Práctica 4

Herencia, interfaces y excepciones

Inicio: Semana del 14 de marzo.
Duración: 3 semanas.
Entrega: Semana del 11 de abril.
Peso de la práctica: 30%

El objetivo de esta práctica es el diseño de una aplicación utilizando técnicas de programación orientada a objetos más avanzadas que en las prácticas precedentes, como son la herencia, la ligadura dinámica, las excepciones y las *interfaces*. Se trata de servirse de estas técnicas para reducir código redundante, obtener una aplicación fácilmente extensible, y desarrollar un programa bien estructurado que refleje de una forma directa los conceptos del dominio del problema.

En el desarrollo de esta práctica se utilizarán principalmente los siguientes conceptos de Java y de programación orientada a objetos:

- *Herencia y ligadura dinámica*
- *Manejo de excepciones*
- *Uso de interfaces Java*
- *Uso básico de parámetros genéricos*
- *Estructuración en paquetes*

La práctica consiste en el desarrollo de un sistema experto basado en casos (case-based reasoning o CBR) para la clasificación de objetos.

Desarrollaremos un sistema genérico para clasificar objetos con un número fijo de características, cuyo valor se pueda representar como un número real. Los objetos solo podrán pertenecer a una clase. El sistema partirá de una base de datos inicial en la que un experto ha clasificado manualmente una serie de objetos. Esta base la utilizaremos para clasificar los nuevos objetos que lleguen al sistema, por su parecido a objetos de la misma clase en la base de conocimientos.

Probaremos nuestro sistema con la base de datos Iris, disponible en la página de Moodle. La base de datos original se encuentra en https://archive.ics.uci.edu/ml/datasets/Iris [Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.]

Nuestro sistema experto ha de poderse utilizar con instancias de distintos tipos de objetos, los cuales pueden tener distintas características y clases. Para conseguir el grado de abstracción necesario partiremos de las siguientes interfaces:

```
public interface Attribute<T>{
  String getName();
  double fromString(String v) throws AttributeFormatException;
  int getOrdinal();
}

public interface Instance<T>{
  double getValue (Attribute<T> a)
  Type<T> getType();
}

public interface Type<T>{
  public List<Attribute<T>> getAttributes();
  public Instance<T> newInstance(List<String> values) throws InvalidValuesException;
  public List<String> getClasses();
}
```

Cada instancia de datos que gestiona nuestro sistema tiene una serie de atributos o características, y un valor numérico asociado a cada una. Todas las instancias del mismo tipo tendrán los mismos atributos.
Los atributos tienen un nombre, que se obtiene mediante getName(), y una posición, obtenida mediante getOrdinal(). Los atributos se han de poder convertir a un tipo double, a partir de un String, ya que las instancias solo trabajan con valores numéricos, para ello Attribute proporciona el método fromString(String).

La posición del atributo es un número entre 0 y el número de atributos menos 1. Esta posición la utilizaremos más adelante para identificar a qué atributo se corresponde cada columna de datos en el archivo CSV de entrada, o al crear instancias de datos a partir de una lista de valores.
En cada instancia podemos obtener el valor asociado a un determinado atributo, mediante getValue(Attribute<T>)

El parámetro genérico T presente en estas interfaces nos permite trabajar con los distintos objetos de forma más segura, ya que podemos ligar mediante este parámetro, en el código, tipos con características e instancias.

La interfaz Type contiene toda la información que describe nuestro tipo de problema, y permite crear instancias de datos, mediante newInstance(). Devuelve la lista de atributos, con getAttributes(), ordenados por la posición del atributo, y la lista de clases en las que se pueden clasificar las instancias, mediante getClasses().

El siguiente ejemplo ilustra como usar las clases e interfaces anteriores, siendo Iris una clase que implementa Type<Iris>:

```
Type <Iris> iris = new Iris();
//Podemos crear instancias compatibles con nuestro problema,
// pasando una lista de valores, uno por cada atributo:
Instance<Iris> instancia= iris.newInstance( Arrays.asList("5.1","3.5","1.4","0.2") );
//También podemos imprimir la lista de atributos y el valor para una instancia:
for (Attribute<Iris> attr : iris.getAttributes())
  System.out.println(attr.getName() +"="+instancia.getValue(attr));
```

Apartado 1. Modelo de datos (2,5 puntos)

El primer paso para crear nuestra aplicación será implementar las interfaces Attribute, Instance y Type mediante clases. Algunas de estas interfaces se pueden implementar completamente, y en otros casos se creará una clase abstracta que facilite la implementación de la interfaz.

También será necesario crear en este apartado las clases de excepción indicadas anteriormente.

Completaremos la implementación, basándonos en las clases abstractas anteriores, con la creación de la clase Iris que implemente Type<Iris>.

Clasificaremos todas las clases, excepciones, e interfaces en distintos paquetes para facilitar su gestión y mantenimiento.

Para probar el modelo está disponible en la página de Moodle una clase Java con distintos casos de prueba. Esta clase utiliza la librería JUnit. Puede ser necesario adaptarla, si por ejemplo se han usado unos nombres de atributos distintos a los del test, o las clases e interfaces están en otros paquetes.

Apartado 2. Lectura de datos (2 puntos)

Crearemos una clase CSVCaseReader, que permita leer casos desde ficheros CSV sencillos, especificando el archivo, el carácter separador, el juego de caracteres (charSet), y el tipo de problema (Type<T>)de datos.

Para poder asociar las instancias a su clasificación, crearemos una interfaz y la clase que la implementa.

```
public interface Case<T>{
  Instance<T> getInstance(); //devuelve la instance
  String getClassName(); //devuelve la clase en la que está clasificada la instancie
}
```

Nuestro lector de casos deberá tener un método que devuelva la lista de casos leídos. Esta clase (de Java) ha de generar excepciones en caso de errores de E/S, o de formato.

En cada linea del archivo CSV encontraremos los valores para cada atributo de una instancia y, como último campo de la línea, la clase en la que se ha clasificado dicha instancia en nuestra base de conocimiento. El archivo con la base de datos Iris está en formato UTF-8 y usa comas como separador. Es habitual que el archivo tenga una línea en blanco extra al final, nuestro lector no ha de fallar en ese caso, y es suficiente con ignorarla o dar por completada la lectura.

Comprobaremos que la clase CSVCaseReader funciona correctamente leyendo los datos de la base de conocimiento Iris, comprobando que no se generan errores y se leen todas las instancias.

Apartado 3. Estadística eficiente (2,5 puntos)

Por lo general, los sistemas expertos como el que deseamos implementar, han de poder procesar un gran número de datos, y se debe evitar la necesidad de almacenar dichos datos en memoria, o de tener que realizar varias pasadas para recalcular datos estadísticos.

La solución a nuestro problema consistirá en crear versiones de dichas funciones que funcionan de forma incremental, y que no requieren almacenar todos los datos anteriores para actualizar la variable estadística cuando se presenta un nuevo dato.

Crearemos las clases Mean, Variance y StdDev. Todas implementarán la siguiente interfaz:

```
public interface IncrementalFunction{
  void addValue(double value);
  double getResult();
}
```

El método addValue añade un nuevo dato, y actualizará el estado interno para tener en cuenta dicho valor.
El método getResult devuelve el valor actual de la variable estadística.

Para simplificar, se puede asumir que tanto la media como la varianza devolverán cero si no existen datos. En el caso de la varianza también será cero si solo hay un dato.

La implementación de la media ha de minimizar errores de cálculo, para ello se sugiere usar como estado interno dos variables, la suma de los valores observados y el número de valores.

Para el cálculo de la varianza de la muestra nos basaremos en un versión muy similar a la de Knuth en "The Art of Computer Programming", que a su vez se basa en el algoritmo de Welford.

```
def online_variance(data):
    n = 0
    mean = 0.0
    m2 = 0.0

    for x in data:
        n += 1
        delta = x - mean
        mean += delta/n
        m2 += delta*(x - mean)

    if n < 2:
        return 0  // en el algoritmo original se devuelve float('nan') en lugar de 0
    else:
        return m2 / (n - 1)
```

El estado de este algoritmo lo componen las variables n, mean y m2, y se actualiza en el bucle, que en nuestro caso será substituido por el método addValue(v). El método getResult() se corresponde con el "if" de la función anterior.

Hay dos formas sencillas de implementar la deviación típica. Creando un subclase de Variance, y sobreescribiendo el método getResult() para devolver la raíz cuadrada de la varianza, o mediante delegación.

Probaremos las clases implementadas creando una clase con un método main que añada una serie de datos a nuestras variables estadísticas, y compare la media, varianza, y desviación típica calculadas con los valores esperados. Es posible que por pequeños errores de cálculo los valores no sean exactos, por lo que podría ser útil redondearlos antes de realizar la comparación, o de imprimirlos.

Apartado 4. Sistema Experto (3 puntos)

Crearemos un sistema experto que implemente la siguiente interfaz

```
public interface Expert<T>{
  void addCase ( Case<T> aCase ) throws InvalidClassException;
  void addCase ( Instance<T> instace,  String aClass ) throws InvalidClassException;
  String testInstance (Instance<T> instance );
}
```

El sistema experto se inicializará con un objeto de tipo Type<T> para conocer las clases y atributos del tipo de datos, de forma que pueda crear el estado inicial del sistema experto.

La implementación del sistema experto clasificará cada instancia calculando la distancia cuadrática del vector formado por los valores de sus atributos, al vector que representa la media de dichos atributos para todas las instancias de una clase. El sistema seleccionará la clase cuya distancia sea menor. Para mantener la media de los atributos de una clase usaremos la clase Mean implementada en el apartado anterior.

Para ahorrar memoria, los métodos addCase() no añaden realmente el caso al sistema experto, si no que refleja ese nuevo conocimiento modificando las variables estadísticas correspondientes. Solo será necesario por lo tanto modificar el vector con las medias para cada atributo, para la clase indicada. Se producirá la excepción InvalidClassException si la clase no está entre las admitidas por el tipo de problema.

El método testInstance clasifica una instancia nueva indicando a qué clase pertenece. Este último método no debe modificar la base de conocimiento.

Para mayor eficiencia se recomienda utilizar Map para mantener la información estadística asociada a cada clase del problema.

Probaremos nuestro sistema experto inicializándolo con la mitad de casos de la base de datos Iris, y probando la clasificación con el resto de instancias. Antes de eso se ha de permutar de forma aleatoria la lista de casos, esto se puede conseguir con el método de la librería estándar Collections.shuffle(lista).

Crearemos dos variantes de las pruebas, una en la que las instancias nuevas no se añaden a la base de conocimiento, y otra en la que se añaden tras ser clasificadas, con la clasificación detectada por el sistema, aunque sea errónea.

Mostraremos un resumen de las instancias clasificadas correcta e incorrectamente en las dos variantes de las pruebas, indicando el número de instancias correctas y el porcentaje de fallos.

Opcional. Normalización (1 punto)

Crearemos una versión más avanzada del sistema experto del apartado anterior, heredando de la implementación del apartado 4. Esta nueva versión mantendrá, para cada atributo, su media y la desviación típica, independientemente de la clase en la que se ha clasificado una instancia, usando las clases creadas en el apartado 3.

La normalización se hará al calcular las distancias entre el vector correspondiente a la instancia, y el vector asociado a la clase. Cada valor de estos vectores se corresponde con un atributo. La normalización consistirá en calcular el número de desviaciones típicas por encima o debajo de la media para el atributo, con la siguiente fórmula:

$$v1 = (v0 - mean) / stdDev$$

Siendo v0 el valor original del atributo, y v1 el valor normalizado. mean y stdDev son los valores de la media y desviación típica de todas las instancias de la base de conocimiento para el atributo a normalizar.
En el caso de que stdDev sea cero, v1 tomará el valor cero.

La normalización hace que todos los atributos tengan la misma importancia, por lo tanto es necesario además poder ajustar el peso de cada atributo al calcular la distancia. Añadiremos a nuestro sistema un método para cambiar dichos pesos.

Se han de sobreescribir o añadir los métodos necesarios para mantener las nuevas variables estadísticas, y modificar como se calcula la distancia, normalizando los vectores antes de dicho cálculo, y usando el peso del atributo a la hora de calcular la distancia total.

Se probará el nuevo sistema experto con distintos pesos, y se compararán los resultados con el sistema anterior.

Nota: En el caso de la base de datos Iris, normalizar sin ajustar pesos puede producir peores resultados, ya que pasan a tener más importancia atributos que antes tenían una desviación típica baja, y que influían poco en el resultado.
Esto se puede ver fácilmente si se fijan a cero todos los pesos menos uno, para comprobar con qué atributo se clasifica mejor. Los siguientes pesos por ejemplo suelen producir un buen resultado para esta base de datos: {0.1, 0.025, 0.55, 0.325}

Normas de Entrega:

- Se deberá entregar
  - un directorio **src** con todo el código Java, incluyendo también el código de las pruebas
  - un directorio doc con la documentación **Javadoc** generada
  - un directorio txt con todos los archivos utilizados y generados en las pruebas
  - un documento en **PDF** con una breve justificación de las decisiones que se hayan tomado en el desarrollo de la práctica, los problemas principales que se han abordado y cómo se han resuelto, así como los problemas pendientes de resolver
  - el documento PDF incluirá el **diagrama de clases** del sistema
- Se debe entregar un único fichero ZIP con todo lo solicitado, que deberá llamarse de la siguiente manera: GR<numero_grupo>_<nombre_estudiantes>.zip. Por ejemplo Marisa y Pedro, del grupo 2261, entregarían el fichero: GR2261_MarisaPedro.zip.