TP2: Clases genéricas y polimorfismo

Programación II

Primer cuatrimestre 2016

La fecha de entrega de este TP es el **16 de junio de 2016**. Se puede realizar de manera individual, o en grupos de a lo sumo dos personas.

El TP consta de dos partes, que se entregarán de manera conjunta en un único ZIP con el código Java, respetando la estructura del proyecto de Eclipse que se proporciona como parte de la consigna.

El proyecto de Eclipse se puede descargar desde esta página.

Parte 1: Estructuras de datos genéricas

En esta parte se pide implementar un diccionario conforme a una interfaz pre-determinada, empleando un TAD Conjunto como soporte.

Como parte de la consigna se incluye:

- la definición de la interfaz *Diccionario*.
- la implementación de la clase *Conjunto*.

Y se pide:

- la implementación de la clases *DiccConjunto* y *TuplaDic*.

Se incluye también una guía de implementación.

La interfaz *Diccionario*

La interfaz pública *Diccionario* define cinco operaciones con dos tipos paramétricos (la clave *K* y el valor *V):*

```
public interface Diccionario<K, V>
{
    void guardar(K clave, V valor);

    V obtener(K clave);

    boolean pertenece(K clave);

    void eliminar(K clave);

    int tamaño();
}
```

La documentación de cada método se incluye en el archivo Diccionario.java.

No se permite realizar ningún cambio a esta interfaz.

La clase *Conjunto*

Se incluye la implementación de un *Conjunto* genérico con el tipo paramétrico *T* (el tipo del elemento):

```
class Conjunto<T extends Comparable<T>>
{
    void agregar(T elem) { ... }

    boolean pertenece(T elem) { ... }

    void quitar(T elem) { ... }

    T recuperar(T elem) { ... }
}
```

De la definición de la clase se deduce que los elementos del conjunto deben ser **comparables entre sí**.

La documentación de cada método se incluye en el archivo Conjunto.java.

Se proporciona la implementación completa de la clase. No se pueden realizar alteraciones, ni agregar métodos adicionales. En particular, nótese que, de manera intencional, este Conjunto no proporciona el método *tamaño()*.

DiccConjunto y *TuplaDic*

La clase *DiccConjunto* debe implementar la interfaz *Diccionario* sobre el TAD *Conjunto*.

Como el número de tipos paramétricos difiere entre estas dos clases, será necesario usar un TAD intermedio que actúe como puente entre *Conjunto<T>* y *DiccConjunto<K, V>*.

Se recomienda usar *TuplaDic<K, V>* como TAD intermedio. Es una tupla de dos elementos en la que la comparación se realiza solamente por el primero de ellos (en este caso la clave).

Archivos a completar:

- TuplaDic.java: implementar `toString()`, `equals()` y `compareTo()`.
- DiccConjunto.java: implementar `guardar()`, `obtener()`, `pertenece()` `tamaño()` y `eliminar()`.

Guía de implementación

1. La idea es emplear un conjunto cuyos elementos sean tuplas `(clave, significado)`. Es decir, instanciar **dentro de DiccConjunto** un conjunto cuyo tipo paramétrico sea *TuplaDic<K, V>*. Así:

   ```
   /**
    * Conjunto privado que DiccConjunto usa para almacenar
    * sus parejas (clave, significado).
    */
   private Conjunto<TuplaDic<K, V>> elementos;
   ```

2. Para que esto funcione, el *equals()* y *compareTo()* de la tupla deben fijarse solamente en el primer elemento (la clave). Así, se puede realizar una búsqueda dentro del conjunto con una tupla `(clave, null)`.

3. Como no hay *Conjunto.tamaño()*, *DiccConjunto* deberá llevar la cuenta de su número de elementos. Para ello, necesitará una variable de instancia *numElementos* que será incrementada en *guardar()* y decrementada en *eliminar()*.

   Puede ocurrir, no obstante que un elemento a guardar ya exista, y por tanto simplemente se reemplace; en ese caso no se debe incrementar

numElementos. Algo similar ocurre con *eliminar()* y elementos que no existen.

Ambos casos se pueden solucionar usando *pertenece()* antes de modificar el valor de *numElementos*.

Parte 2: Polimorfismo y desacoplamiento

En esta segunda parte se pide la refactorización de una clase haciendo uso de polimorfismo como mecanismo para desacoplar componentes.

En otras palabras: se tiene una clase con un único método, demasiado largo, que se desea descomponer en módulos distintos (clases), cada uno de las cuales se encargue de una pequeña parte de la tarea global.

La clase se llama `BDExport`, y guarda una lista de objetos en un archivo. Se proporciona una clase Main que muestra su uso:

```
public static void main(String[] args) {
    BDExport.export("clientes.csv", Formato.CSV, listaClientes());
    BDExport.export("empleados.json", Formato.JSON, listaEmpleados());
}
```

Y esta es la documentación y la firma del método `export()`, tal y como aparece en BDExport.java:

```
/**
 * Exporta una serie de objetos de la base de datos a un archivo.
 *
 * Recibe el nombre del archivo, el formato deseado (CSV o JSON), y
 * la lista de objetos (Empeados o Clientes).
 */
public static void export(String archivo, Formato formato, List<?> objetos) {
    // 70 líneas de código ...
    //
    // Abrir archivo
    // Determinar el tipo del objeto (!)
    // Extraer los atributos según la clase a exportar (!)
    // Exportar a CSV o JSON, con varios switch/if/else combinados (!)
    // Capturar excepciones e imprimir a System.err (!)
}
```

A lo que se quiere llegar es a un código más elegante y extensible que permita, en el futuro, agregar nuevos formatos y tipos de objetos sin tener que reescribirlo entero.

Nota: el diseño de la la solución es libre, pero a continuación se detalla un posible diseño. El alumno puede decidir seguirlo en su totalidad, o en parte, o implementar uno propio desde cero.

Sugerencia de diseño

Se sugiere el uso de una clase abstracta *FormatWriter* y una interfaz *Exportable* tal que el código de `export()` quede como sigue:

```
public static void export(FormatWriter exporter, Iterable<? extends Exportable> objeto
    Atributos attrs = new Atributos();
    for (Exportable e : objetos) {
        attrs.clear();
        e.saveAtributos(attrs);
        exporter.guardar(attrs);
    }
}
```

Y la función `main()`:

```
public static void main(String[] args) {
    try (FormatWriter csv = new CSV("clientes.csv");
         FormatWriter json = new JSON("empleados.json")) {
        BDExport.export(csv, listaClientes());
        BDExport.export(json, listaEmpleados());
    } catch (IOException e) {
        System.err.println("No se pudo realizar la copia de seguridad");
    }
}
```

La interfaz *Exportable*

En lugar de dejar que *BDExport* "averigüe" el tipo de cada objeto a exportar, y extraiga sus atributos con un cast, los propios objetos deberían saber exportarse a sí mismos.

Esto lo pueden conseguir mediante la interfaz sugerida *Exportable:*

```
public interface Exportable
{
    /**
     * Guarda los atributos de la instancia en un diccionario.
     *
     * En esta versión simplificada, los valores siempre son
     * strings (ver clase Atributos).
     */
    void saveAtributos(Atributos attrs);
}
```

Así, se deberían modificar las clases Cliente y Empleado para que implementen esta interfaz.

La clase *FormatWriter* y sus subclases

La clase abstracta *FormatWriter* abstrae el concepto de serializar, en cualquier formato, los atributos de un objeto:

```
public abstract class FormatWriter
{
    public abstract void guardar(Atributos attrs);
}
```