EVALUACIÓN CONTINUA

Análisis y Diseño de Software (2017/2018)

NO SE PERMITEN PREGUNTAS DURANTE EL EXAMEN

Limita tus respuestas a lo que dice estrictamente el enunciado. Es caso de que tengas que hacer alguna suposición sobre algún detalle que no incluye el enunciado, hazlo de manera fundamentada, razonable y sin contradecir el enunciado. Documenta tales suposiciones en tu respuesta.

RESPONDE A CADA APARTADO EN HOJAS SEPARADAS

Apartado 1. (2,5 puntos)

Nos piden diseñar una aplicación para gestionar las tarifas de precios de una empresa de fabricación de material eléctrico. La empresa fabrica distintos productos, y conserva todas las tarifas por motivos históricos, aunque solo tiene una tarifa en vigor en cada momento.

Cada tarifa tiene una fecha de inicio y de finalización, que permite saber el periodo de tiempo en el que estuvo en vigor. Las tarifas tienen, para cada producto que aparezca en la misma, el precio de venta recomendado por unidad, y el número mínimo de unidades que se pueden comprar del mismo.

Los productos tienen un nombre, y pueden ser sencillos o compuestos. Los productos compuestos están formados por varios productos sencillos y/o compuestos. Un producto compuesto puede contener productos repetidos. De esta forma NO es necesario especificar el número de unidades de cada uno que forman el producto compuesto.

De todos los productos hemos de poder conocer su coste de fabricación. En los sencillos el coste de fabricación se especifica al crear el producto. En los compuestos el coste se calcula sumando al coste de los productos de los que está formado, el coste de montaje. Al crear el producto compuesto se especifica el coste de montaje, y los productos de los que está compuesto. Una vez creado un producto sus datos o composición no se podrá cambiar.

Se pide:

(a) Representar el diagrama de clases en UML para el fragmento de aplicación descrito arriba, con todos los métodos excepto *getters* sencillos, que simplemente devuelvan el valor del atributo correspondiente. Tampoco hace falta incluir los constructores.[2 puntos]

(b) ¿Qué patrones de diseños has utilizado y qué clases de tu diseño se corresponden con ellos? [0,5 puntos]

EVALUACIÓN CONTINUA

Análisis y Diseño de Software (2017/2018)

Responde a cada apartado en hojas separadas

Apartado 2. (2,5 puntos)

Debes implementar un sistema que a partir de unos términos de búsqueda acceda a categorías de productos, que a su vez almacenan los nombres o direcciones web de sus anunciantes (representados por cadenas de caracteres). Los términos de búsqueda son palabras, que para simplificar asumimos que siempre están en minúsculas.

Cada término podrá tener asociada una categoría como máximo. Si un término ya está asociado con una categoría y se asocia después con otra, la más reciente reemplazará a la anterior. El método `asociar` permite asociar una categoría a cada uno de los términos de búsqueda dados como parámetros. Esta operación se puede repetir varias veces para asociar una categoría a más términos.

A las categorías se les pueden añadir varios anunciantes uno a uno (método `addAnunciante`). Los anunciantes de una categoría se obtienen uno a uno con sucesivas invocaciones al método `getAnunciante`. Su primera invocación devuelve el primer anunciante que se añadió a la categoría. Cada vez que se obtiene un anunciante, éste se debe colocar automáticamente al final, de forma que se vaya rotando sobre el orden en que se añadieron.

En previsión de que en el futuro se puedan crear otros tipos de categorías, se ha definido la siguiente interfaz:

```
public interface Categoria {
   String getNombre();
   String getAnunciante();
   void addAnunciante(String a);
}
```

Por último, la aplicación tendrá un método, que dado un término de búsqueda, devuelva un anunciante de la categoría asociada, con la rotación de anunciantes descrita arriba, o `null` si no hay categoría asociada a ese término, o la categoría asociada no tenía anunciantes.

No es necesario evitar anunciantes o categorías repetidas. Tampoco es necesario implementar un método para eliminar anunciantes, eliminar categorías, o reducir los términos o anunciantes asociados a una categoría.

Se pide:

a) Implementar las clases necesarias para que el siguiente código produzca la salida que se indica en los comentarios. Se valorará específicamente la calidad del diseño, el uso correcto de conceptos de orientación a objetos y la concisión del código [2,25p]

b) ¿Qué patrón(es) de diseño has utilizado? Explica los roles de las clases, interfaces y métodos en los patrones usados [0,25p]

```
public static void main(String[] args) {
   GestorAnunciantes app = new GestorAnunciantes();
   Categoria coches = app.creaCategoria("coches");
   Categoria casas = app.creaCategoria("casas");
   app.asocia(coches, "taxi", "4x4");
   app.asocia(coches, "motor");
   app.asocia(casas, "alquiler", "apartamento", "piso", "casa");
   casas.addAnunciante("idealista.com");
   coches.addAnunciante("toyota.es");
   coches.addAnunciante("seat.es");
   System.out.println(app.getAnunciante("taxi"));  // Imprime toyota.es
   System.out.println(app.getAnunciante("4x4"));   // seat.es, ya que toyota.es paso al final
   System.out.println(app.getAnunciante("motor")); // De nuevo toyota.es
   System.out.println(app.getAnunciante("notas")); // null, no existe el término
   System.out.println(app.getAnunciante("piso"));  // idealista.com
}
```

EVALUACIÓN CONTINUA

Análisis y Diseño de Software (2017/2018)

Responde a cada apartado en hojas separadas

Apartado 3. (2,5 puntos)

Se quiere asignar a distintos ingenieros la resolución de *errores de ejecución* detectados en algunos de nuestros programas. Para ello se crea un equipo de ingenieros indicando los nombres de sus miembros, y sólo a ellos se les podrá asignar la resolución de errores. Se lanzará una excepción si se intenta asignar un error a un ingeniero que no pertenezca al equipo. A cada ingeniero del equipo se le puede asignar la resolución de un número variable de errores *sin duplicados*.

Cada error de ejecución se identifica mediante el nombre del archivo, el tipo de error detectado (que puede ser excepción, bucle infinito, resultado incorrecto o formato de salida), y el número de línea de dicho archivo donde se localiza el error. En un mismo archivo no pueden existir varios errores detectados en el mismo número de línea (aunque sean de distinto tipo), es decir, si se intenta asignar a un ingeniero un error con el mismo nombre de archivo y mismo número de línea que otro error que ya tenga asignado ese ingeniero, se ignora ese intento de asignación y se imprime un aviso de error con el error duplicado y no añadido.

Se pide: Completa el siguiente programa con las clases, enumerados y excepciones necesarias para que produzca la salida de más abajo, donde las asignaciones de errores al equipo se imprimen ordenadas, en primer lugar, por nombre de ingeniero, y para cada ingeniero sus errores asignados aparecen ordenados por nombre de archivo y dentro de cada archivo por número de línea. Además, nótese que al imprimir cada error, entre paréntesis se indica el nivel de gravedad del error: 10 para formato de salida, 30 para resultado incorrecto, 60 para bucle infinito, y 90 para excepción.

```
public enum TipoError {
        FORMATO_SALIDA(10), RESULTADO_INCORRECTO(30), BUCLE_INFINITO(60), EXCEPCION(90);
        private int nivel;
        private TipoError(int s) { this.nivel = s; }
        public String toString() { return "nivel->"+ this.nivel; }
}
public class Main {
  public static void main(String[] args) {
     Equipo eq = new Equipo("Jon", "Ana", "Leo");
     try {
        eq.reparaError("Leo", new Error("fileX", TipoError.EXCEPCION, 21));
        eq.reparaError("Leo", new Error("fileX", TipoError.BUCLE_INFINITO, 21)); // error duplicada
        eq.reparaError("Ana", new Error("fileX", TipoError.EXCEPCION, 21));//fileX:21, otro ingeniero
        eq.reparaError("Ana", new Error("fileX", TipoError.RESULTADO_INCORRECTO, 69));
        eq.reparaError("Ana", new Error("fileY", TipoError.FORMATO_SALIDA, 155));
        eq.reparaError("Mar", new Error("fileY", TipoError.RESULTADO_INCORRECTO, 100));
        eq.reparaError("Leo", new Error("fileZ", TipoError.RESULTADO_INCORRECTO, 9));// no se ejecuta
     } catch (IngenieroIncorrecto ex) {
        System.out.println(ex);
     }
     System.out.println("Equipo: " +  eq);
  }
}
```

Salida esperada:

```
Error fileX:21(nivel->60) duplicado no asignado a Leo
Error asignado a ingeniero/a Mar fuera del equipo
Equipo: {Ana=[fileX:21(nivel->90), fileX:69(nivel->30), fileY:155(nivel->10)],
Leo=[fileX:21(nivel->90)]}
```

EVALUACIÓN CONTINUA

Análisis y Diseño de Software (2017/2018)

Responde a cada apartado en hojas separadas

Apartado 4. (2,5 puntos)

Al procesar un *stream* es muy frecuente hacer filtros elementales para combinarlos en secuencia (como en s.filter(f1),filter(f2)...) y eso equivale a hacer la conjunción lógica de los filtros. En cambio, en este ejercicio te pedimos que añadas a la clase `STools` un método `filterOr` que, teniendo como parámetros un *stream*, `s`, y un número indeterminado de filtros, devuelva un *stream* que contiene solo los elementos de `s` que cumplen al menos uno de los filtros dados, es decir, aplica la disyunción de filtros al *stream* `s`, como se ve en el ejemplo de abajo.

Además, surge la idea de que cualquier conjunto también podría ser empleado para componer filtros como se ha hecho arriba. Es decir, un conjunto visto como filtro acepta o deja pasar solamente aquellos elementos que contiene. Así pues, como nos interesan los filtros disyuntivos, debes añadir a la clase `STools` un método `filterSets` que, como se ve en el ejemplo, es similar al método del párrafo anterior *pero acepta cualquier tipo de conjuntos como filtros.* Nótese que, en el ejemplo, "Ja" sale dos veces porque está dos veces en la entrada y el conjunto se usa solamente para filtrar, no para almacenar las palabras.

Bien pensado, si alguien prevé usar su conjunto como filtro, podría crear un tipo especial de filtro, que fuese combinable, en disyunción, no solo con otros filtros de su tipo, sino con cualquier otro filtro que sea aceptado por el método `filterOr` que se describió en el primer párrafo. Es decir, define la clase `TestSet` que se usa en el programa ejemplo, para que se pueda pasar como argumento al método `filterOr` de forma que el programa de abajo (tercera línea de salida) pueda combinar disyuntivamente filtros de varios tipos.

Se pide:

Implementar las clases e interfaces necesarias para que el código dado produzca la salida de abajo. Se valorará específicamente la calidad del diseño, la flexibilidad en el uso de genéricos y la concisión del código. [2p]

```
public class Main {
   public static boolean longWord(String s) { return s.length() > 4; }

   public static void main(String[] args) {
      Collection<String> pal = Arrays.asList("Ja", "Ja", "Reírse", "es", "algo", "saludable");

      Stream<String> ss = STools.filterOr(pal.stream(),
                                          s -> s.startsWith("J"),
                                          s -> s.length() > 4);
      System.out.println(ss.collect(Collectors.toList()));

      System.out.println(STools.filterSets(pal.stream(),
                                           Set.of("Ja", "Je", "Jo"),
                                           Collections.emptySet(),
                                           Set.of("es", "no") )
                               .collect(Collectors.toList()));

      System.out.println(STools.filterOr(pal.stream(),
                                          new TestSet<>(Set.of("Ja", "Je", "Jo")),
                                          Main::LongWord,
                                          new TestSet<>(Set.of("es", "no")) )
                               .collect(Collectors.toList()));
   }
}
```

Salida esperada:

```
[Ja, Ja, Reírse, saludable]
[Ja, Ja, es]
[Ja, Ja, Reírse, es, saludable]
```

Prueba 2 de Evaluación Continua (2017/18)
Análisis y Diseño de Software

Apartado 1 (5 puntos)

Se pide: Completar las secciones 1, 2 y 3 con un buen diseño, para que la salida de la ejecución del siguiente código sea la indicada abajo.

Las entradas tienen descripción, tipo de evento, precio base, y un precio final. Excepto el precio final, el resto de datos se indican al crear las entradas. Las entradas preferentes, de forma opcional, pueden además tener un código de reserva.

El precio final se calcula a partir del precio base, aplicando una serie de descuentos o recargos. Los descuentos o recargos están definidos como porcentajes y son acumulables, de forma que el primero se calcula sobre el precio base, y los siguientes sobre el precio resultante de aplicar los anteriores, aunque el orden de aplicación no afecta al resultado. Existen los siguientes descuentos/recargos :

- En todas las entradas, si es un evento social, el precio final tendrá un descuento del 50%.
- En las entradas preferentes el precio final sufre un recargo adicional del 10% cuando no hay reserva, y del 20% cuando sí la hay.

Nota: No es necesario añadir las declaraciones package o import.

```
public enum Evento {  MUSICAL, DEPORTIVO, SOCIAL;  }
```

```
public class Entrada implements Comparable<Entrada>{
  private final String descripcion;
  private final Evento evento;

  public Evento getEvento() { return evento; }
  public String getDescripcion() { return descripcion; }
  public String toString() {
    return descripcion +" : "+ precioBase() +" -> "+ precioFinal();
  }

 // COMPLETAR LA CLASE Entrada en hoja de solución (Sección 1)
 // añadiendo los atributos, métodos y/o constructores necesarios

} // fin class Entrada
```

```
// DEFINIR LA CLASE EntradaPreferente en hoja de solución
// (Sección 2)
```

```
public class Ejercicio1 {
  public static void main(String[] args) {
    Entrada e1 = new Entrada("A Rock & River", 80, Evento.MUSICAL);
    Entrada e2 = new Entrada("Mitín Electoral", 10, Evento.SOCIAL);
    Entrada e3 = new EntradaPreferente("The Globetrotters", 40.0, Evento.DEPORTIVO);
    Entrada e4 = new EntradaPreferente("Cancer Day", 50.0, Evento.SOCIAL, "RX-69/18");

 // COMPLETAR declaración de variable x  (Sección 3)

    x.add(e1);  x.add(e2);  x.add(e3);  x.add(e4);  x.add(e2);  x.add(e4);

    System.out.println(x); //Se imprimen sin repetidos y ordenados por descripción
  }
}
```

Salida esperada:

```
[A Rock & River : 80.0 -> 80.0, Cancer Day : 50.0 -> 30.0, Mitin Electoral : 10.0 ->
5.0, The Globetrotters : 40.0 -> 44.0]
```

Prueba 2 de Evaluación Continua (2017/18)
Análisis y Diseño de Software

Apartado 2 (5 puntos)

Los objetos de la clase `Operandos` se configuran con una serie de tamaño variable de números enteros, y tienen un método para `aplicar` secuencialmente un número variable (mayor o igual que cero) de *transformadores elementales* que recibe como parámetros. Este método devuelve la serie de enteros resultante de aplicar dichas transformaciones. Todos los objetos *transformadores elementales* tienen un método `Integer transformar (Integer x)` que transforma un entero en otro, según corresponda al significado de dicho transformador; p.ej.: elevar al cuadrado, módulo N (o resto división por N), etc.

Se pide: Diseñar y programar el código Java necesario para que la salida resultante del siguiente programa sea la que se muestra más abajo, de acuerdo con las especificaciones anteriores.

```
public class Ejercicio2 {
  public static void main(String... args) {
    Operandos op = new Operandos(-4, 7, -2, 8);

    System.out.println(op.aplicar() );
    System.out.println(op.aplicar( new Cuadrado() ) );
    System.out.println(op.aplicar( new Cuadrado(), new Modulo(10), new Cuadrado() ) );
  }
}
```

Salida esperada:

```
[-4, 7, -2, 8]
[16, 49, 4, 64]
[36, 81, 16, 16]
```