Examen Final de Junio (Evaluación Continua)

Análisis y Diseño de Software (2010/2011)

Contesta cada apartado en hojas separadas

Apartado 1. (3 puntos)

Se quiere hacer una aplicación muy sencilla para el almacenamiento de cócteles. La aplicación debe contemplar tanto cócteles oficiales del IBA (*International Bartenders Association*), como cualquier otro tipo de cóctel. Mientras que los cócteles oficiales del IBA sólo pueden ser de los tipos: `seco, dulce, collins o fancy`, los otros tipos de cócteles admiten cualquier clasificación (véase por ejemplo el tipo enumerado `MisCocteles`). Para simplificar, los cócteles estarán descritos por un nombre y un tipo. Además, se debe poder añadir ingredientes a cualquier tipo de cóctel con el método `addIngredient`. La aplicación debe tener un mecanismo para poder obtener todos los cócteles oficiales del IBA que se hayan creado previamente. Completa el siguiente programa para que la ejecución de la clase *Main* produzca la salida que se indica más abajo.

Fichero TipoCocktail.java

```
public interface TipoCocktail {
       String getDescripcion();
}
```

Fichero Main.java

```
import java.util.List;

enum MisCocteles implements TipoCocktail {
  BURBUJAS, CLASICO, CONTEMPORANEO, SINALCOHOL;

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

public class Main {
  public static void main(String[] args) {
    Cocktail<MisCocteles> amarettine = new Cocktail<MisCocteles>("Amaretine", MisCocteles.BURBUJAS);
    IBACocktail kir = new IBACocktail("Kir", TipoIBACocktail.SECO);
    new IBACocktail("Caipirinha", TipoIBACocktail.FANCY);

    amarettine.addIngredient("amaretto", "media medida").
               addIngredient("vino blanco espumoso", "al gusto").
               addIngredient("vermut seco", "media medida");

    kir.addIngredient("vino blanco", "9 partes").addIngredient("crema de Cassis", "1 parte");

    for (TipoCocktail t : TipoIBACocktail.values()) {
       try {
              List<IBACocktail> lista = IBACocktail.allCocktails(t);
              System.out.println(lista);
       } catch (NoCocktailsException e) {
              System.out.println("Error! no se han creado cocteles de tipo: "+e.getTipo());
       }
    }
    System.out.println(amarettine);
  }
}
```

Salida

```
[Kir[SECO]:({crema de Cassis=1 parte, vino blanco=9 partes})]
Error! no se han creado cocteles de tipo: DULCE
Error! no se han creado cocteles de tipo: COLLINS
[Caipirinha[FANCY]:({})]
Amaretine[BURBUJAS]:({amaretto=media medida, vermut seco=media medida, vino blanco espumoso=al gusto})
```

Nota:

- Debe ser posible parametrizar la clase `Cocktail` con cualquier clase que implemente la interfaz `TipoCocktail`.
- En la salida, los ingredientes se muestran por orden alfabético. Si no se han introducido ingredientes (como en el caso de la Caipirinha) se imprime el cóctel con los ingredientes vacios.
- Como recordarás, el método estático `values()` sobre un tipo enumerado devuelve un array con todos los objetos enumerados definidos.

Una posible solución sería la siguiente:

Fichero Cocktail.java

```
import java.util.HashMap;
import java.util.Map;

public class Cocktail<T extends TipoCocktail> {
	protected String nombre;
	protected T tipo;
	protected Map<String, String> ingredientes = new TreeMap<String, String>();

	public Cocktail(String n, T t) {
		this.nombre = n;
		this.tipo = t;
	}

	public String toString() {
		return this.nombre+"["+tipo.getDescripcion()+"]:("+this.ingredientes.toString()+")";
	}

	public Cocktail<T> addIngredient(String ingrediente, String medida) {
		this.ingredientes.put(ingrediente, medida);
		return this;
	}
}
```

Fichero IBACocktail.java

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;


enum TipoIBACocktail implements TipoCocktail {
	SECO, DULCE, COLLINS, FANCY;
	@Override
	public String getDescripcion() { return this.toString(); }
}

public class IBACocktail extends Cocktail<TipoIBACocktail> {
	public static Map<TipoIBACocktail,List<IBACocktail>> all = new
HashMap<TipoIBACocktail,List<IBACocktail>>();

	public IBACocktail(String n, TipoIBACocktail t) {
		super(n, t);
		if (!all.containsKey(t)) all.put(t, new ArrayList<IBACocktail>());
		all.get(t).add(this);
	}

	public static List<IBACocktail> allCocktails(TipoCocktail t) throws NoCocktailsException{
		if (!all.containsKey(t))
			throw new NoCocktailsException(t);
		return all.get(t);
	}
}
```

Fichero NoCocktailsException.java

```
public class NoCocktailsException extends Exception {
	private static final long serialVersionUID = -4900837198527441400L;
	private TipoCocktail t;

	public NoCocktailsException(TipoCocktail t) {
		this.t = t;
	}

	public TipoCocktail getTipo() {
		return t;
	}
}
```

Contesta cada apartado en hojas separadas

Apartado 2. (3.5 puntos)

Completa el siguiente programa para que su salida sea la que se indica más abajo. El objetivo es acumular el consumo de ordenador realizado por cada de usuario, en relación con *tiempo de CPU*, *tamaño de las descargas*, y *espacio de disco utilizado*. Tras un periodo de acumulación de consumos, se debe poder calcular el importe a facturar a cada usuario por sus consumos acumulados. Los usuarios se identifican mediante su DNI (sin letra). Además, los usuarios solo pueden ser de dos tipos: *usuario abonado* o *usuario PPU* (de pago por uso), que tienen distintos criterios de facturación. Los abonados no pagarán nada por el espacio de disco utilizado, mientras que los PPU pagarán 0,03€ por unidad de espacio en disco utilizada. Cada abonado se crea con un valor de tamaño de descarga que no se le cobrará, mientras que todos los PPU tienen una descarga gratuita fija de 3000 unidades. Por encima de sus asignaciones de descarga gratuita todos los usuarios pagarán 0,05€ por cada 200 unidades de descarga. Finalmente, la facturación por tiempo de CPU es uniforme para todos los usuarios, a 0,02€ por unidad de tiempo de CPU consumida.

```
public class Apartado2 {

	public static void main(String[] args) {
                        // abonado con 8000 unidades de descarga gratis
		Usuario u1 = new UsuarioAbonado(51123456, 8000);
		Usuario u2 = new UsuarioPPU(54000001);
		Usuario u3 = new UsuarioPPU(53999000);
		Usuario u4 = new UsuarioAbonado(45000222, 4000);
                       // 100 CPUtime, 5000 descarga, 200 espacio en disco
		Consumo c1 = new Consumo(100, 5000, 200);
                       // 1000 CPUtime, 3000 descarga, 100 espacio en disco
		Consumo c2 = new Consumo(1000, 3000, 100);

		Facturador g = new Facturador();
		g.acumular(u1, c1);
		g.acumular(u2, c2);
		g.acumular(u1, c2);
		g.acumular(u3, c2);
		g.acumular(u3, c2);
		g.acumular(u4, c1);

		System.out.println(g);
		g.facturacionPeriodica();
	}
}
```

Salida:

```
{
45000222=Consumo=100:5000:200,
53999000=Consumo=2000:6000:200,
54000001=Consumo=1000:3000:100,
51123456=Consumo=1100:8000:300}

Facturación Periódica por Orden de DNI
45000222 facturado por 2.25
51123456 facturado por 22.0
53999000 facturado por 46.75
54000001 facturado por 23.0
```

Solución al apartado 2:

NI ES ÚNICA SOLUCION, NI ES PERFECTA, PERO SACÓ MUY BUENA NOTA.
Los comentarios que faltan se pueden obtener consultando al profesor.

```
//  archivo Facturador.java
import java.util.ArrayList;
import java.util.List;
import java.util.HashMap;
import java.util.Map;
import java.util.Collections;

public class Facturador {
   public static final double precioUnidadDisco = 0.03;
      public static final double precioUnidadCPU = 0.02;
      public static final double precioPorBloqueDescarga = 0.05;
      public static final double precioTamanyoBloqueDescarga = 200;

      Map<Usuario, Consumo> m = new HashMap<Usuario, Consumo>();

      public void acumular(Usuario u, Consumo c) {
           if (! m.containsKey(u)) {
             m.put(u, new Consumo());
           }
           m.get(u).incrementa(c);
      }
      public void facturacionPeriodica(){
            System.out.print("\nFacturación Periódica");
            List<Usuario> usuarios = new ArrayList<Usuario>(m.keySet());
            Collections.sort(usuarios);
            for (Usuario u : usuarios) {
                  System.out.print(u + " facturado por " + u.facturar(m.get(u)));
            }

      }
      public String toString() { return m.toString(); }
}

//  archivo Usuario.java
public abstract class Usuario implements Comparable<Usuario>{
      private Long dni;
      protected Usuario(long dni) {
            this.dni = dni;
      }
      public double facturar(Consumo c) {
            return costeCpu(c.cpuTime())
                 + costeDownloads(c.downloadSize())
            + costeLocalDiskUsage(c.localDiskUsage());
      }
      public abstract int freeDownloadSize();
      protected double costeCpu(int cpuUnits) {
            return Facturador.precioUnidadCPU * cpuUnits;
      }
      protected double costeDownloads(int downloadUnits){
            double total = 0.0;
            if (downloadUnits > freeDownloadSize()) {
                  total += Facturador.precioPorBloqueDescarga
                              * ((downloadUnits - freeDownloadSize())
                        / Facturador.precioTamanyoBloqueDescarga);
                  if (((downloadUnits - freeDownloadSize())
                        % Facturador.precioTamanyoBloqueDescarga) != 0)
                       total += Facturador.precioPorBloqueDescarga;
              }
            return total;
      }
      protected abstract double costeLocalDiskUsage(int diskUnits);
```

```java
	@Override
	public int compareTo(Usuario u) {
		return dni.compareTo(u.dni);
	}
	public String toString() {
		return "\n" + dni;
	}
}


//  archivo UsuarioAbonado.java
public class UsuarioAbonado extends Usuario {
	private final int freeDownloadSize;
	public UsuarioAbonado(long dni, int freeDownloadSize) {
		super(dni);
		this.freeDownloadSize = freeDownloadSize;
	}
	public int freeDownloadSize() {return freeDownloadSize;}
	protected double costeLocalDiskUsage(int diskUnits) {
		return 0.0;
	}
}

//  archivo UsuarioPPU.java
public class UsuarioPPU extends Usuario {
	private final int freeDownloadSize = 3000;
	public UsuarioPPU(long dni) {
		super(dni);
	}
	public int freeDownloadSize() {return freeDownloadSize;}
	protected double costeLocalDiskUsage(int diskUnits) {
		return Facturador.precioUnidadDisco * diskUnits;
	}
}

//  archivo Consumo.java
public class Consumo {
	private int cpuTime;
	private int downloadSize;
	private int localDiskUsage;

	public Consumo() {
		this.cpuTime = 0;
		this.downloadSize = 0;
		this.localDiskUsage = 0;
	}
	public Consumo(int cpuTime, int downloadSize, int localDiskUsage) {
		this.cpuTime = cpuTime;
		this.downloadSize = downloadSize;
		this.localDiskUsage = localDiskUsage;
	}
	public void incrementa(Consumo c) {
		this.cpuTime += c.cpuTime;
		this.downloadSize += c.downloadSize;
		this.localDiskUsage += c.localDiskUsage;
	}
	public int cpuTime() { return cpuTime; }
	public int downloadSize() { return downloadSize; }
	public int localDiskUsage() { return localDiskUsage; }

	public String toString() {
		return "Consumo=" + cpuTime + ":" + downloadSize + ":" + localDiskUsage;
	}
}
```

Contesta cada apartado en hojas separadas

Apartado 3. (1.5 puntos)

Indica razonadamente la salida que produciría el siguiente programa, tras explicar los errores de compilación que pudiera contener y eliminar las líneas que los produzcan.

```
public class Apartado3 {
	public static void main(String[] args) {
		ClaseA ab = new ClaseB(2);
		ClaseB bb = new ClaseB(4);
		ClaseC cc = new ClaseC(5);
		System.out.println("ab " + ab);
		System.out.println("bb " + bb);

		ab.f(ab);
		ab.f(bb);
		bb.f(cc);
		bb.f(ab);
		bb.f((ClaseB)ab);

		ab.g(cc);
		cc.g(bb);
	}
}

public abstract class ClaseA {
	protected long x;
	public String s;

	public ClaseA(long n) {
		s = "InitN.";
		x = n;
	}
	public abstract void f(ClaseA a);
}

public class ClaseB extends ClaseA {
	private double w;
	public ClaseB(long m) {
		super(m);
		w = m / 2;
		s = s + "B.";
	}
	public void f(ClaseA a) { System.out.println("B.f(A a)" +  a);  }
	public void f(ClaseB b) { System.out.println("B.f(B b)" +  b); }
	public String toString() {return "x:"+ x + " w:"+ w + " -> " + s;}
}

public class ClaseC extends ClaseA {
	private int y;
	public ClaseC(int m) {
		super(7*m);
		y = m;
		s = s + "C.";
	}
	public void f(ClaseA a) { System.out.println("C.f(A a)" +  a); }

	public void g(ClaseB b) { System.out.println("C.g(B b)" +  b); }
	public void g(ClaseC c) { System.out.println("C.g(C c)" +  c); }

	public String toString() {return "x:"+ x + " y:"+ y + " -> " + s;}
}
```

Solución:

Error:

```
ab.g(cc);  // The method g(ClaseC) is undefined for the type ClaseA
```

Salida:

```
ab x:2 w:1.0 -> InitN.B.
bb x:4 w:2.0 -> InitN.B.
B.f(A a)x:2 w:1.0 -> InitN.B.
B.f(A a)x:4 w:2.0 -> InitN.B.
B.f(A a)x:35 y:5 -> InitN.C.
B.f(A a)x:2 w:1.0 -> InitN.B.
B.f(B b)x:2 w:1.0 -> InitN.B.
C.g(B b)x:4 w:2.0 -> InitN.B.
```

Contesta cada apartado en hojas separadas

Apartado 4. 2 puntos

Se desea diseñar una aplicación para la edición de documentos. Los documentos pueden ser de dos tipos: de texto plano o gráficos. Para cualquier tipo de documento se ha de guardar el nombre de fichero, el idioma en que está escrito, así como el autor. Un documento gráfico está compuesto de páginas, que tienen una cabecera y un pie (que pueden ser distintos para cada página). Las páginas a su vez pueden contener imágenes, sonidos, fragmentos de texto, o bien tener incrustados documentos (de cualquiera de los dos tipos). Una imagen está descrita por el nombre de fichero, su tipo (JPEG, GIF, PNG o BMP), tamaño y posición. Un sonido está descrito por el nombre de fichero y su tipo (WAV, MP3 o WMA), y un fragmento de texto por un nombre de fichero y el texto que contiene. Un documento de texto plano no contiene páginas, sino que está compuesto directamente por fragmentos de texto.

Se pide:

a) Construye el diagrama de clases UML que refleje el diseño.

b) Añade los métodos necesarios a las clases correspondientes para implementar los siguientes requisitos:
   a. Cuando una página se abre en el editor, ha de mostrar todos los componentes que contiene (imágenes, sonidos, fragmentos de texto y documentos).
   b. Por defecto, los documentos dentro de una página se muestran como un icono, reflejando simplemente el nombre del fichero. Los documentos han de poderse desplegar, y se mostraría entonces su contenido.
   c. Debe ser posible justificar los fragmentos de texto individuales, así como todos los fragmentos de la misma página.

Una posible solución: