Prueba 2 de Evaluación Continua

Análisis y Diseño de Software (2021/2022)

Entrega cada ejercicio en hojas separadas
No se permiten preguntas durante el examen

Ejercicio 1 ***(4.0 puntos)***

Se va a desarrollar una nueva plataforma de servicio de *streaming* bajo demanda donde los abonados podrán comprar los vídeos que desea visionar. De cada abonado, la plataforma dispondrá de su nombre de usuario, el plan de precios al que está subscrito y la lista de vídeos que ha comprado. Los planes de precios pueden ser Básico (que no tiene tarifa mensual y en el que se paga sólo por los productos que se compran), Estándar (que tiene una tarifa mensual de 24.99€, lo que da derecho a contar con un descuento del 50% sobre todo lo que compre), o Premium (que tiene una tarifa mensual de 49.99€, lo que da derecho a todo el contenido sin tener que pagar nada adicional). El sistema debe registrar los vídeos comprados por los abonados y calcular el adeudo a pagar por este. El adeudo debe contemplar todo lo comprado por el abonado más la cuota de subscripción según su plan. Los vídeos tienen un título, un identificador único y un precio. Por el momento, estos vídeos serán solo de dos tipos: películas o series. Para fomentar campañas de marketing, las películas pueden contar con un descuento de promoción. Por su parte, el precio de las series se proporciona por capítulo, aunque el abonado paga por toda la serie al completo. Así, para conocer el coste total que pagará el abonado por la serie, se debe conocer el número de temporadas y el número de capítulos por temporada (por simplificar, se asume que todas las temporadas tienen el mismo número de capítulos).

Se pide: el código Java necesario para modelar el sistema descrito arriba, de forma que al ejecutar el siguiente código de prueba se produzca la salida mostrada abajo. No escribas métodos innecesarios para esta ejecución, pero presta especial atención al diseño de las clases necesarias y las relaciones entre ellas y entres sus métodos.

```
package uam.eps.acme.streaming;

public class Main {
   public static void main(String[] args) {
      Abonado pedro = new Abonado("Pedro", Plan.BASICO);
      Abonado pablo = new Abonado("Pablo", Plan.ESTANDAR);
      Abonado vilma = new Abonado("Vilma", Plan.PREMIUM);

      Vídeo batman = new Película("The Batman", 4.99, 0.15); // precio de 4,99€, y descuento de 15% por promoción
      Vídeo morbius = new Película("Morbius", 4.99);  // precio de 4,99€
      Vídeo calamar = new Serie("El juego del calamar", 0.99, 1, 9);// 0,99€ por capítulo, 1 temporadas, 9 cap. por temporada
      Vídeo stranger = new Serie("Stranger Things", 0.99, 3, 8); // 0,99€ por capítulo, 3 temps. y 8 capítulos por temporada
      Vídeo animales = new Película("Animales fantásticos: Los secretos de Dumbledore", 4.99); // precio de 4,99€

      pedro.comprar(stranger)
           .comprar(calamar)
           .comprar(animales);
      vilma.comprar(stranger)
           .comprar(morbius)
           .comprar(batman);
      pablo.comprar(morbius)
           .comprar(calamar)
           .comprar(stranger);

      System.out.println(pedro);
      System.out.println(pablo);
      System.out.println(vilma);
   }
}
```

SALIDA:

```
Pedro compró:
[
Serie ['Stranger Things', id: 4, precio: 23,76€, temporadas: 3, capítulos por temporada: 8],
Serie ['El juego del calamar', id: 3, precio: 8,91€, temporadas: 1, capítulos por temporada: 9],
Película ['Animales fantásticos: Los secretos de Dumbledore', id: 5, precio: 4,99€]]
plan: BASICO; adeudo: 37,66€
Pablo compró:
[
Película ['Morbius', id: 2, precio: 4,99€],
Serie ['El juego del calamar', id: 3, precio: 8,91€, temporadas: 1, capítulos por temporada: 9],
Serie ['Stranger Things', id: 4, precio: 23,76€, temporadas: 3, capítulos por temporada: 8]]
plan: ESTANDAR; adeudo: 43,82€
Vilma compró:
[
Serie ['Stranger Things', id: 4, precio: 23,76€, temporadas: 3, capítulos por temporada: 8],
Película ['Morbius', id: 2, precio: 4,99€],
Película ['The Batman', id: 1, precio: 4,99€]]
plan: PREMIUM; adeudo: 49,99€
```

Solución: 0,6 enum + 0,85 x 4 clases = 4,0 puntos

```
package uam.eps.acme.streaming;

public enum Plan {
	BASICO(0.00, 0.00), ESTANDAR(0.50, 24.99), PREMIUM(1.00, 49.99);

	private double descuento;
	private double tarifa;

	private Plan(double descuento, double tarifa) {
		this.descuento = descuento;
		this.tarifa = tarifa;
	}

	public double descuento() {
		return this.descuento;
	}

	public double tarifa() {
		return this.tarifa;
	}
}
```

```
package uam.eps.acme.streaming;

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.List;

public class Abonado {
	private String usuario;
	private Plan plan;
	private List<Video> visionados;

	public Abonado(String usuario, Plan plan) {
		this.usuario = usuario;
		this.plan = plan;
		this.visionados = new ArrayList<Video>();
	}

	public Abonado comprar(Video v) {
		if (v != null)
			this.visionados.add(v);

		return this;
	}

	public double adeudo() {
		double saldo = plan.tarifa();

		for (Video v : visionados) {
			saldo += v.getPrecio() * (1 - plan.descuento());
		}

		return saldo;
	}

	@Override
	public String toString() {
		DecimalFormat df = new DecimalFormat(); // No necesario en la solución
		df.setMaximumFractionDigits(2);         // solo ayuda a la salida por consola

		return usuario + " compró: \n" + this.visionados + "\n" +
			   "plan: " + plan + "; " +
			   "adeudo: " + df.format(adeudo()) + "€";
	}

}
```

```java
package uam.eps.acme.streaming;

import java.text.DecimalFormat;

public abstract class Video {
	private String titulo;
	private static int generaId = 0;
	private int idVideo;
	private double precio;

	public Video(String titulo, double valor) {
		this.titulo = titulo;
		this.idVideo = ++generaId;
		this.precio = valor;
	}

	public int getIdMedia() {
		return idVideo;
	}

	public double getPrecio() {
		return precio;
	}

	@Override
	public String toString() {
		DecimalFormat df = new DecimalFormat();
		df.setMaximumFractionDigits(2);

		return "'" + título + "', " +
		    "id: " + idVideo + ", " +
			    "precio: " + df.format(precio) + "€";
	}
}
```

```java
package uam.eps.acme.streaming;

public class Pelicula extends Video{
	private double promo;

	public Pelicula(String titulo, double precio) {
		super(titulo, precio);
	}

	public Pelicula(String titulo, double precio, double promo) {
		this(titulo, precio);
		this.promo = promo;
	}

	@Override
	public double getPrecio() {
		return super.getPrecio() * promo;
	}

	@Override
	public String toString() {
		return "\nPelícula [" + super.toString() + "]";
	}
}
```

```java
package uam.eps.acme.streaming;

public class Serie extends Video{
	int capitulos;
	int temporadas;

	public Serie(String titulo, double precio, int temporadas, int capitulos) {
		super(titulo, precio * temporadas * capitulos);
		this.temporadas = temporadas;
		this.capitulos = capitulos;
	}

	@Override
	public String toString() {
		return "\nSerie [" +
		    super.toString() + ", " +
			    "temporadas: " + temporadas + ", " +
			    "capítulos por temporada: " + capitulos + "]";
	}
}
```

Prueba 2 de Evaluación Continua

Análisis y Diseño de Software (2021/2022)

Entrega cada ejercicio en hojas separadas

Ejercicio 2 ***(3,5 puntos)***

Considérese una clase `Movie` que almacena y da acceso al título y año de estreno (mayor o igual que 1900) de una película.

Se ha implementado un buscador que, dada una lista de películas, devuelve aquellas que cumplen un criterio dado. Un criterio puede ser simple o compuesto. Un criterio simple requiere que el título o año de una película satisfaga cierta restricción, mientras que un criterio compuesto permite agregar dos criterios (simples o compuestos) mediante operadores Booleanos AND y OR.

El siguiente código muestra por pantalla películas cuyos títulos tienen la palabra clave (*keyword*) "star" o la palabra clave "space", y cuyos años de estreno están entre 1968 y 2000 (ambos incluidos):

```
SearchCriterion criterion
                      = new ANDCriterion(
                             new ORCriterion(new TitleCriterion("star"), new TitleCriterion("space")),
                             new YearCriterion(1968, 2000));
List<Movie> movies = Arrays.asList(
                      new Movie("Star Trek", 1966),
                      new Movie("2001: A Space Odyssey", 1968),
                      new Movie("Star Wars", 1977),
                      new Movie("Starship", 1984),
                      new Movie("The Matrix", 1999),
                      new Movie("Battle in Space", 2021));
List<Movie> results = criterion.satisfy(movies);
System.out.println(results);
```

Salida:

```
[(Star Wars, 1977), (2001 - A Space Odyssey, 1968)]
```

En particular, a partir de la siguiente interfaz:

```
public interface SearchCriterion {
       public List<Movie> satisfy(List<Movie> movies) throws EmptyCollectionException;
}
```

se implementan los criterios simples `TitleCriterion` y `YearCriterion`, y los criterios compuestos `ANDCriterion` y `ORCriterion`.

Por otra parte, se considera que en el uso del buscador pueden darse errores: al construir un criterio con algún argumento incorrecto (`InvalidArgumentException`) y al satisfacer un criterio sobre una lista de películas vacía (`EmptyCollectionException`). Para ello, se establece una clase `SearchException` general que guarda la razón (*reason*) del error:

```
private String reason;
public SearchException(String reason) {
       this.reason = reason;
}
public String toString() {
       return "Error en el buscador de películas. " + this.reason + ".\n";
}
```

Y dos clases concretas:

- `InvalidArgumentException`, que especifica como razón de error "`Argumento no válido: <reason>`", donde `<reason>` debe reemplazarse por "`Palabra clave nula`", "`Año mínimo menor que 1900`" o "`Año máximo menor que <mínimo>`" en las clases que correspondan.
- `EmptyCollectionException`, que especifica como razón de error "`Colección de películas vacías`".

Se pide:

a) La declaración de la clase `SearchException` (no proporcionar el código). (**0,5 puntos**)

b) El código Java de las clases `InvalidArgumentException` y `EmptyCollectionException`. (**1 punto**)

c) El código Java de las clases `TitleCriterion` y `YearCriterion`, con el tratamiento de excepciones adecuado. (**2 puntos**)

Solución:

a)

```
public abstract class SearchException extends Exception
```

b)

```
public class InvalidArgumentException extends SearchException {
    public InvalidArgumentException(String argument) {
        super("Argumento no válido: " + argument);
    }
}

public class EmptyCollectionException extends SearchException {
    public EmptyCollectionException() {
        super("Colección de películas vacías");
    }
}
```

c)

```
public class TitleCriterion implements SearchCriterion {
    private String keyword;

    public TitleCriterion(String keyword) throws InvalidArgumentException {
        if (keyword == null) throw new InvalidArgumentException("Palabra clave nula");
        this.keyword = keyword;
    }

    @Override
    public List<Movie> satisfy(List<Movie> movies) throws EmptyCollectionException {
        if (movies == null || movies.isEmpty()) throw new EmptyCollectionException();

        List<Movie> results = new ArrayList<>();
        for (Movie movie: movies) {
            StringTokenizer tokenizer = new StringTokenizer(movie.getTitle(), " ");
            while (tokenizer.hasMoreTokens()) {
                String token = tokenizer.nextToken();
                if (token.equalsIgnoreCase(keyword.toLowerCase())) results.add(movie);
            }
        }
        return results;
    }
}
public class YearCriterion implements SearchCriterion {
    private int minimum;
    private int maximum;

    public YearCriterion(int minimum, int maximum) throws InvalidArgumentException {
        if (minimum < 1900) throw new InvalidArgumentException("Año mínimo menor que 1900.");
        if (maximum < minimum) throw new InvalidArgumentException("Año máximo menor que " + minimum + ".");

        this.minimum = minimum;
        this.maximum = maximum;
    }

    @Override
    public List<Movie> satisfy(List<Movie> movies) throws EmptyCollectionException {
        if (movies == null || movies.isEmpty()) throw new EmptyCollectionException();

        List<Movie> results = new ArrayList<>();
        for (Movie movie: movies) {
            if (movie.getYear() >= minimum && movie.getYear() <= maximum) {
                results.add(movie);
            }
        }
        return results;
    }
}
```

Prueba 2 de Evaluación Continua

Análisis y Diseño de Software (2021/2022)

Entrega cada ejercicio en hojas separadas
No se permiten preguntas durante el examen

Ejercicio 3 ***(2.5 puntos)***

Se quiere desarrollar una clase genérica `BiMap` que permita almacenar asociaciones entre 3 tipos de objetos. Por ejemplo, `BiMap<String,Person,Integer>` permitiría almacenar asociaciones entre un `String` (por ejemplo, con nombres de ciudades), un objeto `Person` (personas que vivan en dicha ciudad), y un `Integer` (por ejemplo, la edad de las personas). Los objetos `BiMap` deben ser compatibles con la interfaz `Map` y almacenarán los datos por el orden natural del primer objeto, y luego por orden de inserción del segundo objeto. Por tanto, debes evitar la creación de `BiMaps` si el primer tipo no define un orden natural.

Adicionalmente, `BiMap` define un método llamado `fold`, que recibe un objeto conforme a la siguiente interfaz:

```
public interface IOperation<T1, T2> {
        T2 init();
        T2 accumulate(T1 element);
}
```

El método fold debe utilizar su objeto de entrada `IOperation` para generar un `Map` que "resuma" la información que se ha ido añadiendo en el `BiMap`. En el ejemplo anterior, podríamos usar `fold` para obtener un `Map<String,Double>` con la media de las edades de las personas que residen en cada ciudad. Para ello, se pasará a `fold` un objeto `IOperation<Integer,Double>` que calcula la media de manera iterativa (esto es, init() devuelve 0, y luego accumulate(`element`) devuelve la media de los valores acumulados, incluido `element`).

El siguiente listado muestra un ejemplo de uso de `BiMap`, con un hueco que debes rellenar, y la salida esperada.

```
public static void main(String[] args) {
   // cada persona está definida por un nombre, ciudad de residencia y edad
   Person[] people = {   new Person("Luca Bottoni", "Rome", 14),   new Person("Ana Sanchez", "Madrid", 35),
                         new Person("Pierre Artaud", "Paris", 75), new Person("Luisa Fernandez", "Madrid", 43) };
   BiMap<String,Person,Integer> population = new BiMap<>();

   for (Person p: people)
        population.put(p.getCity(), p, p.getAge());

   System.out.println(population);

   Map<String, Map<Person, Integer>> map = population; // compatible con Map
   Map<Person, Integer> madrid = population.get("Madrid");
   System.out.println("Madrid: "+ madrid);

   Map<String, Double> avg = population.fold(new IOperation...
        // Completar el código con una clase anónima que calcule la media
   );
   System.out.println(avg);
}
```

Salida Esperada:

```
{Madrid={Ana Sanchez=35, Luisa Fernandez=43}, Paris={Pierre Artaud=75}, Rome={Luca Bottoni=14}}
Madrid: {Ana Sanchez=35, Luisa Fernandez=43}
{Madrid=39.0, Paris=75.0, Rome=14.0}
```

Se pide:

(1) **[2.25 puntos]** Usando principios de orientación a objetos, desarrolla la clase `BiMap` y completa el código del listado anterior con una clase anónima que implemente la media de las edades. Asume que existe una clase `Person` con el constructor de tres parámetros que se utiliza en el código, así como los métodos `getCity` (que devuelve un `String` con la ciudad de la persona), `getAge` (que devuelve un `int` con la edad de la persona) y un método `toString`.

(2) **[0.25 puntos]** ¿Qué deberíamos añadir a la clase `Person` si quisiéramos evitar que el `BiMap` añadiera dos veces a dos personas con el mismo nombre que residen en la misma ciudad? (no es necesario que escribas código, sólo que indiques qué métodos habría que añadir y por qué).

Solución (1):

```
public class BiMap<T1 extends Comparable<T1>, T2, T3> extends TreeMap<T1, Map<T2, T3>>{

	public <S> Map<T1, S> fold(IOperation<T3, S> operation) {
		Map<T1, S> folded = new TreeMap<>();
		for (T1 pkey: this.keySet()) {
			S ret = operation.init();
			for (T2 skey: this.get(pkey).keySet())
				ret = operation.accumulate(this.get(pkey).get(skey));
			folded.put(pkey, ret);
		}
		return folded;
	}

	public T3 put(T1 pkey, T2 skey, T3 value) {  // the method can just return void, but this is inline with Map.put
		this.putIfAbsent(pkey, new LinkedHashMap<>());
		Map<T2, T3> map = this.get(pkey);
		return map.put(skey, value);
	}

}

// Clase anónima en el listado:

new IOperation<Integer, Double>() {
			private long sum = 0;
			private int num = 0;

			@Override
			public Double init() {
				this.sum = 0;
				this.num = 0;
				return 0.0;
			}

			@Override
			public Double accumulate(Integer element) {
				sum += element;
				num++;
				return sum*1.0/num;
			}
		}
```

Solución (2): Habría que sobreescribir los métodos equals y hashCode (considerando iguales a las personas con igual nombre y ciudad), ya que los objetos Person se utilizan como clave en un LinkedHashMap.