Paradigmas de la Programación – Segundo Parcial

11 de Junio de 2015

Apellido y Nombre: _____

Multiple Choice [10 pt.] Seleccione todas las respuestas correctas entre las diferentes opciones que se dan para completar cada oración:

1. La diferencia entre interfaces de Java y clases abstractas de C++ es que...
   a) las clases abstractas de C++ se pueden usar para expresar herencia múltiple en sus descendientes, mientras que las interfaces de Java no.
   b) las interfaces de Java no pueden heredar de ninguna clase, mientras que las clases abstractas de C++ sí.
   c) las interfaces de Java se pueden usar para expresar herencia múltiple en sus descendientes, mientras que las clases abstractas de C++ no.
   d) las clases abstractas de C++ no pueden ser instanciadas, mientras que las interfaces de Java sí.
2. Tenemos un *name clash* cuando...
   a) una clase hereda dos miembros con el mismo nombre de dos de sus madres.
   b) las signaturas de dos clases cualquiera tienen un miembro en común.
3. Señale cuáles de las siguientes expresiones son `true` en Prolog, es decir, en qué casos se encuentra una unificación exitosa para la expresión:
   a) `'Pan' = pan`
   b) `Pan = pan`
   c) `comida(pan) = pan`
   d) `comida(pan,X) = comida(Y,salchicha)`
   e) `comida(pan,X,cerveza) = comida(Y,hamburguesa)`
   f) `comida(X) = X`
   g) `meal(comida(pan),bebida(cerveza)) = meal(X,Y)`
   h) `meal(comida(pan),X) = meal(X,bebida(cerveza))`

Programación Lógica [20 pt.] La búsqueda en Prolog es exhaustiva, mientras que en otros lenguajes se ejecuta la sentencia controlada por el primer patrón que hace *pattern matching*. Sin embargo, existe una forma en prolog para cortar una búsqueda exhaustiva, usando el operador `cut`, que termina la búsqueda cuando se encuentra el primer patrón que hace *pattern matching*.

El siguiente fragmento de Prolog es un ejemplo del uso de `cut`. Escriba un pseudocódigo en imperativo (usando `if ... then ... else ...`) o funcional (usando guardas) con la misma semántica.

```
apostar(X) :- tenerdinero(X),!.
apostar(X) :- tomarprestado(X).
```

Herencia múltiple [20 pt.] Cuando tenemos herencia múltiple hay que resolver la herencia de información conflictiva de las diferentes clases madre. Detecte en el siguiente código información conflictiva heredada de las diferentes clases madre y especifique en lenguaje natural por lo menos dos estrategias posibles para resolverla (por ejemplo, la estrategia de Ruby con `mixins` y la estrategia de C++, pero también podría especificar una heurística de su propia autoría para resolver el problema).

```
class DispositivoUSB
{
private:
    long ID;

public:
    DispositivoUSB(long lID)
        : ID(lID)
    {
    }

    long ObtenerID() { return ID; }
};

class DispositivoRed
{
private:
    long ID;

public:
    DispositivoRed(long lID)
        : ID(lID)
    {
    }

    long ObtenerID() { return ID; }
};

class AdaptadorWireless: public DispositivoUSB, public DispositivoRed
{
public:
    AdaptadorWireless(long lUSBID, long lNetworkID)
        : DispositivoUSB(lUSBID), DispositivoRed(lNetworkID)
    {
    }
};

int main()
{
    AdaptadorWireless c54G(5442, 181742);
    cout << c54G.ObtenerID();

    return 0;
}
```

Acoplamiento [20 pt.] El grado de interdependencia de dos componentes de software se puede expresar en términos de acoplamiento. En las siguientes versiones del mismo programa, argumente qué versión es más acoplada que la otra y por qué.

```
class RepositorioClientes {

    private readonly BasededatosID basededatos;

    public RepositorioClientes(BasededatosID basededatos)

    { this.basededatos = basededatos; }

    public void Aniadir(string NombreCliente)

    { basededatos.AniadirFila("Cliente", NombreCliente); }
}

interface BasededatosID {

    void AniadirFila(string Tabla, string Valor);
}

class Basededatos : BasededatosID {

    public void AniadirFila(string Tabla, string Valor) {}
}
```

```
class RepositorioClientes {

    private readonly Basededatos basededatos;

    public RepositorioClientes(Basededatos basededatos)

    { this.basededatos = basededatos; }

    public void Aniadir(string NombreCliente)

    { basededatos.AniadirFila("Cliente", NombreCliente); }
}

class Basededatos {

    public void AniadirFila(string Tabla, string Valor) {}
}
```

Concurrencia [20 pt.] Escriba en pseudocódigo dos versiones secuenciales (no concurrentes) del siguiente programa, una en paradigma funcional y otra en paradigma imperativo. Argumente las ventajas y desventajas de las tres versiones del programa usando los conceptos de *velocidad*, *overhead* y *determinismo*.

```
class Ejemplo extends RecursiveTask<Integer> {
  final int n;
  Ejemplo(int n) { this.n = n; }
  Integer compute() {
    if (n <= 1)
       return n;
    Ejemplo f1 = new Ejemplo(n - 1);
    f1.fork();
    Ejemplo f2 = new Ejemplo(n - 2);
    return f2.compute() + f1.join();
  }
}
```

Visibilidad [10 pt.] En el siguiente código en Ruby, describa la visibilidad de la variable `cuenta`.

```
class Ser

    @@cuenta = 0

    def initialize
        @@cuenta += 1
        puts "creamos un ser"
    end
    def muestra_cuenta
        "Hay #{@@cuenta} seres"
    end
end
class Humano < Ser
   def initialize
       super
       puts "creamos un humano"
   end
end
class Animal < Ser
   def initialize
       super
       puts "creamos un animal"
   end
end
class Perro < Animal
   def initialize
       super
       puts "creamos un perro"
   end
end

Humano.new
d = Perro.new
puts d.muestra_cuenta
```