
      guitarra : STRING
      interpretar is do
            mensaje("El guitarrista"+nombre+"toca la guitarra"+guitarra)
      end
end
class Cantante inherit Interprete feature
      estilo : STRING
      interpretar is do
            mensaje("El Cantante"+nombre+"canta con estilo"+estilo)
      end
end
class Cantautor inherit Guitarrista
                  redefine interpretar
                inherit Cantante
                  redefine interpretar
feature
      interpretar is do
            mensaje("El Cantautor"+nombre+"canta con estilo"+estilo+
                   "y toca la guitarra"+guitarra)
      end
end
```

a) (**0'5 ptos**) ¿Existe algún conflicto en la jerarquía definida en Eiffel? Justifica la respuesta.

b) (**0'5 ptos**) Implementa en C++ una jerarquía equivalente.

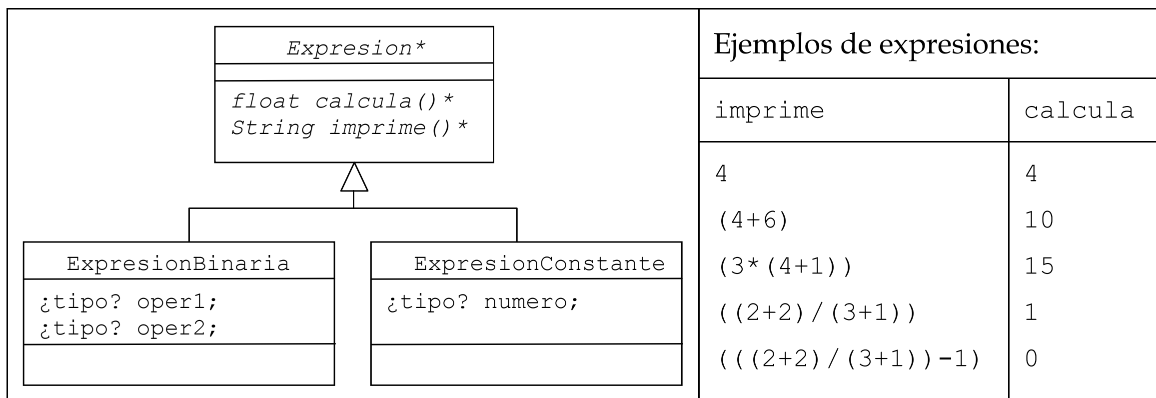
6. En una empresa se guarda un fichero de log con apuntes textuales de las transacciones que se van haciendo como se ve en el código de la clase `Transaccion`. De acuerdo con la ley de protección de datos, este tipo de información se debe encriptar antes de registrarse en el fichero de log. Existen distintos tipos de algoritmos de encriptamiento cada uno de ellos permite encriptar (`String encriptar (String textoSinEncriptar)`) y desencriptar un texto (`String desencriptar(String textoEncriptado)`). Dependiendo del tipo de transacción el algoritmo de encriptamiento será distinto, pero todas las transacciones del mismo tipo utilizan el mismo algoritmo de encriptamiento. Por ejemplo, las transacciones que manejan datos sensibles deben utilizar un algoritmo más seguro.

```
public abstract class Transaccion{
      …
      public void salvar(){
            log.registrar(this.toString());
      }
}
```

a) (**0'75 ptos**) Explica y modifica el código de la clase `Transaccion` para tener en cuenta los distintos algoritmos de encriptamiento a la hora de salvar las transacciones.

b) (**0'75 ptos**) Supongamos que no se salva cada transacción de manera individual sino que se van agrupando (`List<Transaccion> transacciones;`) y en un momento dado se salvan todas las transacciones que se han agrupado invocando al método `salvarTodas(List<Transaccion> transacciones)`. Utiliza la implementación de los ***iteradores internos*** vista en clase para implementar el método `salvarTodas`. (NOTA: hay que implementar tanto el método `salvarTodas` como el iterador necesario para dicha implementación).

7. Necesitamos proporcionar una solución para el cálculo e impresión de ciertas expresiones aritméticas. Las expresiones con las que vamos a trabajar pueden ser constantes, es decir, cualquier número real, o pueden ser binarias (suma, resta, multiplicación y división), es decir, representa operaciones sobre dos expresiones cualquiera. En el primer caso, expresiones constantes, el cálculo de la expresión es el propio número, mientras que en el caso de las expresiones binarias se tiene que sumar, restar, multiplicar o dividir los dos operandos que la componen (`oper1` y `oper2`).



Ejemplos de expresiones:

imprime	calcula
4	4
(4+6)	10
(3*(4+1))	15
((2+2)/(3+1))	1
(((2+2)/(3+1))-1)	0

a) (**0'5 ptos**) Completa la jerarquía de la figura: (1) estableciendo los tipos de los atributos, (2) especificando si las subclases de `Expresion` son abstractas o efectivas y (3) añadiendo los mecanismos necesarios para evaluar todos los tipos de expresiones.

b) (**0'5 ptos**) Explica en qué clase y cómo se implementaría el método `calcula` para las expresiones binarias. Utiliza el caso de la expresión binaria de suma para mostrar el código de la implementación.

c) (**0'5 ptos**) Implementa el método `imprime` para las expresiones binarias siguiendo el patrón de diseño del ***Método Plantilla***.

8. a) (**0'5 ptos**) Dada la siguiente clase genérica en Java:

```
public class Contenedor<T> {
      private List<T> contenido = new LinkedList<T>();

      public void add(T elemento) {
            contenido.add(elemento);
      }
}
```

Y la clase `Jefe` que hereda de `Empleado`, explica por qué el compilador no da por bueno el siguiente código:

```
Contenedor<Empleado> plantilla = new Contenedor<Jefe>();
```

b) (**0'5 ptos**) Explica por qué todos los lenguajes orientados a objetos que hemos estudiado (Eiffel, Java, C++ y C#) incluyen la definición de clases genéricas como elemento del lenguaje.