Nombre:..... ***Legajo:.....***

Segundo Parcial de Programación Orientada a Objetos

23 de Noviembre de 2009

Ej. 1	*Ej. 2*	*Ej. 3*	*Nota*

- **Condición de aprobación: Tener dos ejercicios calificados como B o B-.**
- **Los ejercicios que no se ajusten estrictamente al enunciado, no serán aceptados.**
- **No es necesario agregar las sentencias import**
- **Además de las clases pedidas se pueden agregar las que se consideren necesarias.**

Ejercicio 1

Se cuenta con la interface **`Function`** que representa una función de N parámetros de entrada y uno de salida (resultado), además se ofrece **`MyFunction`** como ejemplo de implementación de dicha interface:

Function.java

```
/**
 * Modela una función de N parámetros de entrada y uno de salida
 *
 */
public interface Function {

	/**
	 * Evalúa una función de N parámetros representados a través
	 * de un arreglo de Object
	 *
	 * @param params Parámetros a usar
	 * @return resultado de evaluar la función
	 */
	public Object evaluate(Object[] params);
}
```

MyFunction.java

```
/**
 * Modela una función que retorna la longitud promedio de un conjunto de strings
 */
public class MyFunction implements Function {

	@Override
	public Object evaluate(Object[] params) {
		System.out.println("Evaluando MyFunction");
		Double total = 0.0;
		for (int i =0; i< params.length; i++) {
			String s = (String)params[i];
			total += s.length();
		}
		return (Double)(total/params.length);
	}
}
```

Se pide realizar otra implementación de **`Function`**, denominada **`CacheableFunction`** que recibe otra implementación de función y es capaz de evitar que se repitan cálculos innecesarios a través del almacenamiento de los resultados ya calculados (Se considera que la invocación a la función con los mismos parámetros, en el mismo orden, retorna el mismo resultado). El siguiente código muestra una prueba de la clase **`CacheableFunction`** y su salida.

TestFunction.java

```
public class TestFunction {

	public static void main(String[] args) {
		Function f = new CacheableFunction(new MyFunction());

		System.out.println("Prueba 1:");
		System.out.println(f.evaluate(new Object[] {"Hola", "mundo"}));

		System.out.println("Prueba 2:");
		System.out.println(f.evaluate(new Object[] {"lalalala"}));

		System.out.println("Prueba 3:");
		System.out.println(f.evaluate(new Object[] {"Hola", "mundo"}));

		System.out.println("Prueba 4:");
		System.out.println(f.evaluate(new Object[] {"mundo", "Hola"}));
	}
}
```

Salida de la ejecución de TestFunction.java

```
Prueba 1:
Evaluando MyFunction
4.5
Prueba 2:
Evaluando MyFunction
8.0
Prueba 3:
4.5
Prueba 4:
Evaluando MyFunction
4.5
```

Ejercicio 2

Se cuenta con la clase **`CellPhoneBill`** que permite registrar las llamadas realizadas por un teléfono celular. Dicha clase cuenta con el método **`processBill`** que calcula el monto de la factura en base a las llamadas registradas. Cada llamada está modelada mediante la clase **`Call`**, que calcula el precio de una llamada conociendo el costo de la misma por segundo. El código fuente es el siguiente:

CellPhoneBill.java

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

public class CellPhoneBill {

	private String number;
	private List<Call> calls;

	public CellPhoneBill(String number) {
		this.number = number;
		calls = new ArrayList<Call>();
	}

	public void registerCall(String toNumber, int duration) {
		addCall(new Call(this.number, toNumber, duration));
	}

	protected void addCall(Call c) {
		calls.add(c);
	}

	public double processBill() {
		double total = 0;
		for (Call c: calls) {
			total += c.getCost();
		}
		return total;
	}

	public String getNumber() {
		return number;
	}

	public void showCalls() {
		for (Call c: calls) {
			System.out.println(c);
		}
	}
}
```

Call.java

```
public class Call {

	private String from;
	private String to;
	private int duration;

	private static final double COST_PER_SECOND = 0.01;

	public Call(String from, String to, int duration) {
		this.from = from;
		this.to = to;
		this.duration = duration;
	}

	public double getCost() {
		return duration * COST_PER_SECOND;
	}

	@Override
	public String toString() {
		return "Call from:" + from + " to: " + to + "(" + duration + " seconds)";
	}
}
```

Se quiere agregar la promoción de “números amigos”, mediante la cual el usuario registra cierta cantidad de amigos, con los cuales sus llamadas tendrán un precio especial que corresponde a un porcentaje del valor real. La promoción contempla que se agreguen y eliminen números amigos en cualquier momento (en cada instante no puede haber más de 3), y también que se modifique el porcentaje a cobrar. Los valores a cobrar se consideran en base al estado en el momento de realizarse la llamada y NO en el momento de calcular el monto total. Lo que sigue es un programa de prueba basado en la clase **`FriendCellPhoneBill`** que responde a lo descripto anteriormente. Se pide implementar lo necesario para que el programa siguiente genere la salida que se indica en los comentarios.

TestCellPhoneBill.java

```
public class TestCellPhoneBill {

	public static void main(String[] args) {
		FriendCellPhoneBill bill = new FriendCellPhoneBill("1234");

		bill.setPercentage(0.5);

		bill.registerCall("1111", 200);
		bill.registerCall("2222", 100);

		//Muestra "Costo: 3.0" ya que son 300 segundos por el costo (0.01)
		System.out.println("Costo: " + bill.processBill());

		bill.addFriend("1111");

		//Muestra "Costo: 3.0" ya que no se agregaron nuevas llamadas
		System.out.println("Costo: " + bill.processBill());

		bill.setPercentage(0.25);
		bill.registerCall("1111", 400);

		//Muestra "Costo: 4.0" porque se agregó una llamada de 400 segundos
		//pero solo se cobra el 25% de la misma por ser número amigo
		System.out.println("Costo: " + bill.processBill());

		bill.removeFriend("1111");
		bill.registerCall("1111", 400);

		//Muestra "Costo: 8.0" porque a lo anterior se agrega una nueva llamada
		//de 400 segundos y el numero dejó de ser amigo
		System.out.println("Costo: " + bill.processBill());

		bill.showCalls(); //Muestra lo siguiente
				//Call from:1234 to: 1111(200 seconds)
				//Call from:1234 to: 2222(100 seconds)
				//Call from:1234 to: 1111(400 seconds)0.25%
				//Call from:1234 to: 1111(400 seconds)

		bill.addFriend("1111");
		bill.addFriend("2222");
		bill.addFriend("3333");

		//Muestra "Costo: 9.0" porque a lo anterior se agrega una nueva llamada
		//de 400 segundos y el numero es amigo
		bill.registerCall("3333", 400);
		System.out.println("Costo: " + bill.processBill());

		bill.addFriend("4444");	// Se produce TooManyFriendsException
	}
}
```

Ejercicio 3

Escribir la interface `ConvertList<T>` que extiende de `List<T>` agregando un método que dada cierta función que construye un objeto de tipo S en base a un objeto de tipo T (implementada a través de una nueva interface `Function<T,S>`) retorne una nueva lista con los objetos obtenidos al aplicar la función a los objetos de la lista original. Implementar la clase `ConvertArrayList<T>` que herede de `ArrayList<T>` e implemente la nueva interface `ConvertList<T>`. Implementar además un pequeño programa de prueba `TestConvertList` que almacene en una `ConvertList` una lista de objetos `String` y luego le aplique una función para obtener otra `ConvertList` con las longitudes de cada uno.

Se pide: Implementar `ConvertList.java`, `Function.java`, `ConvertArrayList.java` y `TestConvertList.java`