CONVOCATORIA EXTRAORDINARIA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas

Apartado 1. (2,5 puntos)

Tienes que diseñar una aplicación para la gestión de los modelos de teléfono móvil que vende una tienda. La tienda ha de poder configurar algunas características de la cámara y batería de los modelos de teléfono. Los teléfonos son de dos marcas: LG y HTC. Los teléfonos de ambas marcas llevan exactamente una cámara y una batería, cuya marca debe coincidir con la del teléfono. Es decir, un teléfono LG lleva una batería y cámara LG, y un teléfono HTC lleva cámara y batería HTC. Tu diseño debe impedir que se puedan crear configuraciones de teléfono con marcas desiguales (desde otras partes de la aplicación), para lo que debes añadir el soporte adecuado para la creación de teléfonos con configuración correcta de cámara y batería.

Los teléfonos, cámaras y baterías, tienen una garantía (en años) y un precio base (en euros). Los teléfonos deben configurarse con el sistema operativo de serie. El precio del teléfono se calcula como la suma de su precio base, el de la cámara y el de la batería que contiene, más un impuesto estándar del 21.0%. Además, los teléfonos HTC tienen un impuesto adicional del 5%. Las cámaras tienen un número de pixels, y pueden ser duales o no (aunque las cámaras HTC nunca son duales). Las baterías tienen una duración estimada (en horas), y las baterías HTC deben configurarse con el número de ciclos de carga soportados.

Se pide:

(a) Representar el diagrama de clases en UML para el fragmento de aplicación descrito arriba, sin incluir *getters*, *setters*, ni clases relacionadas con una posible interfaz de usuario. No olvides añadir todos los elementos necesarios (clases, interfaces, enumerados, atributos, métodos, etc) que aseguren una creación correcta de configuraciones de teléfono.
Se valorará específicamente la calidad, extensibilidad y concisión del diseño, así como el uso correcto de nociones de orientación a objetos y de UML.

(b) Añade los métodos/constructores y otros elementos necesarios para:

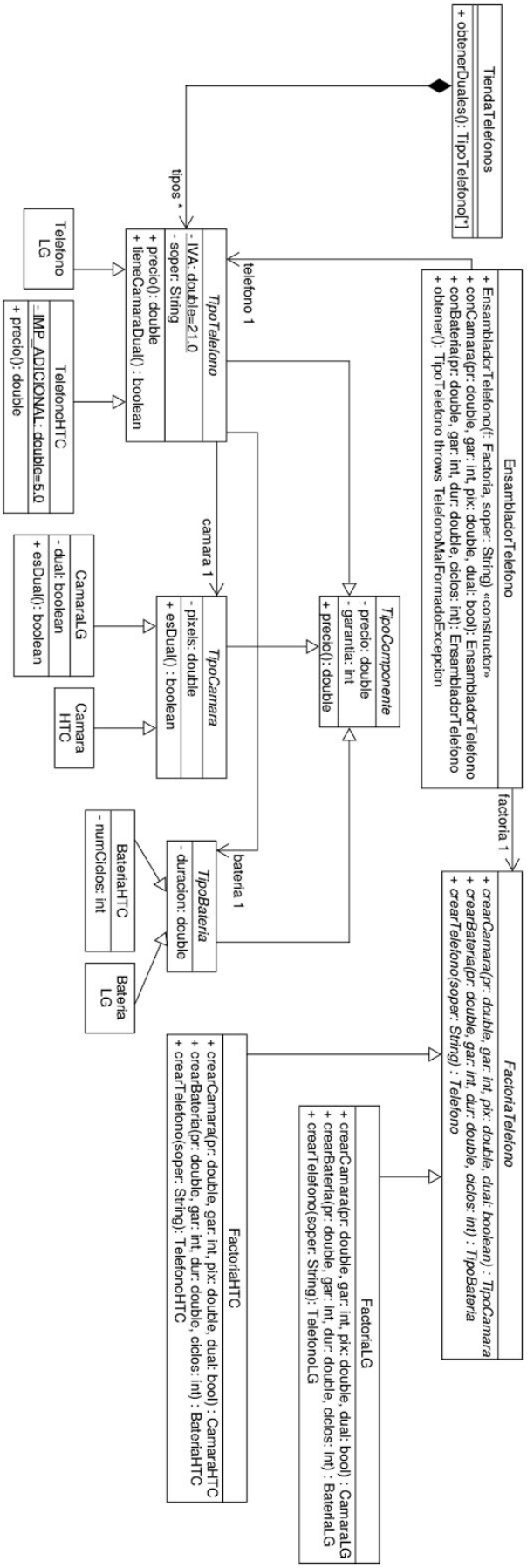
1. Construir los teléfonos.
   (clase EnsambladorTelefono y sus métodos, factorías asbtractas y concretas y sus métodos).
2. Calcular el precio de un teléfono.
   (TipoTelefono.precio(), TelefonoHTC.precio(), y TipoComponente.precio())
3. Obtener todos los modelos de teléfono con cámara dual.
   (TiendaTelefono.obtenerDuales())

(c) ¿Qué patrones de diseño has utilizado y qué roles juegan en ellos los distintos elementos de tu diseño?
Se ha utilizado el patrón AbstractFactory. Hay dos familias de productos: HTC y LG. Por tanto, se ha creado una factoría abstracta (FactoriaTelefono) y dos factorías concretas (FactoriaLG y FactoriaHTC). Cada factoría crea tres tipos de producto: tipos de teléfonos, tipos de cámara y tipos de batería. Por tanto, existen tres productos abstractos (TipoTelefono, TipoCamara y TipoBatería), y dos productos concretos de cada tipo. De manera adicional, para evitar la mezcla de marcas, se ha creado una clase EnsambladorTelefono, que se configura con una factoría y el SO del teléfono, y permite añadirle el tipo de cámara y el tipo de batería.

CONVOCATORIA EXTRAORDINARIA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas



CONVOCATORIA EXTRAORDINARIA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas

Apartado 2. (2,75 puntos)

Se desea gestionar el aparcamiento regulado de vehículos calculando su precio. Cada vehículo, ya sea moto o coche, se identifica mediante su matrícula (inalterable desde la creación del vehículo). El precio del aparcamiento se calcula inicialmente a partir del número de minutos que se solicita aparcar y el precio base por minuto para cada tipo de vehículo (0,50 €/min. para coches y 0,25€/min. para motos). Además, en el caso de los coches, su nivel de emisiones contaminantes determina un % de ajuste (descuento o recargo) en dicho cálculo inicial del precio de aparcamiento. Los niveles de emisión son: ECO, BAJO, MEDIO, ALTO, y sus respectivos porcentajes de ajuste son –50%, –25%, 0%, y +25%.

Para un coche, el cálculo del precio solamente tiene sentido si el tiempo de aparcamiento que se solicita, acumulado a todos los tiempos de aparcamientos previamente consumidos por ese mismo coche, no excede el tiempo máximo acumulado permitido. Dicho máximo se fija, de manera general, en 120 minutos para cualquier coche, pero hay coches con permisos especiales (ver ejemplo abajo). Los coches con permiso especial no tienen límite de tiempo de aparcamiento. En caso de excederse el límite de tiempo acumulado máximo permitido de aparcamiento, el cálculo del precio debe lanzar una excepción. En cualquier caso, además se desea tener constancia el total de precios recaudados por aparcamientos válidos (sin exceso de tiempo máximo) para el conjunto de todos los coches.

Las motos no tienen tiempo límite de aparcamiento ni debemos calcular su total de recaudación.

No es necesario preocuparse de restablecer a cero los acumuladores de tiempo y recaudación.

Se pide: **Codificar todo lo necesario** para que, una vez completadas correctamente las **líneas 1, 2 y 3** marcadas abajo y relacionadas con el manejo de excepciones, el programa ejemplo produzca la salida de dada abajo.
Nota: Acuérdate de incluir **en tu hoja de respuesta** el contenido de las **líneas 1, 2 y 3** marcadas abajo.

```
public class Main {
  public static void main(String[] args) throws TiempoMaxSuperado {
    Coche c1 = new Coche("5555-ADS", Emisiones.ECO, true); //true = permiso especial sin límite
    Coche c2 = new Coche("6666-UML", Emisiones.ALTO);
    Moto  m  = new Moto("9999-JRE");
    // línea 1
        System.out.println( c1.precioParking( 40 )); // 40 min x 0,50€/min con descuento -50%
        System.out.println( c2.precioParking( 40 )); // 40 min x 0,50€/min con recargo +25%
        System.out.println( m.precioParking( 180 )); // 180 min x 0,25€/min, moto: sin limite
        System.out.println( c1.precioParking( 160 )); // c1 sin limite de tiempo acumulado
        System.out.println( c2.precioParking( 81 )); // c2 acumularía demasiado tiempo aparcado
        System.out.println( m.precioParking( 5 )); // no imprime nada
    // línea 2
         System.out.println( tms );
         System.out.println( c2.precioParking( 80 )); // OK, le quedaban 80 min
    // línea 3
         System.out.println( Coche.totalRecaudado() ); // Total tickets válidos de coches
    }
  }
}
```

Salida esperada:

```
10.0
25.0
45.0
40.0
matrícula 6666-UML excede tiempo máximo, intentando aparcar 81 minutos.
50.0
125.0
```

CONVOCATORIA EXTRAORDINARIA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas

```
// línea 1 se rellena con:      try{
// línea 2 se rellena con:      } catch(TiempoMaxSuperado tms) {
// línea 3 se rellena con:      } finally{
```

```
class TiempoMaxSuperado extends Exception {
	private String matricula;
	private Integer minutos;
	public TiempoMaxSuperado(String m, Integer minutos) {
		this.matricula = m; this.minutos = minutos;
	}
	@Override public String toString() {
		 return "matricula " + this.matricula
		      + " excede tiempo máximo, intentando aparcar " +
				this.minutos + " minutos.";
	}
}
abstract class Vehiculo {
	protected final String matricula;
	protected Vehiculo(String m) { this.matricula = m; }
	public abstract double precioMinuto();
	public double precioParking(Integer minutos) throws TiempoMaxSuperado {
		return minutos * precioMinuto();
	}
}
enum Emisiones {
	ECO(-50), BAJO(-25), MEDIO(0), ALTO(25);
	private int ajuste; // en % y puede ser ser positivo (recargo) o negativo (descuento)
	private Emisiones(Integer ajuste) { this.ajuste = ajuste; }
	public int getAjuste() { return this.ajuste; }
}
class Coche extends Vehiculo {
	public static final int MAX_MINUTOS = 120;
	public static final double PRECIO_BASE_POR_MINUTO = 0.50;
	private static double totalRecaudado = 0;
	private int totalMinutos = 0;
	private Emisiones emisiones;
	private boolean especial = false;
	public Coche(String m, Emisiones e) {
		super(m);
		this.emisiones = e;
	}
	public Coche(String m, Emisiones e, boolean especial) {
		this(m,e);
		this.especial = especial;
	}
	@Override
	public double precioParking(Integer minutos) throws TiempoMaxSuperado {
		if ( (! this.especial) && (minutos + totalMinutos >  MAX_MINUTOS ) )
			throw new TiempoMaxSuperado(this.matricula, minutos);
		if (! this.especial) totalMinutos += minutos;
		double ticket = super.precioParking(minutos) * (1.0
					+ this.emisiones.getAjuste() / 100.0);
		totalRecaudado += ticket;
		return ticket;
	}
	@Override public double precioMinuto() { return PRECIO_BASE_POR_MINUTO; }
	public static double totalRecaudado() {
		return Coche.totalRecaudado;
	}
}
```

CONVOCATORIA EXTRAORDINARIA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas

Apartado 3. (2,75 puntos)

Necesitamos gestionar agregados de objetos fusionables, de forma que cuando añadimos un objeto al agregado, este objeto se fusione con los objetos contenidos con los que pueda hacerlo. Si no se puede fusionar con ninguno simplemente se añade. Al fusionar un elemento con varios elementos del agregado, desaparecen los elementos fusionados, y solo queda el resultado de la fusión.
Para modelar el contenido del agregado definiremos la siguiente interfaz:

```
interface Fusionable<T>{
	T fusiona(T elem); //Devuelve el resultado de la fusión con elem
	boolean fusionableCon(T elem); //Si es fusionable con elem
}
```

El método fusiona(elem) no modifica los objetos fusionados, sino que crea un elemento nuevo. Si no son fusionables, el método fusiona devolverá null.

Para gestionar fusionables, de cualquier tipo, crearemos la clase AgregadoFusionables. El método add permitirá añadir elementos al agregado, y se podrán encadenar llamadas a dicho método. El método asUnmodifiable devuelve una vista no modificable del agregado, de tipo Collection. El orden de los elementos en el agregado o la vista no es relevante.

Como elemento Fusionable, usaremos intervalos cerrados de números enteros [a..b], con a<=b. Dos intervalos son fusionables si están solapados, o son contiguos.

Se pide: codificar las clases AgregadoFusionables e Intervalo, para que este programa ejemplo produzca la salida dada abajo

```
public static Intervalo i(int a, int b){ //crea un intervalo
	return new Intervalo(a, b);
}
public static void main(String[] args) {
	System.out.println(i(1,4).fusiona(i(5,15))); //Contiguos, se fusionan como [1..15]
	System.out.println(i(1,4).fusionableCon(i(20,25))); //False
	AgregadoFusionables<Intervalo> f= new AgregadoFusionables<>();
	Collection<Intervalo> vista = f.asUnmodifiable();
	System.out.println(f.add(i(1,4)).add(i(10,15)).add(i(40,50))); //Ninguno se fusiona
	System.out.println(f.add(i(3,12)).add(i(20,25)));
	// Imprime f= [[40..50], [1..15], [20..25]], fusiona [1..4],[3..12],[10..15]
	System.out.println(vista); // La vista sigue actualizada
	// El orden de los elementos en el agregado o la vista no es relevante
	System.out.println(vista.contains(i(1,4))); //No contiene [1..4] al haberse fusionado
	// Nota: contains solo busca objetos que sean equivalentes
	System.out.println(vista.contains(i(1, 15))); //[1..15] sí esta
}
```

Salida esperada

```
[1..15]
false
[[1..4], [10..15], [40..50]]
[[40..50], [1..15], [20..25]]
[[40..50], [1..15], [20..25]]
false
true
```

CONVOCATORIA EXTRAORDINARIA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas

```
public class Intervalo implements Fusionable<Intervalo>{
	private final int a;
	private final int b;
	public Intervalo(int a, int b){
		if (a>b) throw new IllegalArgumentException();
		this.a=a;
		this.b=b;
	}
	public Intervalo fusiona(Intervalo r){
		if (fusionableCon(r))
			return new Intervalo(Math.min(a, r.a), Math.max(b, r.b));
		else return null;
	}
	public boolean fusionableCon(Intervalo r) {
		return b>=(r.a-1) && a<=(r.b+1);
	}
	public String toString(){
		return "["+a+".."+b+"]";
	}
	public boolean equals(Object o){
		if (o instanceof Intervalo){
			Intervalo i= (Intervalo) o;
			return (a==i.a && b==i.b);
		} else return false;
	}
	public int hashCode(){
		return a*13+b;
	}
}

public class AgregadoFusionables<T extends Fusionable<T>> {
	private final List<T> elementos= new ArrayList<>();
	private final List<T> view= Collections.unmodifiableList(elementos);

	public AgregadoFusionables<T> add(T f){
		Iterator<T> it=elementos.iterator();
		while (it.hasNext()){
			T e= it.next();
			if (f.fusionableCon(e)){
				f=f.fusiona(e);
				it.remove();
			}
		}
		elementos.add(f);
		return this;
	}
	public String toString(){
		return elementos.toString();
	}
	public Collection<T> asUnmodifiable(){
		return view;
	}
}
```

CONVOCATORIA EXTRAORDINARIA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas

Apartado 4. (2 puntos)

Se quiere construir una clase `StreamTools` con utilidades para el manejo de streams. En primer lugar, tendremos el método `mapSelector` similar al método `map` de la clase `java.util.stream.Stream`, excepto que será un método de clase (estático) y que, por tanto, recibirá como primer parámetro el stream `s` que va a procesar; además `mapSelector` recibirá otros tres parámetros adicionales, `selector`, `fTrue` y `fFalse` de manera que el resultado de `mapSelector(s,selector,fTrue,fFalse)` debe ser similar a la aplicación `map`, pero usando `fTrue` solo para los elementos del stream que cumplan la condición dada mediante `selector`, y usando `fFalse` para los elementos que no la cumplan. Debe tenerse en cuenta que los tipos de los tres parámetros deben aprovechar al máximo la flexibilidad de los genéricos, como se muestra en el código dado abajo.

En segundo lugar, añadiremos el método `mapMultiplex` similar al método anterior, excepto que recibirá, además del stream `s`, un parámetro `index` seguido de un número indefinido de parámetros adicionales, `f0`, `f1`, `f2` ... y `fn` de manera que el resultado de `mapMultiplex(s,index,f0,f1,f2,...,fn)` debe ser similar a `map`, pero ejecutando para cada elemento `x` del stream la `fi`, donde `i` es el valor calculado con `index` para ese elemento `x`. Debe tenerse en cuenta que los tipos de los parámetros deben aprovechar al máximo la flexibilidad de los genéricos, como se muestra en el código dado abajo.

Se pide: implementar la clase `StreamTools` con los métodos *`mapSelector`* y *`mapMultiplex`* descritos arriba, para que el siguiente método `main` produzca la salida indicada abajo.

```
import java.util.*;
import java.util.stream.*;

public class Main {
  public static Integer index(Number n) { return n.intValue() % 4; }
  public static Double f1(Number n) { return  n.intValue() * 0.5; }
  public static String f2(Integer n) { return  n +":"+ n; }
  public static Object f3(Integer n) { return  null; }

  public static void main(String[] args) {
     List<Integer> datos = Arrays.asList( 3, 21, 6, 19, 400 );

     Stream<Number> s1 =  StreamTools.mapSelector( datos.stream(),
                                                   (Number x) -> x.intValue() < 20,
                                                   (x) -> x.toString().length(),
                                                   (Integer x) -> x * 0.5
                                                 );
     System.out.println( s1.collect(Collectors.toList()) );

     System.out.println( StreamTools.mapMultiplex( datos.stream(), Main::index,
                                                   Main::f1, Main::f2, Main::f1, Main::f3 )
                                    .collect(Collectors.toList() ) );
   }
}
```

Salida esperada:

```
[1, 10.5, 1, 2, 200.0]
[null, 21:21, 3.0, null, 200.0]
```

CONVOCATORIA EXTRAORDINARIA

Análisis y Diseño de Software (2016/2017)

Responde a cada apartado en hojas separadas

```
public class StreamTools {
  public static <I, O> Stream<O> mapSelector(Stream<I> s, Predicate<? super I> selector ,
             Function<? super I, ? extends O> fTrue,
             Function<? super I, ? extends O> fFalse) {
       return mapMultiplex(s, e->selector.test(e)?0:1, fTrue, fFalse);
  }
  public static <I, O> Stream<O> mapMultiplex(Stream<I> s,
             Function<? super I, Integer> index,
             Function<? super I, ? extends O> ... f){
       return s.map(e -> f[index.apply(e)].apply(e));
  }
}
```