Análisis y Diseño de Software (2012/2013)

Convocatoria Extraordinaria

Responde a cada apartado en hojas separadas

Apartado 1 (2.5 puntos)

El ayuntamiento de Madrid quiere construir una aplicación para la gestión de un impuesto especial sobre bienes inmuebles (pisos, locales, etc.). Todos los inmuebles vienen descritos por una dirección, y su valor en euros. Por simplificar, se considerarán solamente dos tipos de Inmuebles: Apartamentos y Locales. Cuando se crea un inmueble, se le debe asignar automáticamente un código, como puedes ver en la salida del programa más abajo.

Un apartamento puede ser usado como vivienda habitual, segunda vivienda o estar en alquiler. Para los locales, hay que registrar sus metros cuadrados. El impuesto a pagar en cada caso, viene dado por un impuesto base, que es igual para apartamentos y locales. Este impuesto base se calcula como:

- El 0,5% del valor del inmueble si este tiene un valor mayor de 300.000€
- El 0,35% del valor del inmueble si este tiene un valor mayor de 200.000€ y menor o igual que 300.000€
- El 0,1% del valor del inmueble en otro caso

Para los apartamentos, el impuesto se incrementa en un 0,1% del valor si es segunda vivienda, y en un 0,2% si es en alquiler. Para locales, los impuestos se incrementan por un valor igual a la mitad de sus metros cuadrados.

El gestor de impuestos deberá agrupar los inmuebles por código postal, para luego poder presentarlos ordenados por código postal de menor a mayor, y dentro del mismo código postal, por el valor del inmueble.

Se pide: Completar el programa siguiente para que, cumpliendo los requisitos del enunciado, produzca la salida de más abajo (sin tener en cuenta los saltos de línea ni espacios en blancos, que se han añadido al enunciado por claridad).

```
public class Main {
  public static void main(String args[]) {
    GestorImpuestos gi = new GestorImpuestos();
    // Añadir un apartamento al CP 28003, con valor 250000€, vivienda habitual (código A-1)
    gi.añadeInmueble(28003, new Apartamento("Alenza 12 5C", 250000, UsoVivienda.VIVIENDAHABITUAL));
    // Añadir un local al CP 28003, con valor 150000€, y 145m^2 (código L-2)
    gi.añadeInmueble(28003, new Local("Alenza 12 Bajo", 150000, 145));
    gi.añadeInmueble(28001, new Apartamento("Ayala 7 3A", 570000, UsoVivienda.SEGUNDAVIVIENDA));
    gi.añadeInmueble(28001, new Apartamento("Ayala 7 3B", 575000, UsoVivienda.ALQUILER));
    gi.añadeInmueble(28001, new Local("Ayala 7 Bajo", 270000, 445));

    System.out.println(gi);
    System.out.println("Total impuestos: "+gi.totalImpuestos()+" €");
  }
}
```

Salida:

```
{28001=[(L-5) Ayala 7 Bajo con valor=270000.0€ e impuesto aplicable de 1167.5€,
        (A-3) Ayala 7 3A con valor=570000.0€ e impuesto aplicable de 3420.0€,
        (A-4) Ayala 7 3B con valor=575000.0€ e impuesto aplicable de 4025.0€],
 28003=[(L-2) Alenza 12 Bajo con valor=150000.0€ e impuesto aplicable de 222.5€,
        (A-1) Alenza 12 5C con valor=250000.0€ e impuesto aplicable de 875.0€]}

Total impuestos: 9710.0 €
```

```
class GestorImpuestos {
	private TreeMap<Integer, Set<Inmueble>> inmuebles  = new TreeMap<Integer, Set<Inmueble>>();

	public void añadeInmueble(Integer cp, Inmueble i) {
		if (! this.inmuebles.containsKey(cp)) this.inmuebles.put(cp, new TreeSet<Inmueble>());
		this.inmuebles.get(cp).add(i);
	}

	@Override public String toString() { return this.inmuebles.toString();  }

	public double totalImpuestos () {
		double imps = 0;
		for (Integer cp : this.inmuebles.keySet())
			for (Inmueble i : this.inmuebles.get(cp))
				imps += i.calcularImpuesto();
		return imps;
	}
}

enum UsoVivienda{
	VIVIENDAHABITUAL, SEGUNDAVIVIENDA, ALQUILER
}

abstract class Inmueble implements Comparable<Inmueble>{
	protected static int numInmuebles;
	protected String codigo;
	private String direccion;
	protected double valor;

	public Inmueble(String d, double v) {
		this.direccion = d;
		this.valor = v;
		Inmueble.numInmuebles++;
	}

	public  double calcularImpuesto() {
		if (valor > 300000) return this.valor*0.005;
		else if (valor > 200000) return this.valor*0.0035;
		else return this.valor*0.001;
	}

	@Override public String toString() {
		return "("+this.codigo+") "+this.direccion+" con valor="+this.valor+"€ e impuesto aplicable de
"+this.calcularImpuesto()+"€";
	}

	@Override public int compareTo(Inmueble i) {
		return new Double(this.valor).compareTo(i.valor);
	}
}

class Apartamento extends Inmueble {
	private UsoVivienda uso;

	public Apartamento(String d, double v, UsoVivienda u) {
		super(d, v);
		this.uso = u;
		this.codigo = "A-"+Inmueble.numInmuebles;
	}

	@Override public double calcularImpuesto() {
		double imp = super.calcularImpuesto();
		if ( this.uso.equals(UsoVivienda.SEGUNDAVIVIENDA)) imp += this.valor*0.001;
		else if ( this.uso.equals(UsoVivienda.ALQUILER)) imp += this.valor*0.002;
		return imp;
	}
}

class Local extends Inmueble {
	private double metrosCuadrados;

	public Local(String d, double v, int mc) {
		super(d, v);
		this.metrosCuadrados = mc;
		this.codigo = "L-"+Inmueble.numInmuebles;
	}

	@Override public double calcularImpuesto() {
		return super.calcularImpuesto()+metrosCuadrados*0.5;
	}
}
```

Apartado 2 (3.5 puntos)

a) En un paquete Java para la gestión de averías se han definido la interfaz `Averia`, que deben implementar las clases que representen averías, y la interfaz `ElementoAveriable` que deben implementar las clases que representen cosas que puedan sufrir averías. Las averías tienen un nivel de importancia, que viene dado por el tipo enumerado `NivelAveria`, y se puede acceder al elemento que las causó con el método `causa()`. La definición de estas dos interfaces está en el listado de más abajo.

De manera adicional, se desea definir una interfaz `GestorAverias`, que deben implementar las clases encargadas de solucionar las averías. Esta interfaz debe contener un método `procesarAveria(…)`, que reciba la avería a solucionar. Queremos construir una clase `Averiable` que podrá ser utilizada como clase base por las clases que implementen la interfaz `ElementoAveriable`. La clase `Averiable` debe incluir implementaciones por defecto para los métodos de la interfaz `ElementoAveriable`. De este modo, el método `incluirGestorAverias` añadirá un nuevo `GestorAverias` (no se permiten duplicados), y el método `detectadaAveria` iterará por todos los gestores de averías hasta que uno de ellos pueda solucionarla (y entonces devolverá true). Este método debe detectar si la avería recibida tiene como causa un elemento averiable que no es el que está ejecutando el método, lanzando una excepción en este caso.

Se pide: implementa la clase `Averiable`, la excepción `AveriaInvalida` y la interfaz `GestorAverias` **(1 punto)**

```
package averias;
// cada interface en un fichero separado
public interface Averia {
        enum NivelAveria  { BAJO, MEDIO, ALTO, MUY_ALTO }
        NivelAveria       nivelAveria();
        ElementoAveriable causa();
}
public interface ElementoAveriable {
        boolean detectadaAveria      ( Averia av ) throws AveriaInvalida;
        boolean incluirGestorAverias ( GestorAverias gestor);
}
// Añade la interfaz GestorAverias, la excepción AveriaInvalida, y la clase Averiable
```

b) Se quiere usar el paquete anterior para construir un gestor de averías para `Vehiculo`s. Los `Vehiculo`s pueden sufrir averías de tipo `AveriaVehiculo`, que pueden solucionarse por talleres o fabricantes. Los talleres sólo pueden atender averías producidas en coches de entre una colección de marcas con las que trabajan, de una complejidad MEDIO o menor. Los fabricantes sólo pueden arreglar averías de una marca determinada de coche, de complejidad ALTA o mayor. Averías de menor complejidad (o de otra marca) se redirigen a un taller asociado.

Se pide: completa el siguiente listado para que, usando el paquete averias, se obtenga la salida de más abajo. Para ello, debes completar la clase Vehiculo si es necesario, así como añadir las clases Fabricante, Taller, AveriaVehiculo y otras clases adicionales si fuese necesario **(2,25 puntos)**

```
package vehiculos; import averias.*;        // en fichero Vehiculo.java
public class Vehiculo extends Averiable {
   private String marca, matricula;
   public Vehiculo(String marca, String matricula) { this.marca = marca; this.matricula = matricula; }
   @Override public String toString() { return "Vehiculo " + marca + " matricula : " + matricula; }
   // Completar si es necesario
}
```

```
package vehiculos; import averias.*;        // en fichero MainAverias.java
public class MainAverias {
  public static void main(String[] args) {
    Vehiculo v1 = new Vehiculo("Toyota", "0001-GOD"); // marca y matrícula
    Vehiculo v2 = new Vehiculo("Ford", "6969-GGG");
    Taller unTaller = new Taller("AlcoToyo",                                    // nombre del taller
                                 Arrays.asList("Toyota", "Mitsubishi", "Ford")); // marcas que trabaja
    GestorAverias unFabricante = new Fabricante("Toyota", unTaller); // marca del fabricante y taller asociado
    v1.incluirGestorAverias(unTaller);
    v1.incluirGestorAverias(unFabricante);  // v1 memoriza sus dos GestorAverias
    v2.incluirGestorAverias(unFabricante);  // unFabricante es GestorAverias incluido en v1 y v2
    v2.incluirGestorAverias(unFabricante);  // no se incluye dos veces un mismo GestorAverias
    AveriaVehiculo av1 = new AveriaVehiculo("Averia electrónica", Averia.NivelAveria.ALTO, v1);
    AveriaVehiculo av2 = new AveriaVehiculo("Pérdida de aceite", Averia.NivelAveria.BAJO, v2);
    try {
        v1.detectadaAveria(av1);
        v2.detectadaAveria(av2);
        v1.detectadaAveria(av2);      // Esto debe hacer saltar una excepción
    } catch (AveriaInvalida av) { System.out.println( av ); }
 }}
```

Salida:

```
El taller AlcoToyo no atenderá la avería Averia electrónica por ser demasiado compleja
El fabricante Toyota atenderá la avería Averia electrónica
El fabricante Toyota no atenderá la avería Pérdida de aceite ya que no trabaja la marca Ford
El taller AlcoToyo atenderá la avería Pérdida de aceite
Averia inválida! Pérdida de aceite con causa: Vehiculo Ford matricula : 6969-GGG recibida por Vehiculo Toyota matricula :
0001-GOD
```

c) Explica qué patrón o patrones de diseño has utilizado, y qué papel juegan cada una de las clases e interfaces implementadas en dicho patrón. **(0,25 puntos)**

a)

```
public abstract class Averiable implements ElementoAveriable {
	private Set<GestorAverias> gestores = new LinkedHashSet<GestorAverias>();
	@Override public boolean detectadaAveria(Averia av) throws AveriaInvalida {
		if (!av.causa().equals(this)) throw new AveriaInvalida(av, this);
		for (GestorAverias g : this.gestores)
			if (g.procesarAveria(av)) return true;
		return false;
	}
	@Override public boolean incluirGestorAverias( GestorAverias gestor) {
		return this.gestores.add(gestor);
	}
}
//---------------------------------------------------------------
public class AveriaInvalida extends Exception {
	private Averia av;
	private ElementoAveriable ea;
	public AveriaInvalida (Averia a, ElementoAveriable ea) { this.av = a; this.ea = ea; }
	@Override
	public String toString() {
	   return "Averia inválida "+this.av+" con causa: "+this.av.causa()+" recibida por "+this.ea;
	}
}
//---------------------------------------------------------------
public interface GestorAverias {
	boolean procesarAveria(Averia averia) ;
}
```

b)

```
public class Vehiculo extends Averiable {
	private String marca, matricula;
	public Vehiculo(String marca, String matricula) { this.marca = marca; this.matricula = matricula; }
	@Override public String toString() { return "Vehiculo " + marca + " matricula : " + matricula; }
	public String marca() { return this.marca; }

}
//---------------------------------------------------------------
public class Fabricante implements GestorAverias {
	private String marca;
	private GestorAverias siguiente;

	public Fabricante(String marca, GestorAverias next) {
		this.marca = marca;
		this.siguiente = next;
	}

	@Override
	public boolean procesarAveria(Averia averia) {
		Vehiculo v = (Vehiculo)averia.causa();
		if (this.marca != v.marca()) {
			System.out.println("El fabricante "+this.marca+" no atenderá la avería "+averia+" ya que no
trabaja la marca "+v.marca());
			return this.siguiente.procesarAveria(averia);
		}
		if ( averia.nivelAveria().ordinal() <= Averia.NivelAveria.MEDIO.ordinal() ) {
			System.out.println("El fabricante "+this.marca+" no atenderá la avería "+averia+" de nivel
"+averia.nivelAveria()+" se lo reenvia a "+v.marca());
			return this.siguiente.procesarAveria(averia);
		}
		else {
			System.out.println("El fabricante "+this.marca+" atenderá la avería "+averia);
			return true;
		}
	}
}
//---------------------------------------------------------------------
public class Taller implements GestorAverias {
	private String nombre;
	private Set<String> marcas = new HashSet<String>();

	public Taller(String nombre, Collection<String> marcas) {
		this.nombre = nombre;
		this.marcas.addAll(marcas);
	}
	@Override
	public boolean procesarAveria(Averia averia) {
		Vehiculo v = (Vehiculo)averia.causa();
		if (!this.marcas.contains(v.marca())) {
			System.out.println("El taller "+nombre+" no atenderá la avería "+averia+" ya que no trabaja la
marca "+v.marca());
			return false;
		} else if ( averia.nivelAveria().ordinal() > Averia.NivelAveria.MEDIO.ordinal() ) {
```

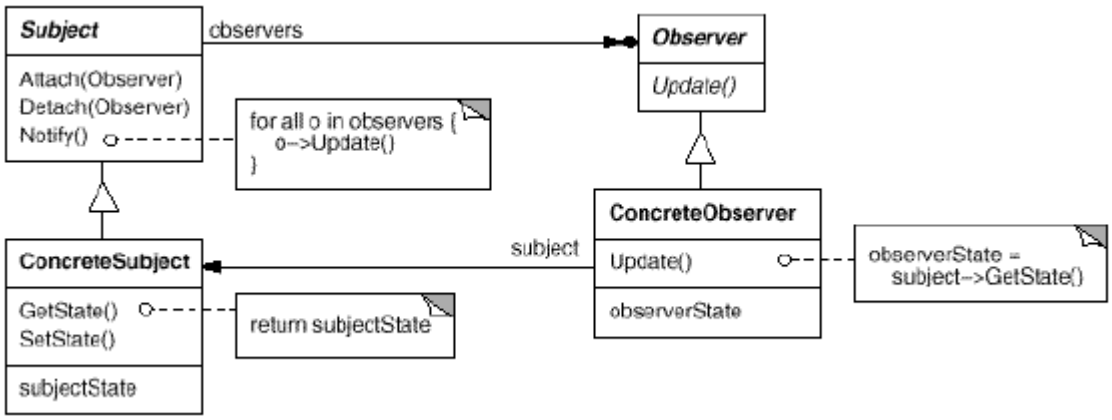
```
                        System.out.println("El taller "+nombre+" no atenderá la avería "+averia+" por ser demasiado
compleja");
                        return false;
                } else {
                        System.out.println("El taller "+nombre+" atenderá la avería "+averia);
                        return true;
                }
        }
}
//-----------------------------------------------------------------
public class AveriaVehiculo implements Averia {

        private Averia.NivelAveria nivel;
        private String      descripcion;
        private Vehiculo    causa;

        public AveriaVehiculo( String d, NivelAveria nivel, Vehiculo causa) {
                this.nivel = nivel;
                this.descripcion = d;
                this.causa = causa;
        }

        @Override public NivelAveria nivelAveria()  { return nivel; }
        @Override public String toString() { return descripcion;}
        @Override public Vehiculo causa()  { return causa; }
}
```

c) El patrón usado es el *Observer*, ya que los vehículos (sujetos) pueden recibir averías, momento en el que se notifica a los observadores (Talleres y Fabricantes).



Donde:

- Averiable hace el papel de Subject
- El método detectadaAvería hace el papel de “notify”.
- El método incluirGestorAverias hace el papel de “attach”.
- GestorAveria hace el papel de Observer
- El método procesarAveria hace el papel de “update”, donde se le pasa como parámetro una Averia.
- Vehiculo es un ConcreteSubject.
- Hay dos ConcreteObservers: Taller y Fabricante (en nuestro caso no ha hecho falta una referencia desde Taller o Fabricante a Vehiculo, ya que la información de la Averia llega como parámetro de procesarAveria).

Apartado 3 (1.5 puntos)

Necesitamos crear una estructura de datos que permita asociar información, en forma de pares clave-valor, de forma similar al tipo *Map* de Java. Para ello, se nos proporciona esta interfaz:

```
 public interface SimpleMap<K, V> {
       V get(K key);
       V put(K key, V value);
       V remove(K key);
       boolean containsKey(K key);
}
```

Además, se nos da una implementación trivial de esta interfaz usando un HashMap (Nota: Recuerda que HashMap admite null como claves y como valor):

```
public class HashSimpleMap<K,V> extends HashMap<K, V> implements SimpleMap<K, V>{
}
```

Queremos crear una implementación más sofisticada de la interfaz SimpleMap (que llamaremos COWMap) que nos permita hacer una copia de cualquier otro SimpleMap que se le pase como parámetro en el constructor, con las siguientes características:

a) La copia sólo almacenará los cambios respecto al original, producidos al modificar, añadir, o eliminar valores anteriores, cuando se llaman a los métodos *get*, *put* y *remove* sobre la copia. Esto se llama Copy On Write (COW), y permite ahorrar memoria en los casos en que no queremos que cambios en la copia afecten al original, ya que sólo se almacenan los cambios.
b) Si un valor no alterado en la copia es alterado posteriormente en el original (mediante la invocación de los métodos *get*, *put* y *remove* sobre el original), este cambio se verá reflejado también en la copia.
c) Si un valor alterado en la copia es alterado posteriormente en el original, este cambio no se verá reflejado en la copia.

Los tipos de los parámetros genéricos requeridos al original serán lo más flexible posible que permita la implementación. En concreto, se deben poder definir copias para almacenar datos de un tipo más amplio que el de los valores almacenados en el original. Por otro lado, la copia podrá definirse con tipo de claves más restringido que el de las claves del original. De esta forma, se debe poder ejecutar el siguiente método main:

```
public static void main(String[] args) {
       SimpleMap<Object, Integer> original= new HashSimpleMap<Object, Integer>();
       original.put("uno", 1);
       original.put(2, 2); original.put(2.0, 2); original.put("dos", 2);

       SimpleMap<String, Number> copia= new COWMap<String, Number>(original);
       copia.put("uno", 1.0000001); //cambiamos el valor y tipo en copia, ahora es un double
       copia.remove("dos"); //eliminamos "dos" de copia, pero no de original

       original.put("tres", 3); //al modificar original, el cambio se refleja en copia

       System.out.println(original.get("uno"));
       System.out.println(original.get("dos"));
       System.out.println(original.get("tres"));
       System.out.println(copia.get("uno"));
       System.out.println(copia.get("dos"));
       System.out.println(copia.get("tres"));
}
```

Se pide: Implementa la clase COWMap según los requisitos descritos, de forma que la salida del main anterior sea la siguiente:

Salida:

```
1
2
3
1.0000001
null
3
```

Solución:

```
public class COWMap<K, V> implements SimpleMap<K, V> {
	/*
		En copy guardamos los elementos cambiados, ya que source no puede cambiarse.
		También hay que saber si el elemento se ha eliminado, usando null como valor.
		Si necesitásemos que null pudiera ser un valor normal, habría que usar otro mecanismo,
		por ejemplo un conjunto con los eliminados. El enunciado no indica nada, elegimos la
		opción más sencilla, null asociadado a la clave significará eliminado.
	*/

	final private SimpleMap<? super K, ? extends V> source;
	final private SimpleMap<K, V> copy; //también podría servir un HashMap
	public COWMap(SimpleMap<? super K, ? extends V> source){
		this.source=source;
		this.copy= new HashSimpleMap<K, V>();
	}

	@Override
	public V get(K key) {
		if (copy.containsKey(key)) return copy.get(key);
		else return source.get(key);
	}

	@Override
	public V put(K key, V value) {
		V old=get(key);
		copy.put(key,  value);
		return old;
	}

	@Override
	public V remove(K key) {
		if (containsKey(key)) return put(key, null);
		else return null;
	}

	@Override
	public boolean containsKey(K key) {
		return get(key)!=null;
	}
}
```

Apartado 4. (2.5 puntos)

Vamos a desarrollar una aplicación informática para la gestión de los *productos de ahorro* que un banco *contrata* con sus *clientes*. Un *contrato* lo firman varios clientes como titulares del producto de ahorro que se contrata. Cada *cliente* viene identificado por su DNI, nombre y fecha de nacimiento, y tiene un atributo adicional que describe su nivel de riesgo máximo asumible: bajo, moderado, alto o muy alto. Todos los *productos de ahorro* vienen descritos por un identificador, fecha de contratación, saldo actual y nivel de riesgo asociado. En el momento de la contratación, el saldo actual es la cantidad inicial de dinero que los titulares colocan en ese producto, y posteriormente el saldo actual se actualiza mensualmente según las condiciones de cada producto. Existen tres tipos de productos de ahorro elementales denominados *Cuenta Deposito, Fondo Bolsa, y Gestor Activo*, que se pueden contratar individualmente o combinándolos como otro tipo de producto de ahorro denominado *Cartera Combinada* (que a su vez podrán contener también otras carteras combinadas). La *Cuenta Deposito* tiene un perfil de riesgo bajo ya que producirá regularmente un *porcentaje de interés prefijado* en la creación. El *Fondo Bolsa* tiene un perfil de riesgo muy alto ya que su saldo variará (al alza o a la baja) según haya sido la variación de la Bolsa Española durante el último mes, e incluirá un gasto mensual que viene dado por un *porcentaje de comisión prefijado* en la contratación y calculado sobre el saldo actual del producto. El *Gestor Activo* se contrata con un *porcentaje máximo de pérdida mensual*, un *porcentaje de comisión sobre el saldo*, y otro *porcentaje de comisión sobre los beneficios* obtenidos en el mes. Cada Gestor Activo lleva asociado un nivel de riesgo dependiente de su límite de pérdida : muy alto si es > 12%, moderado si es < 3%, y alto en el resto de los casos. Una *Cartera Combinada* viene descrita por la lista de productos de ahorro que la componen y el peso de cada uno de ellos en la cartera, entendiendo por peso el porcentaje del saldo inicial destinado a cada producto. El nivel de riesgo de una cartera combinada es el nivel de riesgo mayor entre los productos que la componen.

Se pide:

(a) Un diagrama de clases que represente el enunciado anterior. **(1.9 puntos)**

(b) Añade los siguientes métodos a la(s) clase(s) adecuada(s) de tu diagrama de clases, indicando sus parámetros de entrada y valor de retorno: **(0.6 puntos)**

   a. Un método para obtener el nivel de riesgo de los productos de ahorro.
   b. Un método que compruebe si en alguno de los contratos del banco se incumplen las condiciones de riesgo máximo asumible de alguno de sus titulares en relación al producto contratado.
   c. Un método que será invocado al final de cada mes para actualizar el saldo actual de un producto de ahorro.
   d. Un método que cambie la composición de una cartera combinada sustituyendo un producto de ahorro por otro.

Nota: No hace falta incluir código Java, ni constructores ni métodos *getters* y *setters* en el diagrama de clases.