UNIVERSIDADE DA CORUÑA

PROGRAMACIÓN ORIENTADA A OBJETOS. CURSO 2007-2008

Boletín de Ejercicios 3

1. Aprovechando la clase `Circulo` creada en el ejercicio 2 del boletín anterior, crea ahora una superclase abstracta `Figura` que generalice el comportamiento básico de una figura entre el que incluimos: (1) su posición central como un punto en el espacio bidimensional (X, Y) incluyendo un constructor que permita pasar por parámetro dichos valores y, (2) el cálculo del área y el perímetro de la figura. Generaliza a `Figura` todas aquellas propiedades de `Circulo` que veas oportuno y crea una nueva subclase `Rectangulo` que extienda a `Figura` y que sobreescriban sus métodos. Incluye en `Rectangulo` métodos constructores (incluyendo un constructor de copia), ten en cuenta que los constructores de las subclases lo primero que hacen es llamar a los constructores de las superclases. Crea también métodos de lectura y escritura para los atributos propios (ancho, alto, radio, etc.).

   Crea una clase `TrataFiguras` que tenga un método estático `sumaAreas` que acepta como parámetro un array de figuras y que devuelve como resultado la suma de todas las áreas de las figuras incluidas en el array. Crea otro método estático denominado `sumaRadios` que devuelva la suma de los radios de los círculos incluidos en el array. Ahora se plantean dos alternativas: (1) situar el método `getRadio` únicamente en la clase `Circulo` (con lo cual el método `sumaRadios` necesita realizar un typecast sobre las figuras del array) o, (2) sitúa `getRadio` en la clase `Figura` con un comportamiento por defecto (por ejemplo, devolver cero). De esta forma nos evitamos el typecast en `sumaRadios` pero la clase `Rectángulo` heredará un método que no necesita. Decide que es más razonable, la flexibilidad de no usar typecasts (si aparece otra figura con radios, como los cilindros, no hay que modificar `sumaRadios`) frente a la coherencia de no aumentar innecesariamente el interfaz de las superclases (que sentido tiene calcular el radio de un rectángulo). Indica en la prueba del ejercicio qué solución has escogido

   Finalmente crea una clase `Cilindro` que herede que `Figura`, ¿qué métodos estamos obligados a implementar en `Cilindro`? Como los cilindros tienen radio ¿es necesario que hagamos un cambio en el método `sumaRadios`? Dependerá un poco de la opción escogida.

   Tema: Polimorfismo de inclusión, tipificación, ligadura dinámica

2. Crea una clase `Colecciones` que tenga los siguientes métodos de utilidad, presta especial atención a cómo definir los parámetros y tipos de retorno de los métodos:
   - `mayor`: Este método recibe por parámetro un objeto de tipo `List` de cualquier tipo y un objeto de tipo `Comparator` que compara cualquier superclase del tipo de la lista. Debe devolver el mayor elemento de la lista según el comparador pasado por parámetro.
   - `sumaTodos`: Este método acepta como parámetro a una `List` de un tipo genérico `T` que implementa el interfaz `Operable` visto en el ejercicio 8 del boletín 2. Devuelve el resultado de aplicar la operación `suma` del interfaz `Operable` sobre todos los elementos de la lista.

- `elevaCuadrado`: Este método acepta como parámetro a una `List` de un tipo genérico `T` que implementa el interfaz `Operable` visto en el ejercicio 8 del boletín 2. El método no devuelve nada y lo que hace es elevar al cuadrado cada elemento de la lista haciendo uso de la operación `multiplicacion` del interfaz `Operable` .

Tema: Uso de extend, super y wildcards con genericidad.
De utilidad: En el método `sumaTodos` o `elevaCuadrado` se hace necesario especificar un parámetro genérico tal y como se muestra en las transparencias 76 y 77 del tema 3.

3. Una bolsa puede ser definida como una colección de elementos entre los que no se establece ninguna relación de orden. Una vez que introducimos un elemento en la bolsa no podremos decir en qué posición va a quedar dicho elemento con respecto al resto de elementos de la bolsa. De la misma forma, al extraer un elemento de la bolsa todos tienen la misma posibilidad de ser extraídos. Teniendo esto en cuenta crea una interfaz `Bag`, que herede de `Collection` y cuyo objetivo sea definir el comportamiento de una bolsa. Dicho interfaz debe tener los siguientes métodos:
   - `boolean isEmpty()` → Devuelve true si la bolsa no tiene elementos
   - `boolean add(E e)` → Añade el elemento especificado a la bolsa, devuelve true si la bolsa ha cambiado después de realizar la operación.
   - `boolean addAll(Collection<? extends E> c)` → Añade a la bolsa todos los elementos de la colección pasada por parámetro.
   - `public Iterator<E> iterator()` → Devuelve un iterador que permite recorrer los elementos de la bolsa. Cómo el método para extraer elementos de la bolsa es `extract`, el iterador no debe implementar el método `remove` (lanzará una excepción).
   - `void clear()` → Elimina todos los elementos de la bolsa
   - `int size()` → Devuelve el número de elementos de la bolsa
   - `boolean extract()` → Extrae un elemento de la bolsa, como todos los elementos tienen la misma probabilidad de ser extraídos deben usarse números pseudoaleatorios para decidir qué elemento debe ser seleccionado
   - `void emptyIn(Bag b)` → Vacía el contenido de la bolsa actual en la bolsa pasada por parámetro. La bolsa actual queda vacía

   Como vemos salvo los métodos `extract` y `emptyIn` el resto son métodos del interfaz `Collection`. Sin embargo, en `Collection` hay muchos más métodos que los aquí listados. Para poder implementar dicho interfaz y no complicar en exceso el ejercicio los métodos de `Collection` no nombrados en la lista anterior deben sobreescribirse pero creando un método que tenga la siguiente línea: `throw new UnsupportedOperationException("Not supported yet.");` , indicando que su implementación no es necesaria en esta momento. También por motivos de sencillez se puede suponer que la bolsa nunca se va a llenar (en el almacenamiento interno reservaremos espacio suficiente para que esto no ocurra en el código de prueba).

   El código de prueba consistirá en probar los métodos arriba descritos mediante pruebas propias y también usando la colección de forma genérica pasándola por parámetro a alguno de los métodos de utilidad de la clase `Collections`. Por ejemplo, usar el método `max` para calcular el máximo elemento que existe dentro de la bolsa.

   Tema: Colecciones, implementación de interfaces con genericidad
   De utilidad: La parte que puede parecer más complicada es la creación de un iterador, sin embargo, puede ser una buena pista ver el código fuente del `ArrayList` y analizar como los programadores de Java han resuelto el problema.

4. Crea una clase `Factura` que representa a una factura en la que se incluye la siguiente información: cliente al que se le carga la factura, fecha de expedición, lista de items (incluirá descripción, precio unitario y cantidad), precio sin iva, iva aplicable y precio final. El cliente se implementará como una clase `Cliente` y tendrá como identificación su ID (el NIF en las personas o el CIF en el caso de empresas), su nombre y su domicilio. Crea subclases de `Cliente` para representar a las empresas y a las personas. Las empresas añadirán nuevos atributos como número de empleados y las personas atributos del tipo fecha de nacimiento. Crea una clase `Contabilidad` que se encarga de almacenar las facturas pendientes de cobro, esta clase debe tener los siguientes métodos (decide tú como deben ser los parámetros y tipos de retorno):
   - `incluirFactura`: Incluye una nueva factura a la lista de facturas pendientes, los clientes deben pasarse como una referencia genérica de la clase `Cliente` y nunca como una referencia de una subclase.
   - `eliminarFacturas`: elimina todas las facturas de un cliente dado.
   - `procesarFacturas`: Muestra por pantalla la información de cada factura (se supone que es lo que se imprimiría para mandarlo al cliente para proceder a su pago). Cada factura debe incluir al final la información del plazo para proceder al pago. En el caso de los particulares el plazo es de un mes, en el caso de las empresas el plazo es de tres meses

   Para finalizar el ejercicio crea una nueva subclase de cliente que sea `Institución` que tiene como atributos propios un identificador que nos indica si es una institución local, autonómica, estatal o europea y cuyo plazo para proceder al cobro es de seis meses. Comprueba como añadir esta nueva clase a la estructura no ha provocado ningún cambio en las clases ya existentes.

   Tema: Herencia, polimorfismo, ligadura dinámica
   De utilidad: Haz uso del principio abierto-cerrado para que la inclusión de una nueva subclase no signifique la modificación del código ya existente.

5. Crea una clase `MonederoSimple` que servirá para almacenar las monedas de euro de curso legal (1 y 2 euros, 50, 20, 10, 5, 2 y 1 céntimos). La clase `MonederoSimple` servirá para gestionar el cambio de máquinas electrónicas. Básicamente su funcionamiento consistirá en aceptar unas monedas de entrada, esperar una notificación de gasto y después devolver las monedas que no haya utilizado al cliente (similar al funcionamiento de las cabinas de teléfono). Crea una variante que sea `MonederoDeposito` que tiene un deposito adicional de monedas, de tal forma que, siempre tratará de devolver las monedas originales no gastadas, pero si es necesario hecha mano de un depósito de monedas para dar el cambio exacto. Ambos monederos deben tener un método `cancelar` que permite devolver las monedas introducidas por el cliente si se cancela el gasto. Crea también una superclase `Monedero` que defina el comportamiento común de ambos monederos. Crea ahora una clase `MaquinaElectronica` que ofrece una serie de productos con un precio. Para gestionar el pago de los productos a través de monedas y gestionar el cambio la máquina electrónica delega su funcionamiento en una instancia de la clase `Monedero`. A la clase `MaquinaElectronica` se le pasa por parámetro el monedero concreto que va a utilizar y que le es desconocido por completo. Es más, puede que en medio de la ejecución de la `MaquinaElectronica` se decida cambiar el tipo de monedero utilizado.

   Tema: Polimorfismo de inclusión, herencia y ligadura dinámica
   De utilidad: Se trata de la implementación de un patrón de diseño ¿de cuál?

6. Crea una clase `Directorio` que se encarga de almacenar objetos de tipo `Fichero` y también otros objetos de tipo `Directorio`. Para ello crea una superclase común a ficheros y directorios que se denomine `Elemento`. Los ficheros se identificarán por un nombre, un tipo, un tamaño, fecha de creación y fecha de última modificación. Los directorios se caracterizan por su nombre, tamaño, fecha de creación y fecha de última modificación. El tamaño del directorio se calcula dinámicamente como el tamaño de los ficheros y subdirectorios que lo componen. Crea una clase `SistemaDeFicheros` que almacene un elemento especial (que será el directorio raiz `/` . Esta clase se encargará de crear la estructura típica de un árbol de directorios y ficheros que cuelgue del elemento raíz a partir de los métodos que provean las clases `Elemento`, `Directorio` y `Fichero`. Decide qué elementos y métodos debes incluir en `Elemento`, recuerda que se trata de un compromiso entre la flexibilidad de poder tratar de forma genérica a directorios y ficheros, y la congruencia de no subir a las superclases aspectos propios de las subclases.

   Tema: Polimorfismo de inclusión, herencia y ligadura dinámica
   De utilidad: Se trata de la implementación de un patrón de diseño ¿de cuál?