Análisis y Diseño de Software (revisión 26 mayo 2016)

Apartado 1. (2,5 puntos)

Tienes que diseñar un programa para que los nutricionistas **creen dietas y analicen su aporte de calorías y su contenido porcentual de nutrientes** de interés, como proteínas, grasas, sal, gluten, hierro, fósforo, etc.

El concepto de *dieta* es muy amplio. Una dieta puede ser una simple *receta*, o referirse a una combinación de dietas de cualquier tipo. Cada dieta tiene un identificador propio y único, generado automáticamente por el programa al crearla. Las recetas se crean con su descripción en forma de texto y sus *ingredientes* por ración unipersonal. Cada ingrediente de una receta indica el peso en gramos de un *alimento básico* utilizado en la receta. Por alimento básico entendemos cosas como leche entera, pechuga de pollo, zumo de naranja, etc. Por ejemplo, una receta puede constar de estos 2 ingredientes: 200 g de pechuga de pollo y 100 g de zumo de naranja. El aporte de calorías de una receta se calcula sumando el de sus ingredientes, calculado a su vez a partir de los alimentos básicos que forman la receta, tal y como se indica a continuación.

Para cada alimento básico se debe almacenar (además de su nombre) el porcentaje de su peso que corresponde a los diversos *nutrientes* que contiene. Por ejemplo, la pechuga de pollo es 1% grasas, 0% hidratos, 15% proteínas, 0.07% sal, 0.2% fósforo, etc. y el zumo de naranja es 0% grasas, 10% hidratos, 0% proteínas, etc.

Cada nutriente se define mediante dos datos, su nombre y su aporte calórico en kilocalorías por cada gramo de nutriente (kcal/g). Así por ejemplo, las proteínas y los hidratos aportan 4 kcal/g., las grasas 9 kcal/g, y la sal y el fósforo 0 kcal/g. Por lo tanto, 100 g de zumo de naranja aportan 10 g de hidratos (10%) que aportan 40 kcal, y 200 gramos de pechuga de pollo que aportan 2 g de grasas (1%) y 30 g de proteínas (15%), es decir, 18 kcal y 120 kcal, respectivamente. Luego, el aporte del pollo son 138 kcal, y el total de la receta ejemplo sería 178 kcal.

Es decir, el aporte de calorías de una receta se calcula sumando el aporte calórico de sus ingredientes, que a su vez se calcula con el peso de cada alimento básico y la composición de éste en términos de nutrientes (que son en última instancia quienes determinan el origen de las calorías por gramo).

Por otro lado, hemos de calcular el porcentaje de peso de una receta que corresponde a cualquier nutriente que nos interese. En la receta ejemplo, el porcentaje de proteínas es 10% (30g. de proteína en el pollo en los 300g. totales de la receta) y el de hidratos es 3.3% (10g. del zumo en los 300g. totales de la receta). Esto nos debe permitir seleccionar recetas con altos o bajos contenidos en nutrientes de interés especial como sal, gluten, hierro, grasas, etc.

Se pide:

(a) Representar el diagrama de clases en UML para el programa descrito arriba, sin incluir los métodos *getters* o *setters* ni los constructores, ni las clases relacionadas con una posible interfaz de usuario.
[1,6 puntos] **Ver diagrama solución en siguiente página**

(b) Para cada uno de los siguientes requisitos del programa, describe el método correspondiente (con su nombre, argumentos, tipo de resultado, y demás características) e indica la clase más apropiada para contenerlo:

1. Calcular el aporte de calorías de una dieta [0,2 puntos]

   En clase abstracta Dieta, método abstracto: `+ calorias() : Double`
   con implementaciones en subclases Receta y DietaCompuesta

2. Añadir un nuevo ingrediente [0,2 puntos]

   Implementado en clase Receta, que guarda una colección de ingredientes:
   `+ añadir(ing : Ingrediente) : boolean`
   que retorna true si y sólo si se ha podido añadir correctamente.

(c) Implementa, en pseudocódigo o código java, el método para obtener las dietas con menos que un porcentaje dado de un nutriente concreto. [0,2 puntos]

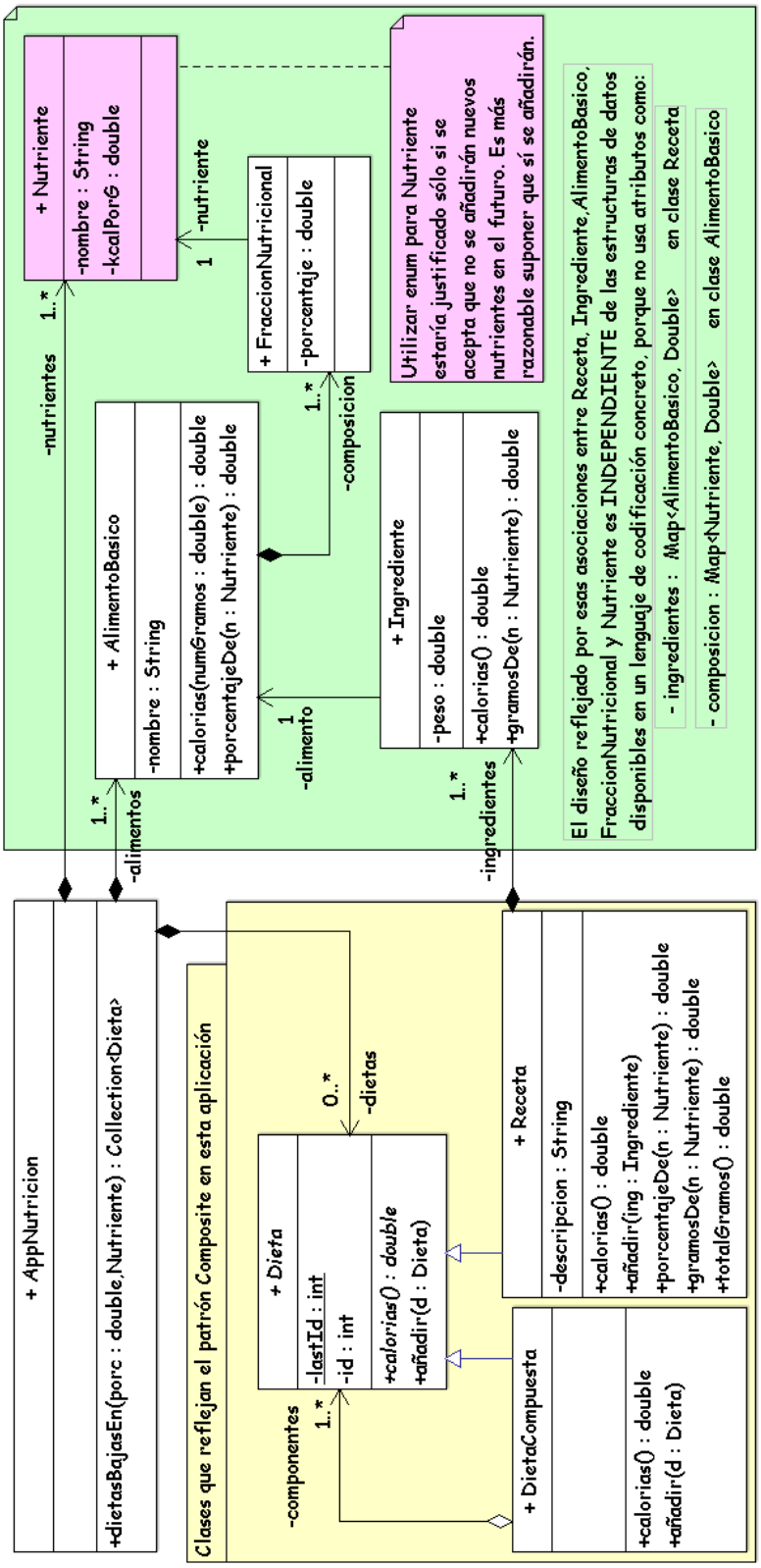
```
En clase AppNutrición:
método dietasBajaEn(Nutriente nutrienteBuscado, Doble porcentajeUmbral):
resultado = crear lista varia de dietas;
para cada receta r de la aplicacion:
   si (100 * r.gramosDe(nutrienteBuscado)) / r.totalGramos()) < porcentajeUmbral)
     resultado.add(r);
return resultado;  // falta desglosar los métodos gramosDe y totalGramos en Receta.
```

Análisis y Diseño de Software (revisión 26 mayo 2016)

(d) ¿Qué patrón de diseño has utilizado y cómo participan en él las clases de tu diseño? [0,3 puntos]

Se utiliza un patrón Composite, donde la clase Receta es la única hoja (Leaf), la clase DietaCompuesta es la clase del compuesto (Composite) y la clase abstracta Dieta es el componente (Component).

Nota: No es necesario detallar las fórmulas específicas para los diversos cálculos. En el enunciado se han mencionado solamente para ilustrar el procedimiento general de cálculo.



Análisis y Diseño de Software (revisión 26 mayo 2016)

Apartado 2. (2,5 puntos).

Tienes que diseñar e implementar parte de un editor gráfico dedicada a la manipulación de colores. Un *color RGB* está definido por tres valores enteros correspondientes cada uno a la intensidad de una de las tres luces básicas: roja, verde y azul.

El editor podrá necesitar de la creación de diversos tipos de colores. En concreto, un *color translucido* es un color RGB que además tiene un factor *alfa* que describe su nivel de opacidad (entre 0.0 para transparente y 1.0 para totalmente opaco, que será el valor por defecto en caso de no indicarse ninguno). Un *color gris* es un color RGB que se define con el mismo valor único para las tres luces (roja, verde y azul) y un grado de grises que es otro valor entero que servirá para que ese color gris varíe de manera más brusca o lenta a la hora de aclararlo.

En general, todos los colores se pueden aclarar disminuyendo en 1 sus tres niveles de luces (sin dejar que ninguno llegue a ser negativo). Además los colores translucidos al aclararse disminuyen en un 10% su nivel de opacidad. Y los colores grises, al aclararse no disminuyen sus luces en 1 sino en una cantidad igual a su grado de grises.

Cada color tendrá un identificador generado automáticamente formado por una etiqueta que indica el tipo de color seguida de un número entero correspondiente al orden de creación (independientemente de cuál sea su clase de color. El sistema debe contabilizar cuántos colores RGB se han generado.

Se pide:
Desarrollar las clases Java que cumplan la especificación anterior con un buen diseño (es decir, fácil de ampliar y modificar, a la vez que la implementación sea segura) y cuya ejecución con el siguiente programa produzca la salida indicada abajo.

Nota: en este caso se considera aceptable suponer que todas las clases directamente relacionadas con la manipulación de colores irán en un mismo paquete `editorgrafico.colores` aisladas del resto del software del editor gráfico.

```
package pruebas;
import editorgrafico.colores.*;

public class Ej2Colores {
  public static void main(String[] args) {
     ColorRGB[] colores = { new ColorTranslucido(200, 50, 20), // nivel de opacidad 1.0
                            new ColorTranslucido(100, 0, 100, 0.5),
                            new Gris(128, 4)  };
     for (ColorRGB a : colores) {  // ERRATA corregida antes era:    objetos) {
          System.out.print( a.id() + " = " + a  );
          a.aclara();
          System.out.println( "   que al aclararlo queda " + a  );
     }
     System.out.println( "Núm. colores generados: " + ColorRGB.numColores() );
                         // ERRATA corregida antes era: Color.numColores() );
  }
}
```

Salida esperada:

```
TR-1 = 200:50:20:1.0 que al aclararlo queda 199:49:19:0.9
TR-2 = 100:0:100:0.5 que al aclararlo queda 99:0:99:0.45
GR-3 = 128:128:128 que al aclararlo queda 124:124:124
Núm. colores generados: 3
```

Análisis y Diseño de Software (revisión 26 mayo 2016)

Solución:

```
package editorgrafico.colores;

public abstract class ColorRGB {
	private static int numObjetos = 0;
	protected int rojo, verde, azul;
	 //usamos protected pq se dice EXPLICITAMENTE: paquete exclusivo para estas clases
	private int id;
	public ColorRGB(int r, int v, int a) {
		rojo = r; verde = v; azul = a; numObjetos++; id = numObjetos;
	}
	public static  int numColores() { return  numObjetos; }
	protected abstract String idEtiqueta();
	public void aclara() {
		if (rojo > 0) rojo--;
		if (verde > 0) verde--;
		if (azul > 0) azul--;
	}
	public String id() {  return "" + idEtiqueta() + id; }
	@Override public String toString() { return "" + rojo + ":" + verde + ":" + azul; }
}
//---------------------------
package editorgrafico.colores;

public class ColorTranslucido extends ColorRGB {
	private double opacidad;
	public ColorTranslucido(int rojo, int verde, int azul) {
		this(rojo,verde,azul, 1.0);
	}
	public ColorTranslucido(int rojo, int verde, int azul, double alpha) {
		super(rojo,verde,azul); this.opacidad = alpha;
	}
	@Override protected String idEtiqueta() { return "TR-" ; }
	@Override public String toString() { return super.toString() + ":" + opacidad; }
	@Override public void aclara() {
		super.aclara();
		opacidad *= 0.9; // disminuir en un 10%
	}
}
//---------------------
package editorgrafico.colores;

public class Gris extends ColorRGB {
	private int gradoGrises;
	public Gris(int x, int g) {
		super(x,x,x);
		gradoGrises = g;
	}
	@Override protected String idEtiqueta() { return "GR-" ; }
	@Override public void aclara() {
		rojo = Math.max(rojo - gradoGrises, 0);
		verde = Math.max(verde - gradoGrises, 0);
		azul = Math.max(azul - gradoGrises, 0);
	}

}
```

Análisis y Diseño de Software (revisión 26 mayo 2016)

Apartado 3. (2,5 puntos)

Se quiere implementar una agenda telefónica, que permita asignar números de teléfono a nombres de personas. Mientras que las personas se representarán como un *String*, los números de teléfono tendrán su propia clase, que debe guardar el prefijo internacional del país correspondiente, así como el número en sí. Por simplicidad, la agenda guardará como máximo un número de teléfono por persona, pero debe indicar un error si el mismo número de teléfono se asigna a dos personas distintas.

Se pide: Completa el siguiente programa con las instrucciones que faltan en los huecos, así como con las clases, enumerados y excepciones necesarias para que produzca la salida de más abajo.

```
public class Apartado3 {
        public static void main (String... args) {
                Agenda misTelefonos = new Agenda();
                _try__ {        // hueco 1. En continua no hay hueco
                        misTelefonos.anyadeTelefono("Lewis Hamilton",  new Telefono(Pais.ReinoUnido, "7654321")).
                                     anyadeTelefono("Fernando Alonso", new Telefono(Pais.España,     "1234567")).
                                     anyadeTelefono("Carlos Sainz",    new Telefono(Pais.España,     "1234567"));

                } _catch____ (ExcepcionNumeroRepetido e) { // hueco 2. En continua no hay hueco
                        System.out.println("Número repetido: "+e.getNumero());
                        return;
                } _finally__ {  // hueco 3. En continua no se usa finally ni aparece el return anterior
                        System.out.println("Mi agenda = "+misTelefonos);
                }
        }
}
```

El resultado del programa anterior debe ser:

```
Número repetido: (0034) 1234567
Mi agenda = {Fernando Alonso=(0034) 1234567, Lewis Hamilton=(0044) 7654321}
```

Donde la agenda se debe imprimir tanto si se produce un error por número repetido como si no hay número repetido. Como ves, la agenda presenta los teléfonos por orden alfabético.

Análisis y Diseño de Software (revisión 26 mayo 2016)

```
class Agenda{
	private final Map<String, Telefono> contactos= new TreeMap<>();
	public Agenda anyadeTelefono(String p, Telefono telefono) throws ExcepcionNumeroRepetido {
		if (!contactos.containsValue(telefono)){
			contactos.put(p, telefono);
			return this;
		}
		else throw new ExcepcionNumeroRepetido(telefono);
	}
	@Override
	public String toString(){
		return contactos.toString();
	}
}
class Telefono{
	private final Pais pais;
	private final String numero;
	public Telefono(Pais pais, String numero) {
		this.pais = pais;
		this.numero = numero;
	}
	@Override
	public boolean equals(Object o){
		if (o!=null && getClass()==o.getClass()){
			Telefono t=(Telefono) o;
			return pais==t.pais && numero.equals(t.numero);
		}
		else return false;
	}
	@Override
	public int hashCode(){
		return pais.hashCode()*13+numero.hashCode();
	}
	@Override
	public String toString(){
		return "("+pais.getPrefijo()+") "+numero;
	}
}
enum Pais{
	ReinoUnido("0044"), España("0034");
	private final String prefijo;
	Pais(String prefijo) {
		this.prefijo = prefijo;
	}
	public String getPrefijo(){
		return prefijo;
	}
}
class ExcepcionNumeroRepetido extends Exception{
	private final Telefono numero;
	public ExcepcionNumeroRepetido(Telefono numero) {
		this.numero = numero;
	}
	public Telefono getNumero(){
		return numero;
	}
}
```

Análisis y Diseño de Software (revisión 26 mayo 2016)

Apartado 4. (2,5 puntos)

Se quiere implementar una clase *Almacen* que almacene datos de cualquier tipo. La clase debe descartar elementos repetidos y, por defecto, mantenerlos en el *orden natural* de la clase que se almacene. Adicionalmente, se debe poder configurar un orden distinto (que tiene preferencia sobre el orden natural), mediante un criterio de comparación, que se da en el método *conCriterio()*.

Debe ser posible crear un número arbitrario de filtros, bien a partir de un almacén, o de otros filtros. Los filtros se implementan con la clase genérica *Filtro*, y se añaden con el método *crear*, que debe admitir otros filtros y almacenes, así como la expresión del filtro.

Como ejemplo, el siguiente programa:

```
public class Colecciones {
  public static void main(String... args) {
    Almacen<String> palabras = new Almacen<String>("este","examen","no","es","facil","no","esta","chupado");
    Filtro<String>       palabrasFiltradas    = Filtro.crear (palabras, p -> p.length()>2);
    IAlmacenable<String> palabrasMasFiltradas = Filtro.crear (palabrasFiltradas,
                                                              p -> p.startsWith("e") || p.startsWith("c"));

    System.out.println("Almacen original              : "+palabras.obtener());
    System.out.println("Almacen por orden de longitud : "+palabras.conCriterio((p,q) -> p.length() - q.length()));
    System.out.println("Resultado del primer  filtro  : "+palabrasFiltradas.obtener());
    System.out.println("Resultado de los dos filtros  : "+palabrasMasFiltradas.obtener());
  }
}
```

Debe producir la siguiente salida:

```
Almacen original              : [chupado, es, esta, este, examen, facil, no]
Almacen por orden de longitud : [es, esta, facil, examen, chupado]
Resultado del primer  filtro  : [esta, facil, examen, chupado]
Resultado de los dos filtros  : [esta, examen, chupado]
```

Como ves, el almacén original mantiene las palabras ordenadas por orden alfabético (el orden natural de los *Strings*), y no aparecen repetidas. Al añadir el criterio de ordenación con el método "conCriterio()", el almacén pasa a mantener las palabras por orden de longitud, y no aparecen palabras con la misma longitud. El primer filtro aplicado al almacén elimina las de longitud menor o igual a 2, mientras que el segundo filtro elimina las que no empiezan por "e" o "c". Fíjate en que los filtros siguen manteniendo las palabras por longitud.

Se pide:

(a) Utilizando patrones de diseño, y Streams (siempre que sea posible), completar el programa anterior con las clases e interfaces necesarias. [2 puntos]

(b) Indica qué patrón o patrones de diseño has utilizado, y qué roles en el patrón juegan las clases e interfaces que has implementado. [0.5 puntos]

Solución: Patrón Proxy, donde Filter es el proxy, IAlmacenable es el Sujeto y Almacen el sujeto real.
Otras opciones descartadas:

- El patrón "Factory Method" no se aplica, en este caso conocemos la clase que se crea (Filtro), y además es un método estático.
- Abstract Factory: No es un kit, solo se crea un tipo de producto.
- Iterator: Se usa el patrón, como en cualquier programa que use colecciones, pero no se está diseñando con dicho patrón.
- Singleton: Filtro.crear crea un nuevo filtro en cada llamada, por lo tanto no es Singleton
- Composite: La estructura de clases es similar, pero Filtro solo se relaciona con un IAlmacenable, mientras que en Composite tenemos agregaciones.

Análisis y Diseño de Software (revisión 26 mayo 2016)

```
interface IAlmacenable<T> {
	Collection<T> obtener();
}
class Almacen<T extends Comparable<? super T>> implements IAlmacenable<T>{
	private Set<T> elementos;
	public Almacen(T ... l) {
		elementos= new TreeSet<>(Arrays.asList(l));
	}
	//conCriterio solo es necesario en No Continua
	public Collection<T> conCriterio(Comparator<? super T> criterio){
		Set<T> datos=elementos;
		elementos= new TreeSet<>(criterio);
		elementos.addAll(datos);
		return elementos;
	}
	@Override
	public Collection<T> obtener() {
		return elementos; //otra opción: Collections.unmodifiableCollection(elementos);
	}
}
class Filtro<T> implements IAlmacenable<T>{
	private final IAlmacenable<T> datos;
	private final Predicate<T> predicado;
	private Filtro(IAlmacenable<T> datos, Predicate<T> predicado){
		this.datos=datos;
		this.predicado=predicado;
	}
	@Override
	public Collection<T> obtener() {
		return datos.obtener().stream().filter(predicado).collect(Collectors.toList());
	}
	public static <T> Filtro<T> crear(IAlmacenable<T> datos, Predicate<T> predicado){
		return new Filtro<>(datos, predicado);
	}
}
```