# Examen Final de Junio

# Análisis y Diseño de Software (2011/2012)

**Contesta cada apartado en hojas separadas**

**Apartado 1. (3,5 puntos)**

Completa el siguiente programa de gestión de contratos telefónicos, para que su salida sea la que se indica más abajo. Hay dos tipos de contratos de **Teléfono: Fijo o Móvil**, ambos se crean con un número de teléfono, el DNI del titular, y una tarifa asociada. En algunos casos se puede crear un contrato de móvil asociándole un contrato de fijo previamente creado (con reducción el coste de llamadas). Cada vez que se realice una llamada sobre un contrato, se debe calcular el coste de la llamada, conforme a los siguientes criterios, para mantener siempre actualizado el gasto actual en cada contrato.

Las **llamadas desde móvil** con destino a un fijo asociado con dicho móvil (ver ejemplo en código main) será siempre gratuitas. Las llamadas desde móvil con tarifa *Plana* o *Especial* tienen un coste fijo por llamada independiente de la duración (0,0625€), PERO no se cobra nada por las llamadas a móviles contratados por el mismo DNI que el origen de la llamada. Con la tarifa *Básica*, el coste de la llamada desde móvil sólo depende de la duración de la misma (0,015625€ por segundo).

Las **llamadas desde fijo** con tarifa *Plana* tienen un coste fijo por llamada independiente de la duración (0,25€ si el destino es un móvil y 0,0€ si el destino es otro fijo). Con la tarifa Básica, el coste de la llamada desde fijo sólo depende de la duración de la misma (0,03125€ por segundo). La tarifa *Especial* desde fijos es similar a la tarifa *Básica*, excepto que una vez alcanzado un gasto de 10€ se aplica un 50% de descuento en todas las llamadas posteriores.

Téngase presente que, además de calcular los costes de las llamadas, la aplicación almacena los datos básicos de los contratos de forma que, mediante el **método directorio**, se puedan obtener fácilmente todos los contratos asociados con un DNI dado, incluido su gasto actual y ordenados por número de teléfono (tal y como se hace en la última línea del programa dado).

```
public class Apartado1 {
  public static void main(String[] args) {
       Fijo f1 = new Fijo("914972222", "53005100G", Tarifa.BASICA);
       Telefono f2 = new Fijo("986555555", "00123456F", Tarifa.PLANA);
       Telefono m3 = new Movil("608123456", "00123456F", Tarifa.ESPECIAL);
       Telefono m4 = new Movil("699610101", "00123456F", f1, Tarifa.BASICA);
       Telefono f5 = new Fijo("930001122", "11990022A", Tarifa.ESPECIAL);

       m3.llamada(m4, 65); // m3 llama a m4 65 segundos, sin COSTE, ESPECIAL mismo DNI
       m4.llamada(m3, 120); // m4 llama a m3 120 segundos, COSTE 1.875 = 120 * 0,015625€
       m4.llamada(f1, 15); // m4 llama a f1 15 segundos, sin COSTE, fijo asociado
       m3.llamada(f1, 100); // m3 llama a f1 100 segundos, COSTE fijo 0,0625 ESPECIAL
       for (int i=1; i<= 50; i++) {f1.llamada(f2, 200);} //coste 50 * 200 * 0.03125 = 312,5
       f2.llamada(f1, 5000); // f2 llama a f1 5000 segundos, sin COSTE, PLANA gratis a fijos
       for (int i=1; i<= 11; i++) { f5.llamada(m3, 32); } // 10 llamadas a 32 * 0.03125 = 10,
                                                          // y siguiente a 50% * 32 * 0.03125 = 0,5
       System.out.println("f1: " + f1);
       System.out.println("f5: " + f5);
       Telefono.directorio("00123456F");
  }
}
```

Salida:

```
f1: 914972222 con gasto actual -> 312.5
f5: 930001122 con gasto actual -> 10.5

Telefonos contratados con DNI: 00123456F
608123456 con gasto actual -> 0.0625
699610101 (asociado con fijo 914972222) con gasto actual -> 1.875
986555555 con gasto actual -> 0.0
Fin de telefonos con DNI: 00123456F
```

**Apartado 2. (3 puntos)**

Se quiere implementar una caché de objetos simple, de tal manera que se puedan almacenar objetos de distinta clase (siempre que esa clase implemente la interface Entity). Para acceder de manera rápida a los objetos, cada objeto tendrá un identificador único, para cada tipo. La interfaz Entity es la siguiente:

```
public interface Entity {
	//devuelve el identificador de la entidad, será único para entidades del mismo tipo
	Long getOID();
	//devuelve el nombre del tipo de la entidad, como un String
	String getTypeName();
}
```

La clase que implementa la cache, debe implementar la funcionalidad dada por la interfaz DBService. En particular, esta interfaz tiene un método para obtener un objeto dado su tipo e identificador (getEntity), para almacenar un objeto (putEntity), para borrar un objeto dado su tipo e identificador (removeEntity) y para obtener la lista de objetos de cierto tipo, cuyo identificador sea mayor que uno dado (getBigger). Como puedes ver, algunos métodos de acceso a la caché pueden lanzar la excepción `DBException`, que también debes diseñar tú.

```
public interface DBService<T extends Entity> {
	T getEntity(String typeName, long oid) throws DBException;
	void putEntity(T e);
	void removeEntity(String typeName, long oid) throws DBException;
	List<T> getBigger(String resultTypeName, long oid) throws DBException;
}
```

Se pide: Escribe el código de la clase que implemente la caché de objetos (llamada `DBCache`), la excepción `DBException` y completa el siguiente programa, para obtener la salida de más abajo. Es requisito indispensable que las operaciones de acceso a objetos tengan la máxima eficiencia posible.

```
class A implements Entity {
	private Long id;
	…… maxId = 0L;           //completar (1)
	public A() {
		this.id = A.maxId;
		A.maxId++;
	}
	@Override public Long getOID()         { return this.id;}
	@Override public String getTypeName() { return "A";   }
	@Override public String toString()     { return "A("+this.id+")"; }
}

public class Test {
	public static void main(String[] args) {
		A[] objs = { new A(), new A(), new A()};
		DBCache<A> dbc = new DBCache<A>();
		……{                                  // completar (2)
			for (A a: objs)  dbc.putEntity(a);

			System.out.println(dbc.getBigger("A", 0));
		}
		……{                                  // completar (3)
		   e.printStackTrace();
		}
	}
}
```

Salida:

[A(1), A(2)]

**Apartado 3 (1.5 puntos)**
Identifica las líneas del método *main* que causen error de compilación (explicando brevemente por qué), e indica la salida que produciría el siguiente programa después de eliminar dichas líneas:

```
//  en ARCHIVO Clase1.java
package paqueteA;
public class Clase1 {
      protected int x = 5;
      public String y = "vacio";

      void dobla() { x = x * 2; }

      public String getY() { return y; }

      public String cadena() { return "paqueteA.Clase1.y = "  + this.getY(); }
}
```

```
//  en ARCHIVO Subclase0.java
package paqueteB;
public class Subclase0 extends paqueteA.Clase1 {

      public void incrementa() { x = x + 1; }
}
```

```
//  en ARCHIVO Subclase1.java
package paqueteA;
import paqueteB.*;

public class Subclase1 extends Clase1 {
      int y = 17;

      public String getY() { return "[[y vale : " + y + "]]"; }

      public static void main(String[] args) {
            Clase1 a1 = new Subclase1();        // linea (1)
            Subclase1 s1 = new Subclase1();     // linea (2)
            Subclase0 s2 = new Subclase0();     // linea (3)
            System.out.println(a1.cadena());    // linea (4)
            System.out.println(s1.y);           // linea (5)
            System.out.println(a1.y);           // linea (6)
            s1.dobla();                         // linea (7)
            s1.incrementa();                    // linea (8)
            System.out.println(s1.x);           // linea (9)
            s2.dobla();                         // linea (10)
            s2.incrementa();                    // linea (11)
            System.out.println(s2.x);           // linea (12)
      }
}
```

## Apartado 4. (2 puntos)

La empresa "ESNAC" se dedica a la venta de entradas para eventos, y te ha encargado una aplicación que ayude a sus empleados en esta labor. Los eventos pueden ser de tres tipos: conciertos, obras de teatro y películas de cine. Todos los eventos tienen una descripción, una fecha, hora, y duración.

Los eventos tienen lugar en un recinto, que tiene un nombre y una dirección. Hay dos tipos de recintos, los que tienen localidades numeradas o no. Los que tienen localidades numeradas, tienen distintas zonas, cada una con un nombre (por ejemplo "palco 3", "platea", "entresuelo", etc), un número de filas y un número de butacas por fila. Los locales que no tienen localidades numeradas tienen un aforo máximo.

En el caso de recintos con localidades numeradas, cada evento establecerá para cada zona el precio de la entrada. Si el evento ocurre en un recinto sin localidades numeradas, entonces es el mismo precio para todas las entradas. En cualquiera de los dos casos, si es un concierto, el precio de una entrada sufre un incremento del 10%. Las proyecciones de cine pueden ser en 3D, y en ese caso tienen un incremento de 1.5 euros.

Así pues, la aplicación debe soportar la venta de entradas numeradas o no numeradas (que se venderán de una en una). En el primer caso, la entrada debe contener la zona, fila y butaca. Obviamente, la aplicación no debe permitir vender la misma localidad dos veces, para el mismo evento. A las entradas no numeradas se les asigna un identificador, que es el número de entradas para ese evento vendidas actualmente. En este caso, no es posible vender más entradas que el aforo del recinto.

Se pide:

a) Realiza un diagrama de clases de la aplicación.
b) Incluye métodos (no es necesario que pongas código) para que la aplicación tenga la siguiente funcionalidad:
    a. Calcular el precio de una entrada.
    b. Consultar si una determinada butaca está ocupada o no para un determinado evento.
    c. Indicar si hay entradas disponibles para un determinado evento.