Convocatoria Extraordinaria

Análisis y Diseño de Software (2014/2015)

Responde a cada apartado en hojas separadas

Apartado 1. (2,5 puntos)

Se quiere construir un sistema para gestionar la elaboración y composición de los pedidos en un restaurante de comida rápida. Cada pedido tiene un número de pedido, un nombre (que sirve para identificar la mesa u otra información adicional), y consta además de una serie de elementos, que a su vez pueden ser sencillos o complejos. Los elementos complejos (por ejemplo un menú estándar, o un tipo de plato combinado) pueden estar compuestos de elementos sencillos (refresco de cola, patatas fritas) u otros elementos complejos (hamburguesa completa, que a su vez tiene varios elementos, como la propia hamburguesa o el pan), y pueden contener elementos repetidos (como en el caso de la hamburguesa doble), para indicar que requiere varias unidades de estos. Los elementos sencillos se elaboran en un único paso y tienen un precio asociado. El precio de los elementos complejos y el de los pedidos resulta de la suma del precio de los elementos que lo componen. Todos los elementos, al igual que los pedidos, tienen un nombre.
El sistema NO tratará cada unidad de cada elemento de forma independiente, si no que se limitará a controlar la demanda y oferta de cada elemento. Para ello se usarán dos valores dinámicos, por un lado el número de unidades listas de dicho elemento (sencillos o complejo), y por otro el número de unidades demandadas. Al entrar un pedido en el sistema se incrementa la demanda de los elementos y subelementos que lo componen. Cuando una unidad de un elemento está lista (porque se ha cocinado, o porque ha llegado al almacén si no requiere preparación) se aumenta el número de unidades listas. En el caso de las unidades de elementos complejos (o pedido), es necesario además que existan suficientes unidades de los elementos que la componen. En este caso, cuando una nueva unidad está lista, se decrementarán las unidades preparadas y demandadas de los elementos que lo componen.

Nota: Queda fuera del ámbito del problema el algoritmo que determina en qué orden se preparan los elementos, por lo que no es necesario disponer de ningún sistema de colas de prioridad, o de notificación de elementos preparados.
Los pedidos deberían tener un sistema parecido a los elementos complejos para saber si están preparados, y podrían usar también un sistema con dos valores igual que dichos elementos, aunque en este caso el número de unidades demandadas o preparadas sea solo cero o uno.

Se pide:

a) Realiza el diagrama de clases que describe el sistema anterior (1,6 puntos).
b) Añade los métodos, incluyendo argumentos y tipo de retorno, necesarios para (0,6 puntos):
    1. Saber si un elemento complejo puede prepararse, al disponer de las unidades necesarias de subelementos.
    2. Añadir un pedido al sistema.
    3. Definir un nuevo elemento al sistema.
    4. Conocer el precio de un elemento

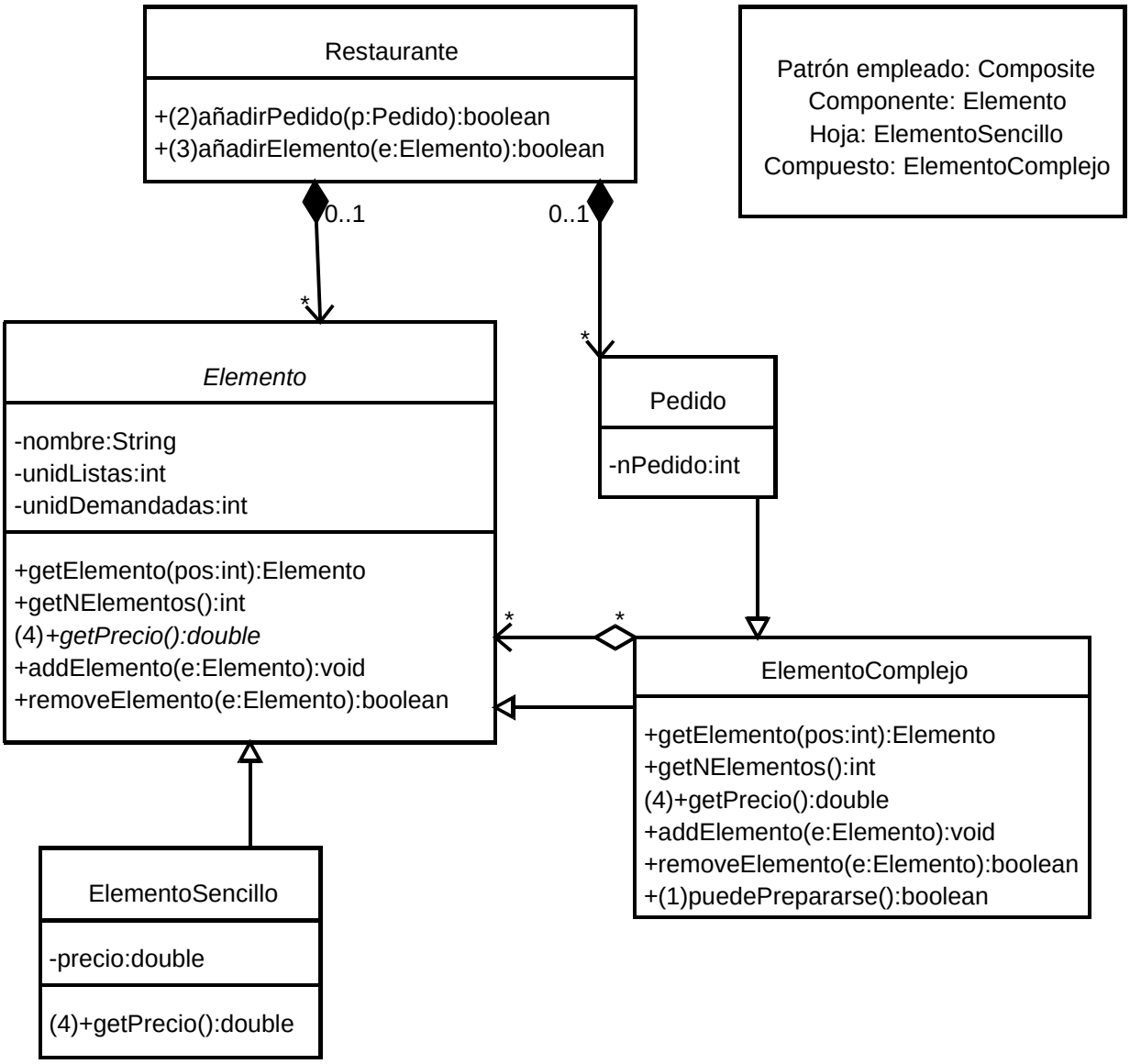
c) ¿Qué patrón o patrones de diseño has empleado o sería conveniente emplear? (0,3 puntos)

Convocatoria Extraordinaria

Análisis y Diseño de Software (2014/2015)

Responde a cada apartado en hojas separadas

Solución:



Notas:
Se han añadido los métodos que se piden y el patrón directamente al diagrama. Los métodos que no están numerados no eran necesarios, y se han añadido únicamente para ilustrar mejor el uso del patrón.

No se ha penalizado las respuestas que han incluido también los patrones Observer o Singleton, aunque no eran necesarios para implementar la funcionalidad descrita en el enunciado.

El patrón Observer podría ser útil en el algoritmo que determina el orden de preparación de los alimentos, como parte del sistema de notificación de elementos preparados. Sin embargo en el enunciado se indica expresamente que esto queda fuera del ámbito del problema, por lo tanto no se utiliza dicho patrón.

Convocatoria Extraordinaria

Análisis y Diseño de Software (2014/2015)

Responde a cada apartado en hojas separadas

Apartado 2. (3 puntos).

Programa en Java una aplicación muy sencilla para la gestión de seguros. Un seguro puede cubrir elementos `Asegurables`. Por simplicidad asumiremos que se pueden asegurar `Vehiculos` y `Casas`. Todo elemento asegurable tiene un valor de compra y una antigüedad en años. Los `Vehiculos` además se describen por su matrícula. Las `Casas` además tienen unos metros cuadrados y una situación, que puede ser: céntrico, playa, afueras y campo.

El valor actual de un elemento asegurable se calcula como el valor de compra menos el 5% de su valor de compra por su antigüedad, hasta un máximo de 10 años. Es decir, si tiene un año de antigüedad se descuenta el 5% del valor de compra, si tiene dos el 10%, y así sucesivamente hasta el 50% (si tiene más de 10 años se calcula como si tuviera 10). No obstante, en el caso concreto de las casas, se sigue este cálculo general sólo si la casa tiene una antigüedad mayor o igual de 5 años. En caso contrario, el valor actual es el valor de compra menos un porcentaje que depende de su situación: un 0% si es céntrico, un 5% si está en el campo, un 10% si está en la playa y un 15% si está en las afueras.

El precio anual del seguro para un elemento asegurable depende de su tipo. Si es una casa, es el 0.1% de su valor actual. Si es un vehículo es el 1% de su valor actual si tiene menos de 10 años, y el 1.5% si tiene más. Un seguro puede cubrir varios elementos asegurables, y se calcula como la suma del precio anual de cada uno de sus elementos.

Finalmente, queremos configurar los seguros de manera global con posibles descuentos, que se aplicarían sobre el precio anual de todos los seguros existentes.

Se pide: Construye la clase `Seguro` y el resto de clases necesarias para que el programa de más debajo de la salida indicada.

```
import java.util.*;

public class Main {

  public static void main(String[] args) {
    Seguro seguroPedro = new Seguro();
    // Añadimos al seguro un vehiculo de 12.000€, 4 años de antiguedad y matricula 1234FG
    seguroPedro.addElemento(new Vehiculo(12000, 4, "1234FG"));
    // Añadimos una casa de 300.000€, 3 años de antigüedad, 70m2 y en situación céntrica
    seguroPedro.addElemento(new Casa(300000, 3, 70, Situacion.CENTRICO));

    System.out.println("Precio anual del seguro para Pedro: "+seguroPedro.precioSeguro());

    Seguro seguroAna = new Seguro();
    // Añadimos una casa de 200.000€, 10 años de antigüedad, 50m2 en la playa
    seguroAna.addElemento(new Casa(200000, 10, 50, Situacion.PLAYA));

    System.out.println("Precio anual del seguro para Ana: "+seguroAna.precioSeguro());

    Seguro.setDescuento(0.1);
    System.out.println("Precio anual con descuento del seguro para Pedro:"+
                       seguroPedro.precioSeguro());
    System.out.println("Precio anual con descuento del seguro para Ana: "+
                       seguroAna.precioSeguro());
  }
}
```

Salida esperada:

```
Precio anual del seguro para Pedro: 396.0
Precio anual del seguro para Ana: 100.0
Precio anual con descuento del seguro para Pedro: 356.4
Precio anual con descuento del seguro para Ana: 90.0
```

Convocatoria Extraordinaria

Análisis y Diseño de Software (2014/2015)

Responde a cada apartado en hojas separadas

Solución:

```
abstract class Asegurable {
	protected double valorCompra;
	protected int antiguedad;
	private static final double MENOR10 = 0.05;
	private static final double MAYOREQ10 = 0.5;

	public double valorActual() {
		if (this.antiguedad < 10)
			return this.valorCompra-MENOR10*this.valorCompra*this.antiguedad;
		else
			return this.valorCompra-MAYOREQ10*this.valorCompra;
	}
	public Asegurable(double valor, int antiguedad) {
		this.antiguedad = antiguedad;
		this.valorCompra = valor;
	}
	public abstract double valorSeguro();
}
class Vehiculo extends Asegurable {
	private String matricula;
	private static final double SMENOR10 = 0.01;
	private static final double SMAYOREQ10 = 0.015;

	public Vehiculo(double valorCompra, int antiguedad, String matricula) {
		super(valorCompra, antiguedad);
		this.matricula = matricula;
	}
	@Override public double valorSeguro() {
		return (this.antiguedad < 10) ? this.valorActual()*SMENOR10 : this.valorActual()*SMAYOREQ10;
	}
}
enum Situacion {
	CENTRICO (0), AFUERAS (0.15), CAMPO (0.05), PLAYA(0.1);
	private double coeficiente;
	Situacion(double c) { this.coeficiente = c; }
	public double getCoeficiente() { return this.coeficiente;	}
}
class Casa extends Asegurable {
	private int metros;
	private Situacion situacion;
	private static final double COSTECASA = 0.001;

	public Casa(double valor, int antiguedad, int metros, Situacion s) {
		super(valor, antiguedad);
		this.metros = metros;
		this.situacion = s;
	}
	public double valorActual()  {
		return this.antiguedad >= 5 ? super.valorActual() : this.valorCompra-
this.valorCompra*this.situacion.getCoeficiente();
	}
	@Override public double valorSeguro() {
		return this.valorActual()*COSTECASA;
	}
}
class Seguro {
	private List<Asegurable> elementos = new ArrayList<Asegurable>();
	private static double descuento = 0.0;

	public void addElemento(Asegurable a) {
		this.elementos.add(a);
	}
	public static void setDescuento(double desc) {
		Seguro.descuento = desc;
	}
	public double precioSeguro() {
		double acum = 0;
		for (Asegurable a: this.elementos)
			acum += a.valorSeguro();
		acum -= acum*Seguro.descuento;
		return acum;
	}
}
```

Convocatoria Extraordinaria

Análisis y Diseño de Software (2014/2015)

Responde a cada apartado en hojas separadas

Apartado 3. (3 puntos).

Se desea representar *artículos* descritos por una serie de *características medibles* que tienen un nombre y un valor entero (p.ej.: edad, peso, estatura, localidad, etc.) y que se usan para comparar artículos. La *similitud* entre dos artículos se calcula sumando la *distancia* entre cada dos características medibles que ambos artículos tengan en común (es decir, características con el mismo nombre). Esta distancia se calculará de forma distinta según el tipo concreto de características que se estén comparando. Por ejemplo, la distancia entre dos características puede ser el valor absoluto de la resta de sus dos valores; este es el caso de la edad en el programa dado abajo. Para otras características bastaría con distinguir entre distancias 0 y 1 para la igualdad y desigualdad de valores, respectivamente, como es el caso de la provincia en el programa dado.

Al crear un artículo se debe evitar almacenar más de un valor para una misma característica; por eso se ignora la edad 99 en `p1` en el código dado abajo. Se debe poder calcular la similitud entre artículos de distinta clase, pero sumando sólo las distancias entre las características medibles de igual nombre. En cambio, si se intenta calcular la distancia entre dos características con distinto nombre, se provocará una excepción tal y como ocurre con peso y edad en el programa dado.

Se pide: implementar la interfaz `IMedible` y las clases `CaracteristicaMedible`, `Edad`, `Provincia`, `Articulo`, `Perfil` y `CaracteristicasIncompatibles` para que, de acuerdo con el comportamiento descrito en el enunciado, el siguiente método `main` produzca la salida indicada abajo, teniendo en cuenta también el código auxiliar dado. Nótese que **no se pide** implementar la clase `Peso` que sería muy similar a `Edad`.

Código auxiliar dado:

```
// Constructor y Métodos dados para la clase CaracteristicaMedible
public CaracteristicaMedible(String n, Integer v) { valor = v; nombre = n; }

@Override    public final String getNombre() { return nombre; }
@Override    public final Integer getValor() { return valor; }

@Override public String toString() { return "<"+this.getNombre()+":"+this.getValor()+">"; }

// Método dado para la clase Articulo
@Override public String toString() { return this.getCaracteristicasMedibles().toString(); }
```

Programa ejemplo:

```
import java.util.*;
public class Main {
   public static void main(String[] args) {
       CaracteristicaMedible miEdad = new Edad(50);
       IMedible miProvincia = new Provincia(28);
       Articulo p1 = new Perfil( miEdad, new Peso(70), miProvincia, new Edad(99));
       Perfil p2 = new Perfil( new Provincia(14), new Edad(55));
       System.out.println(p1);   System.out.println(p2);
       try {
           System.out.println(p1.similitud(p2));   // 6 = distancia 5 edad + 1 en provincia
           System.out.println(new Peso(66).distancia( new Edad(66) )); // incompatibles
       } catch (CaracteristicasIncompatibles e) {
           System.out.println( "Excepción " + e);
       }
   }
}
```

Salida esperada:

```
[<Edad:50>, <Peso:80>, <Provincia:28>]
[<Provincia:14>, <Edad:55>]
6.0
Excepción Medidas incompatibles: <Peso:66><Edad:66>
```

Convocatoria Extraordinaria

Análisis y Diseño de Software (2014/2015)

Responde a cada apartado en hojas separadas

Solución (solo las clases que se piden):

```
public class CaracteristicasIncompatibles extends Exception {
  private IMedible medida1, medida2;
  public CaracteristicasIncompatibles(IMedible m1, IMedible m2) { medida1=m1; medida2=m2; }
  @Override public String toString() { return "Medidas incompatibles: " + m1 + m2; }
}
public interface IMedible  {
  String getNombre();
  Integer getValor();
  default void verificaComparables(IMedible m) throws CaracteristicasIncompatibles {
      if (! this.getNombre().equals(m.getNombre()))
                                throw new CaracteristicasIncompatibles(this,m);
  }
  double distancia(IMedible m) throws CaracteristicasIncompatibles;
}

public abstract class CaracteristicaMedible implements IMedible {
  private String nombre;
  private Integer valor;
  @Override public abstract double distancia(IMedible m) throws
CaracteristicasIncompatibles;
  @Override public boolean equals(Object obj) {
             if (this == obj) return true;
             if (! (obj instanceof CaracteristicaMedible)) return false;
             return this.getNombre().equals(((CaracteristicaMedible) obj).getNombre());
    }
  @Override public int hashCode() { return this.getNombre().hashCode(); }
}
public class Edad extends CaracteristicaMedible {
  public Edad(Integer v) { super("Edad",v); }
  @Override public double distancia(IMedible m) throws CaracteristicasIncompatibles {
             this.verificaComparables(m);
             return Math.abs( this.getValor() - m.getValor() );
  }
}
public class Provincia extends CaracteristicaMedible {
  public Provincia(Integer v) { super("Provincia", v); }
  @Override public double distancia(IMedible m) throws CaracteristicasIncompatibles {
             this.verificaComparables(m);
             return (this.getValor().equals(m.getValor())) ? 0 : 1;
  }
}
import java.util.*;
public class Articulo {
  private Set<IMedible> medidas;
  public Set<IMedible> getCaracteristicasMedibles() { return
                                                    Collections.unmodifiableSet(medidas); }
  public Articulo(IMedible... m) { medidas = new LinkedHashSet<>(Arrays.asList(m)); }

  public double similitud(Articulo otro) throws CaracteristicasIncompatibles {
             double d = 0.0;
             for (IMedible m : medidas)
                    for (IMedible m2 : otro.medidas)
                           if (m.getNombre().equals(m2.getNombre())) d += m.distancia(m2);
             return d;
  }
}

public class Perfil extends Articulo {  public Perfil(IMedible... m) {  super(m); } }
```

Convocatoria Extraordinaria

Análisis y Diseño de Software (2014/2015)

Responde a cada apartado en hojas separadas

Apartado 4. (1,5 puntos).

En nuestra clase `StreamUtil` se agruparán métodos útiles para manipular streams. En primer lugar, tendremos el método `map2` similar al método `map` de la clase `java.util.stream.Stream` (ver su API resumida en comentario del código dado abajo), excepto que será un método de clase (estático) y que, por tanto, recibirá como primer parámetro el stream `s` que va a procesar; además `map2` recibirá otros dos parámetros adicionales, `f1` y `f2` de manera que el resultado de `map2(s,f1,f2)` debe ser equivalente a la aplicación sucesiva de `map` dos veces, primero con `f1` y luego con `f2`, es decir `s.map(f1).map(f2)`. Debe tenerse en cuenta que los tipos `f1` and `f2` deben ser *compatibles entre sí* y con el tipo base de `s`.

En segundo lugar, añadiremos el método `mapN` similar al método `map2`, excepto que recibirá un número indefinido de parámetros adicionales, `f1`, `f2`, `f3` … y `fn` de manera que el resultado de `mapN(s,f1,f2,f3,…,fn)` debe ser equivalente a n aplicaciones sucesivas de `map`, primero con `f1` y luego con `f2`, y así sucesivamente hasta aplicarlo con `fn`, es decir, `s.map(f1).map(f2).map(f3) … .map(fn)`. Debe tenerse en cuenta que los tipos de `f1`, `f2`, `f3` … y `fn` deben ser todos *idénticos entre sí* y compatibles con el tipo base de `s`, y por lo tanto, no sirve implementar `map2` con una simple llamada a `mapN` con los dos parámetros `f1` y `f2` de `map2` (que pueden no ser idénticos).

Se pide: implementar la clase `StreamUtil` con los métodos `map2` y `mapN` descritos arriba, para que el siguiente método `main` produzca la salida indicada abajo.

```
/*   FRAGMENTO DEL API DE java.util.stream.Stream
Devuelve un stream con los resultados de aplicar a los elementos del
stream this la función dada como parámetro.
<R> Stream<R>      map(Function<? super T,? extends R> mapper)
*/
import java.util.stream.*;

public class Main {
  public static String  texto(Double d) { return d.toString(); }
  public static Integer digitos(String s) { return s.length(); }

  public static Double alCuadrado(Double x) { return x * x; }
  public static Double menos1(Double x) { return x - 1.0; }
  public static Double div2(Double x) { return x / 2.0; }

  public static void main(String[] args) {
       Stream<Double> stream = Stream.of(1.0, 25.25, 19.5, 2000.0);
       System.out.println(
             StreamUtil.map2(stream, Main::texto, Main::digitos)
                       .collect(Collectors.toList())
       );
       Stream<Double> otroStream = Stream.of(1.0, 2.0, 4.0, 5.0);
       System.out.println(
             StreamUtil.mapN(otroStream, Main::alCuadrado, Main::menos1, Main::div2)
                       .collect(Collectors.toList())
       );
  }
}
```

Salida esperada:

```
[3, 5, 4, 6]
[0.0, 1.5, 7.5, 12.0]
```

Convocatoria Extraordinaria

Análisis y Diseño de Software (2014/2015)

Responde a cada apartado en hojas separadas

Solución:

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

public class StreamUtil {
  // Nótese lo importante que es la concordancia entre los parámetros genéricos
  // Véase también la solución alternativa más abajo.
  static public <T,V,R> Stream<R> map2(Stream<T> stream,
                                       Function<T,V> f1, Function<V,R> f2) {
            return stream.map(f1).map(f2);
  }

  static public <T> Stream<T> mapN(Stream<T> stream, Function<T,T>... fs) {
            for (Function<T,T> f : fs) {
                  stream = stream.map(f);
            }
            return stream;
      }
} // end StreamUtil
```

Solución alternativa (con expresiones lambda y explicando de los parámetros genéricos)

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

public class StreamUtil {
  // Nótese lo importante que es la concordancia entre los parámetros genéricos
  // El método map2 toma como entrada un stream de tipo IN y genera como
  // resultado otro stream de tipo OUT, mediante la aplicación de la
  // función f1 (cuya entrada debe ser de tipo IN) seguida de la aplicación de
  // la función f2 (cuya salida debe ser de tipo OUT). Y además nótese que el
  // tipo del resultado de f1, W, debe ser idéntico al de la entrada de f2
  //
  static public <IN,W,OUT> Stream<OUT> map2(Stream<IN> stream,
                                            Function<IN,W> f1, Function<W,OUT> f2){
           // implementación usando una expresión lambda
            return stream.map(k -> f2.apply(f1.apply(k)));
  }

  // En este caso, la concordancia entre los parámetros genéricos también es
  // importante aunque mas simple: todos streams (de entrada y salida) son del
  // mismo tipo base, T, también igual al de entrada y salida de las funciones
  static public <T> Stream<T> mapN(Stream<T> stream, Function<T,T>... fs) {
            // implementación usando una expresión lambda
            return stream.map(x -> {
                  T resultado = x;  // si no hay ninguna fs, el resultado es x
                  for (Function<T,T > f : fs) {
                        resultado = f.apply(resultado);
                  }
                  return resultado;
            });
  }
} // end StreamUtil
```