EVALUACIÓN CONTINUA
Examen final ADSOF 2020/21 – Convocatoria Ordinaria
Contesta cada ejercicio en una hoja separada

Ejercicio 1 (2.75 puntos)

El grupo PSA (Peugeot société anonyme) nos ha contactado para gestionar el software de sus fábricas de vehículos. Se trata de un fabricante multimarca que fabrica coches para Peugeot y Opel. El grupo quiere aprovechar el mismo software para las fábricas de las diferentes marcas por lo que el diseño deberá ser flexible.

Las fábricas de cada marca producen los diferentes coches. Cada coche está formado por un motor, un chasis y una carrocería. Todos los vehículos tienen los mismos motores independientemente de la marca. Cada motor tiene una cilindrada y el tipo de combustión (diésel o gasolina). Hay un chasis para todos los coches bajo la marca Peugeot y otro diferente para todos los coches bajo la marca Opel. El chasis debe tener un número de bastidor que se generará automáticamente justo antes de salir de la fábrica. Por su parte, la carrocería depende del modelo específico de cada marca, pero sólo existen dos tipos de carrocería que puedan emplear las marcas: ancha y estrecha. De las carrocerías se debe conocer además el número de puertas para poder ofrecer el acabado final.

Se pide:

a) Realizar el diagrama de clases que represente la parte del diseño detallada arriba. En particular asegúrate de añadir métodos que garanticen que las fábricas pueden crear coches de cada marca sin posibilidad de mezclar elementos de marcas distintas. **[2,25 puntos]**

b) ¿Qué patrón o patrones de diseño has utilizado? Indica los roles del patrón en tu diseño. **[0.5 puntos]**

Nota: No incluyas en este diagrama constructores, *getters* ni *setters*.

Ejercicio 2 (2,75 puntos)

Se quiere desarrollar un conjunto de clases Java para gestionar localizaciones en un mapa. Una localización puede ser un punto o una región, y una región está compuesta de una o más localizaciones (puntos u otras regiones) "hijas". Toda localización está caracterizada por: un nombre, dos coordenadas (latitud y longitud) y una posible región "padre" que la contenga.

Las coordenadas asociadas a una región se corresponden con los valores medios de las latitudes y longitudes de las localizaciones contenidas dentro de la región. De este modo, cada vez que se añade una localización a una región, las coordenadas de esta última han de actualizarse, así como las de las posibles regiones contenedoras en niveles superiores.

Por otra parte, se necesitan métodos para calcular la distancia entre localizaciones, esto es, entre dos puntos, entre dos regiones, o entre un punto y una región. En el caso de regiones, se tendrán en cuenta las distancias entre todos sus puntos, y se elegirá la que sea más pequeña.

El siguiente código muestra un ejemplo de programa de referencia:

```
MapRegion r1 = new MapRegion("España", new Coordinates(40.4165, -3.70256));
MapRegion r2 = new MapRegion("Comunidad de Madrid");
MapRegion r3 = new MapRegion("Cataluña");
MapPoint p1  = new MapPoint("Madrid",     new Coordinates(40.4165,  -3.70256) );
MapPoint p2  = new MapPoint("Alcobendas", new Coordinates(40.54746, -3.64197));
MapPoint p3  = new MapPoint("Getafe",     new Coordinates(40.30571, -3.73295));
MapPoint p4  = new MapPoint("Barcelona",  new Coordinates(41.38879,  2.15899));

r1.addLocation(r2).addLocation(r3);
r2.addLocation(p1).addLocation(p2).addLocation(p3);
r3.addLocation(p4);

System.out.println("Coordenadas de " +  r1.getName()  + ": "  + r1.getCoordinates());
System.out.println("Coordenadas de " +  r2.getName()  + ": "  + r2.getCoordinates());
System.out.println("Distancia entre " + p1.getName()  + " y " + p2.getName() + ": " + p1.distance(p2));
System.out.println("Distancia entre " + r2.getName()  + " y " + r3.getName() + ": " + r2.distance(r3));
```

Salida esperada:

```
Coordenadas de España: (40.90600666666667,-0.7667516666666667)
Coordenadas de Comunidad de Madrid: (40.42322333333333,-3.6924933333333336)
Distancia entre Madrid y Alcobendas: 15.437436392379794
Distancia entre Comunidad de Madrid y Cataluña: 495.85252574538794
```

Se pide:

(a) **[2,5 puntos]** Utilizando un patrón de diseño, completa el programa Java para obtener la salida de arriba.

(b) **[0,25 puntos]** ¿Qué patrón has utilizado? Identifica las clases que corresponden a los elementos de dicho patrón.

Asumir la existencia de:

- Una clase `Coordinates` para representar una posición geográfica, con un constructor `Coordinates(double, double)`, y métodos `double getLatitude()`, `double getLongitude()` y `String toString()`.
- Una clase `HaversineDistance` para calcular la distancia entre dos posiciones y la mínima distancia entre dos conjuntos de posiciones.

```
public class HaversineDistance {
   public static double distance(Coordinates position1, Coordinates position2) { ... }
   public static double distance(Set<Coordinates> positions1, Set<Coordinates> positions2) { ... }
}
```

Ejercicio 3 (3 puntos)

Se quiere desarrollar una clase `StandingTable` para la gestión de resultados de competiciones en juegos o deportes. La clase será lo más reutilizable posible. En particular, asumirá que los jugadores involucrados son conformes a la interfaz `IRated`, que describe cómo obtener y actualizar los puntos de cada jugador en la competición. Debemos facilitar la implementación de esta interfaz lo máximo posible, así que debes completarla si es necesario.

```
/** Completar como sea conveniente para facilitar su implementación, en caso necesario **/
public interface IRated {
	public double getRating();
	public void updateRating(double d);
}
```

Además, podremos configurar los puntos obtenidos por cada jugador cuando gana (`win`) o empata (`tie`) con objetos conformes a la siguiente interfaz:

```
public interface IResultConfiguration {
	public void updateRatingWin(IRated r1, IRated r2); // actualizar puntos si r1 gana a r2
	public void updateRatingTie(IRated r1, IRated r2); // actualizar puntos cuando empatan
}
```

Por ejemplo, en el ajedrez, un empate son 0.5 puntos para cada jugador, mientras que el jugador que gana recibe 1 punto.

El siguiente programa ejemplifica el uso de `StandingTable` para gestionar un torneo de ajedrez (`chess`). El constructor de la clase recibe por un lado la configuración del juego (`ChessResult`) y una lista de jugadores. Los métodos `win` y `tie` se utilizan para indicar que el primer jugador gana al segundo (`win`), o bien que empatan (`tie`). Si se pasa a estos métodos un jugador que no estaba en la lista inicial, ha de lanzarse una excepción. Finalmente, al imprimir el objeto se muestra el ranking de los jugadores, ordenado de mayor a menor puntuación (en caso de empate, por orden alfabético), así como todos los resultados de los enfrentamientos que se hayan producido, donde los jugadores se presentan por orden de inserción.

```
public class Main {
  public static void main(String...args) {
	List<ChessPlayer> players = List.of(new ChessPlayer("Magnus Carlsen"),
	                                    new ChessPlayer("David Anton"),
	                                    new ChessPlayer("Daniil Dubov"));
	StandingTable rc = new StandingTable(new ChessResult(), players);
	try {
		rc.win(players.get(0), players.get(2)); // Carlsen gana a Dubov
		rc.tie(players.get(2), players.get(1)); // Dubov y Anton empatan
		rc.win(players.get(1), new ChessPlayer("Garry Kasparov")); // Garry es desconocido
	} catch (UnknownRatedException e) {
		System.err.println(e);
	}
	System.out.println(rc);
  }
}
```

Se pide: Usando principios de orientación a objetos, desarrollar todo el código necesario para obtener el resultado de más abajo.

```
ratings.exceptions.UnknownRatedException: Unknown Garry Kasparov (0.0)
Ranking: [Magnus Carlsen (1.0), Daniil Dubov (0.5), David Anton (0.5)]
Results: {Magnus Carlsen (1.0)={WIN=[Daniil Dubov (0.5)], LOSE=[], TIE=[]}, David Anton
(0.5)={WIN=[], LOSE=[], TIE=[Daniil Dubov (0.5)]}, Daniil Dubov (0.5)={WIN=[], LOSE=[Magnus
Carlsen (1.0)], TIE=[David Anton (0.5)]}}
```

Ejercicio 4 (1.5 puntos)

Se quiere desarrollar una clase genérica `MultiOrderedList` que será compatible con `List`, pero mantendrá sus elementos ordenados. El criterio de ordenación será el orden natural de los elementos, pero además, se podrán dar criterios adicionales – mediante el método `addOrder` – que se usarán sucesivamente en caso de que los elementos vayan siendo iguales en cada comparación. Cada criterio de ordenación debe devolver 0 si los objetos son iguales, un número positivo si el primer objeto es mayor que el segundo, y un número negativo si el primer objeto es menor que el segundo.

El siguiente programa ejemplifica el uso de `MultiOrderedList` para ordenar una lista de objetos de tipo `Person`. La primera vez que se imprime se usa el orden natural; la segunda vez el orden natural y la edad; y la tercera vez, además el número de hijos.

```
class Person implements Comparable<Person>{ // Una clase de ejemplo
  private String name;
  private int age;
  private int children;
  public Person (String name, int age, int ch) {
      this.name = name;
      this.age = age;
      this.children = ch;
  }
  @Override public int compareTo(Person o) { return this.name.compareTo(o.name.toString()); }
  @Override public String toString() { return "("+name+"; age "+age+"; "+children+" children)"; }
  public int getAge() { return this.age; }
  public int getChildren() { return this.children; }
}

public class OrderedListMain {
  public static void main(String[] args) {
      MultiOrderedList<Person> list = new MultiOrderedList<>( new Person("Sara", 8, 1),
                                                              new Person("Don", 65, 2),
                                                              new Person("Mia", 28, 0),
                                                              new Person("Don", 43, 3),
                                                              new Person("Don", 43, 0));
      List<Person> lst = list;  // Compatible con List
      System.out.println(lst);  // orden natural (alfabético por nombre)
      list.addOrder((p1, p2) -> p1.getAge()-p2.getAge());  // … además por edad
      System.out.println(list);
      list.addOrder((p1, p2) -> p1.getChildren()-p2.getChildren());  // …además por nº hijos
      System.out.println(list);
      // Realiza un diseño general para permitir también criterios de orden como el siguiente
      list.addOrder((Object o1, Object o2) -> o1.hashCode()-o2.hashCode());
  }
}
```

Se pide: Usando principios de orientación a objetos, desarrolla la clase `MultiOrderedList` para obtener el resultado de más abajo.

```
[(Don; age 65; 2 children), (Don; age 43; 3 children), (Don; age 43; 0 children), (Mia; age 28; 0 children), (Sara; age 8; 1 children)]
[(Don; age 43; 3 children), (Don; age 43; 0 children), (Don; age 65; 2 children), (Mia; age 28; 0 children), (Sara; age 8; 1 children)]
[(Don; age 43; 0 children), (Don; age 43; 3 children), (Don; age 65; 2 children), (Mia; age 28; 0 children), (Sara; age 8; 1 children)]
```