Nombre: ..... Fecha: / 10 / 2008

Grupo: 1 ☐ 2 ☐ 3 ☐ 4 ☐

PRÁCTICA 3
EJEMPLOS DE OCULTACIÓN DE LA INFORMACIÓN

En esta práctica se pretende que los alumnos comprendan la noción de "Ocultación de la Información" en la POO y la apliquen en algunos de los ejemplos de clases que hemos introducido en las prácticas anteriores. La ocultación de la información hace uso de la división en estado (atributos) y comportamiento (métodos) propia de las clases para asegurar que el comportamiento de una clase será el mismo, aun cuando los atributos de la misma varíen.

Podríamos plantear el problema de la siguiente manera, distinguiendo entre el punto de vista el programador de la clase y de los usuarios (o clientes) de la misma:

1 El programador de una clase debe asegurar que la parte pública de la misma (generalmente formada por los métodos) va a tener siempre el mismo comportamiento

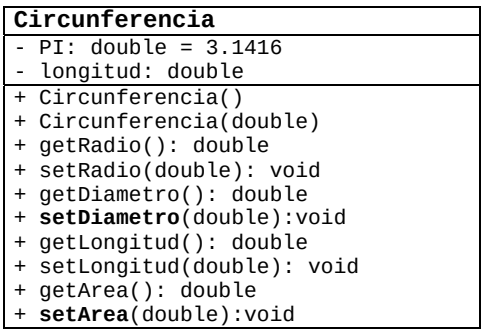
2 Los clientes de una clase deben estar seguros de que siempre que invoquen a un método (o atributo) de la parte pública de una clase, éste va a tener el mismo comportamiento

Partiendo de esa base, hay situaciones en el mundo real donde el programador de una clase decide cambiar la forma de representar dicha clase. Por ejemplo, se puede decidir cambiar el tipo de algunos atributos de la clase para facilitar las operaciones con los mismos (cambiar los atributos de tipo "char *" en C++ por atributos de tipo "string"), o modificar los propios atributos (por ejemplo, si hemos comprobado que una propiedad de una clase es mucho más requerida que otra, decidir guardar la primera como atributo en lugar de la segunda).

En estas situaciones, el programador de la clase ha de ser capaz de modificar dicha clase sin que los clientes de la misma aprecien ningún cambio en el comportamiento de la misma.

Un segundo objetivo de la práctica es utilizar el modificador "static" a la hora de definir atributos y métodos en clases, y ver las utilidades y peculiaridades del mismo.

1. Recuperamos el código de la clase **Circunferencia** programada en la práctica 1 en C++. Tras utilizarla durante algún tiempo, hemos podido comprobar como las propiedades más utilizadas por los clientes de la misma son los métodos que permiten conocer y modificar la longitud de una circunferencia (getLongitud(), setLongitud(double)). Para optimizar el uso de nuestra clase, decidimos crear una nueva representación para la misma que se corresponde con el siguiente diagrama UML:



Programa la nueva clase en ficheros "Circunferencia.h" y "Circunferencia.cpp".

2. Recupera el cliente de la clase "Principal_prac1_1.cpp" y verifica que dicho cliente obtiene exactamente el mismo resultado al utilizar la clase **Circunferencia** programada en la Práctica 1 que al utilizar la programada en el ejercicio 1 (En este caso, el mismo resultado quiere decir exactamente la misma salida por la consola MSDOS).

3. Realiza las mismas modificaciones que has llevado a cabo en el ejercicio 1 de la práctica sobre el fichero "Circunferencia.java".

4. Comprueba con tu cliente "Principal_prac1_1.java" que el comportamiento obtenido con la nueva representación de la clase **Circunferencia** es el mismo que obtenías anteriormente.

5. Vamos a introducir ahora dos nuevos atributos en nuestra clase **Circunferencia**. Uno de ellos (lo llamaremos contadorCircunferencias) nos va a permitir conocer cuántas instancias (u objetos) de la clase han sido generados. El segundo nos va a permitir conocer el número de veces que son invocados los métodos "getArea()" y "setArea (double)" (lo denominaremos contadorAreas).

Dichos atributos los define el programador de la clase para controlar el número de accesos a la clase, así que los debes definir como privados. Además, son atributos de la clase, no de los objetos particulares, por lo cual debes definirlos como estáticos. Por último, su valor inicial será 0, hasta que uno de los constructores o de los métodos "getArea()" y "setArea (double)" sean invocados.

Modifica los constructores "Circunferencia ()" y "Circunferencia (double)" para que incrementen el valor de "contadorCircunferencias" cada vez que sean invocados, y muestren el valor "contadorCircunferencias" por pantalla. Modifica los métodos "getArea()" y "setArea(double)" para que incrementen el valor de "contadorArea" cada vez que sean invocados y muestren el valor "contadorArea" por pantalla.

El diagrama UML de la clase ahora ha pasado a ser:

Circunferencia
- PI: double = 3.1416 - longitud: double - contadorCircunferencias: integer - contadorAreas: integer
+ Circunferencia() + Circunferencia(double) + getRadio(): double + setRadio(double): void + getDiametro(): double + **setDiametro**(double):void + getLongitud(): double + setLongitud(double): void + getArea(): double + **setArea**(double):void

(Observa como los atributos estáticos de la clase, siguiendo la notación UML, aparecen subrayados)

Recuperamos ahora el tercer ejemplo de las prácticas 1 y 2, en el cual representábamos un punto en el plano por medio de la siguiente clase UML:

Punto
- coord_x: double - coord_y: double
+ Punto() + Punto(double,double) + getX(): double + setX(double): void + getY(): double + **setY**(double):void

6. Modifica los ficheros "Punto.h" y "Punto.cpp" para obtener una representación de la clase que responda al siguiente diagrama UML (es decir, en la que los atributos elegidos sean las coordenadas polares del punto):

Punto
- coord_polar: double - coord_angular: double
+ Punto() + Punto(double,double) + getX(): double + setX(double): void + getY(): double + **setY**(double):void

(Nota: te pueden resultar de utilidad algunas de las funciones de la clase "cmath" de C++, como sin (...), cos (...), atan2 (...)).

7. Comprueba, haciendo uso del cliente "Principal_prac1_3.cpp", que el comportamiento de la clase se ha mantenido inalterado.

8. Modifica el fichero "Punto.java" para obtener una representación de la clase **Punto** basada en coordenadas polares.

(Nota: te pueden resultar de utilidad algunos de los métodos disponibles en la clase Math, especificada en http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Math.html como Math.atan2(...), Math.hypot(...), Math.cos(...) o Math.sin(...))

9. Comprueba, haciendo uso del cliente "Principal_prac1_3.java", que el comportamiento de la clase se ha mantenido inalterado.

ENTREGAR los archivos *Circunferencia.cpp, Circunferencia.h, Principal_prac1_1.cpp, Circunferencia.java, Principal_prac1_1.java, Punto.h, Punto.cpp, Principal_prac1_3.cpp, Punto.java, Principal_prac1_3.java*. Los archivos deben contener **todos los cambios** realizados durante los ejercicios de la práctica. La entrega es a través de Belenus en el repositorio de Julio Rubio. También deben contener unas cabeceras que permitan identificarte:

/* Nombre: ___
Grupo: _____
Nombre del fichero: ____
Descripción: _____*/

Guarda una copia de los ficheros para las prácticas sucesivas