Examen de Prácticas de Programación

Septiembre 2006

Cuestiones (4 cuestiones, 30% de la nota)

1) Se dispone de una hilera de bolas que pueden ser azules, blancas, o rojas. Se dispone de un brazo mecánico para manipular las bolas que puede realizar bajo control de un programa las operaciones de obtener el color de una bola y colocar una bola en alguna de las tres hileras siguientes: izquierda, centro, derecha. Se desea derivar un algoritmo que recorriendo una sola vez la hilera de bolas coloque las bolas según los colores de la bandera holandesa: azul (izquierda), blanca (centro), roja (derecha).

   Para resolver el problema supondremos que disponemos de una clase llamada `Bola` de la que sabemos dispone de un método `color( )` que retorna el color de una bola; a su vez los colores pueden compararse mediante un método estático `igual(c1,c2: color)` que retorna booleano. Para modelar una hilera usamos una pila de bolas (clase `PilaDeBolas`) con las operaciones habituales de la clase de pilas:

```
método pilaVacía() retorna PilaDeBolas
método apilar(b: Bola) retorna PilaDeBolas
método desapilar() retorna PilaDeBolas
método cima() retorna Bola
método esVacia() retorna booleano
```

   El algoritmo pedido se encapsula en una acción con la siguiente especificación:

```
acción banderaHolandesa(ent p:PilaDeBolas,
                        sal: izq, cent, der: PilaDeBolas)
{Pre.- p representa la hilera inicial de bolas
       azules, blancas o rojas}
{Post.- izq contiene las bolas azules de p,
        cent las blancas de p y der las rojas}
```

   Diseñar la acción especificada anteriormente sin utilizar ningún contador.

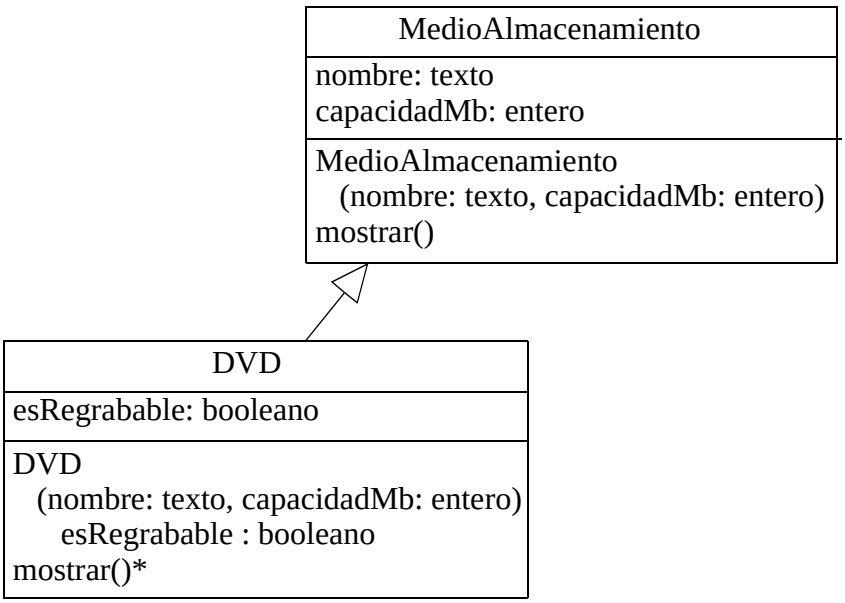
2) Comparación entre estructuras complejas. En algunas ocasiones interesa comparar tipos de datos estructurados tales como pilas, colas, árboles, tablas, etc. Si por ejemplo T1 y T2 son árboles binarios, la igualdad puede definirse recursivamente de la manera siguiente:

```
iguales(T1,T2)= cierto si T1 y T2 son vacíos
iguales (T1,T2)= falso si alguno es vacío y el otro  no
iguales(plantar(x,T1,T2),plantar(y,T1',T2'))=
          (x=y)&&iguales(T1,T1')&&iguales(T2,T2')
```

   Se pide, usando las operaciones del tipo árbol binario y la definición anterior, diseñar una función que tomando como parámetros dos árboles binarios determine si son iguales.

3) Se dispone del diagrama de clases que se muestra en la figura. Los constructores copian los parámetros en los respectivos atributos, mientras que los métodos muestra muestran en pantalla todos los atributos del objeto. El constructor y el método muestra de la clase

`DVD` utilizan el constructor y método muestra de su superclase, respectivamente. Escribir un par de clases Java que lo implementen.



* operación redefinida

4) Escribir un método estático en Java que escriba en un fichero de texto cuyo nombre se le pasa como parámetro todos los `String`s contenidos en un array, que también se le pasa como parámetro. Se escribirán uno por línea.

Examen de Prácticas de Programación

Septiembre 2006

Problema (30% de la nota)

Se desea hacer un sistema para controlar una planta compuesta por un conjunto de paneles solares que producen electricidad. Cada panel solar individual se representa en el computador por medio de un objeto de la clase `PanelSolar`, cuya interfaz es:

```
/**
 * Clase que representa un panel solar
 */
public class PanelSolar
{

    /**
     * Constructor al que se le pasa el identificador del panel
     */
    public PanelSolar(String id) {...}

    /**
     * Retorna el identificador del panel
     */
    public String id() {...}

    /**
     * Método para alinear el panel; retorna true si se ha conseguido
     * alinear con el sol, y false en caso contrario
     */
    public boolean panelAlineado() {...}

    /**
     * Retorna el acimut del panel, en grados, que nos indica
     * hacia dónde apunta en la dirección horizontal.
     * El Sur se representa con el valor 180 grados.
     * Lanza Averiado si el panel está averiado
     */
    public double acimut() throws Averiado{...}

    /**
     * Retorna la elevación del panel, en grados, sobre el horizonte.
     * Cuando el panel apunta al horizonte el valor es 0 grados
     * Lanza averiado si el panel está averiado
     */
    public double elevacion() throws Averiado {...}

    /**
     * Retorna la potencia eléctrica, en vatios, que el panel
     * está produciendo en este instante
     */
    public double potencia() {...}
}
```

Lo que se pide es implementar la clase `PlantaSolar`, que debe responder a la siguiente interfaz:

```
public class PlantaSolar
{
    public void anadePanel(PanelSolar p) throws NoCabe, YaExiste {...}
    public void alinea() {...}
    public PanelSolar panelAnomalo() {...}
}
```

La clase debe contener los siguientes atributos privados

- `panel`: un lista de objetos de la clase `PanelSolar`; usar la clase `LinkedList`
- `acimutMedio`: un número real que indica el acimut medio en grados; valor inicial 0.0
- `elevMedia`: un número real que indica la elevación media en grados; valor inicial 0.0

Los métodos de la clase `PlantaSolar` deben hacer lo siguiente:

- `anadePanel`: Añade un nuevo panel solar, que se pasa como parámetro, al final de la lista, siguiendo los siguientes pasos. En primer lugar comprueba si ya existe un panel con el mismo identificador, y en ese caso lanza `YaExiste` (para ello habrá que recorrer los paneles que tenemos almacenados, comparando sus identificadores con el de `p`). Si no hubo errores (es decir, si no se lanza la excepción), añade el panel `p` al final de la lista.
- `alinea`: Alinea todos los paneles de la planta y con los que resulten correctamente alineados y no estén averiados calcula el acimut y elevacion medios, almacenándolos en los respectivos atributos. Para ello, después de inicializar a cero un par de variables para guardar en ellas respectivamente las sumas de los acimuts y elevaciones de los paneles (para calcular luego la media), y un contador de los paneles solares que están correctos, recorre los paneles solares que tenemos guardados en el array `panel`, y con cada uno de ellos hace lo siguiente:
  - alinea el panel con el sol llamando a `panelAlineado` y guarda el resultado obtenido, que indica si el panel quedó bien alineado
  - si el panel resultó bien alineado, obtener su acimut y elevación y, si no se lanza `Averiado`, añadirlos respectivamente a la suma de acimut y elevación e incrementar el contador de paneles correctos; si se lanza `Averiado`, no hacer nada y seguir con el siguiente panel.

  al finalizar, calcular el acimut medio y la elevación media (como la suma entre el contador de paneles correctos) y almacenarlos en los respectivos atributos
- `panelAnomalo`: Encuentra y retorna un panel anómalo (el primero que se encuentre), que es aquel que sin estar averiado (es decir, que sus métodos no lanzan `Averiado`), su acimut o elevación se diferencia en más (o menos) de un 5% de los valores medios almacenados en los atributos respectivos. Retorna `null` si no encuentra ningún panel solar que cumpla esa condición.

Las excepciones están definidas en clases aparte de la forma:

```
public class Averiado extends Exception{}
public class YaExiste extends Exception{}
```

Nota: Cada parte de la planta solar se valorará según su dificultad, del siguiente modo:

- `anadePanel`: 15%
- `alinea`: 50%
- `panelAnomalo`: 35%