Ejercicios Resueltos

Ejercicio 01.

El siguiente código muestra para calcular la potencia de dos de cualquier número con cualquier cantidad de cifras. Para evitar desbordamientos de tipos básicos, se utiliza una lista para cada cifra y se gestiona a mano los acarreos.

```
public static String povOf2(int exp) {
    if (exp == 0)
        return "0";
    List<Integer> digits = new ArrayList<Integer>();
    digits.add(1);
    int size = 1;
    int tmp;
    int ac = 0;
    //Primer bucle
    for (int i = 0; i < exp; i++) {
        // Segundo bucle
        for (int c = 0; c < size; c++) {
            tmp = digits.get(c);
            tmp *= 2;
            tmp += ac;
            ac = 0;
            if (tmp > 9) {
                tmp -= 10;
                ac = 1;
            }
            digits.set(c, tmp);
        }
        // Segundo if
        if (ac == 1) {
            digits.add(ac);
            size++;
            ac = 0; // (1)
        }
    }
    String s = "";
    // Tercer bucle
    for (int i = digits.size()-1; i >= 0; i--) {
        s += digits.get(i);
    }
    return s;
}
```

A la vista del código anterior. ¿Cuáles serían las pruebas que haría? El objetivo es buscar el número mínimo de pruebas que sea relevante, es decir, que prueben condiciones distintas..

Esta solución está publicada en Solveet en el siguiente enlace: http://www.solveet.com/exercises/Potencias-de-Dos/30/solution-1315

Solución:

Las pruebas están resumidas en la siguiente tabla:

Prueba	Explicación
povOf2(0)	Probamos que el primer if se ejecuta correctamente y devuelve 1
povOf2(1)	Probamos que el primer bucle y el tercer bucle se ejecutan una vez de manera exitosa
povOf2(4)	Probamos que el primer y segundo if se ejecutan
povOf2(5)	Como ambos ifs deja los valores establecidos para la próxima iteración de los bucles, comprobamos que funcionan.
povOf2(valor grande)	Comprobamos que se pueden calcular valores grandes

Cada prueba verifica un comportamiento (o ejecución) distinta del código. Para poder determinar las pruebas a hacer a un código es interesante conocer algunas de las técnicas clásicas que existen desde los años 70 como análisis de caminos, particiones equivalentes, valores límites, etc.

En este caso sería recomendable extraer el código que genera la cadena de texto con el número a un método aparte, ya que convertir el resultado en una cadena es una responsabilidad distinta de calcular la potencia de dos de un número. Otra posible refactorización es mover la gestión del acarreo a una pequeña clase auxiliar para que se vea más claro el por qué se hace cada operación en el algoritmo.

Ejercicio 02.

Escriba un conjunto de pruebas para verificar el comportamiento de un método que recibe como parámetro dos conjuntos ordenados (o dos listas o dos arrays ordenadas de menor a mayor) y devuelva un nuevo conjunto ordenado que contenga los elementos de ambos conjuntos en el mismo orden.

¿Cómo podríamos saber el número de elementos comunes que había en ambos conjuntos de entrada?

Solución

Los valores de entrada son las dos listas ordenadas y el resultado obtenido es una tercera lista. Veamos qué combinaciones podemos hacer.

Lista A	Lista B	Resultado esperado
Vacía	Vacía	Lista vacía
Vacía	No vacía	Lista B
No Vacía	Vacía	Lista A
N elementos	N elementos	Lista con 2N elementos ordenada
N elementos	M elementos	Lista con N+M elementos ordenada
M elementos	N elementos	Lista con N+M elementos ordenada

M es siempre mayor que N.

Según los resultados esperados podríamos ahorrarnos algunas pruebas ya que las últimas combinaciones generan el mismo resultado. Pero si implementamos el código vemos que en los dos últimos casos, aunque el resultado es el mismo, se ejecuta un código distinto, por lo que para cumplir con el criterio de caminos posibles añadiríamos también las dos últimas pruebas.

Ejercicio 03.

Contamos con una clase C con un atributo a (por ejemplo de tipo *String*) y un método *getA* que nos devuelve su valor. También disponemos de una factoría F con un método *creaLista* que no admite ningún parámetro y crea y devuelve una lista (interfaz List) de objetos de clase C en el que cada objeto tiene un valor en su atributo a distinto de los demás.

Escriba utilizando una herramienta XUnit un caso de prueba que verifique que todos los objetos en la lista devuelta por el método *creaLista* son distintos. Intente no presuponer el orden de los elementos ni valores concretos para el atributo *a*.

Solución

El código que el enunciado nos dice que tenemos puede ser algo así:

```
public class C {
	String a;
```

```
	public C(String s)
		{a = s;
	}

	public String getA() {
		return a;
	}
}

public class F {

	public static List<C> creaLista() {
		return Arrays.asList(new C("A"), new C("B"), new C("C"), new C("D"));
	}

}
```

En este caso la idea que aplico es guardar todos los elementos de la lista en un conjunto y comprobar si ambos tienen el mismo tamaño. Si no lo tienen es que había elementos repetidos. Para esto necesito que los objetos de la clase C se puedan comparar, por lo que creo un comparador basado en el atributo A.

```
	class ComparatorForC implements Comparator<C>
		{@Override
		public int compare(C arg0, C arg1) {
			return arg0.getA().compareTo(arg1.getA());
		}
	}

	@Test
	public void testFactory_TodosLosElementosDeLaListaSonDistintos()
		{List<C> listOfC = F.creaLista();
		Set<C> setOfC = new TreeSet<C>(new ComparatorForC());
		setOfC.addAll(listOfC);

		assertEquals(listOfC.size(), setOfC.size());
	}
```

Y la prueba funciona.

Sin embargo no me fio de esta prueba. Estoy dando por hecho que la prueba es correcta pero sólo la he hecho funcionar con una lista cuyos elementos son todos distitis. Si la factoría fallara y repitiera elementos, ¿mi prueba lo detectaría? No lo sé.

Para resolver esta duda voy a crear a mano una lista con elementos repetidos y voy a repetir la prueba. Si es capaz de detectar que la lista no es correcta y falla me sentiré más seguro y confiaré más. Veamos el código.

```
	@Test
	public void testDetectaListasErroneas() {
```

```
        List<C> listOfC = Arrays.asList(new C("A"), new C("B"), new C("C"), new C("A"));

        Set<C> setOfC = new TreeSet<C>(new ComparatorForC());
        setOfC.addAll(listOfC);

        assertEquals(listOfC.size()-1, setOfC.size());

    }
```

Efectivamente, el conjunto tiene un elemento menos que la lista, por lo que mi prueba es correcta.

Además podemos aprovechar el código anterior para practicar refactorización. Aunque la prueba no tiene muchas líneas puede no ser fácil de entender, por ejemplo en el assert puede que no esté claro el por qué se resta un 1. Además hay mucho código repetido.

El primer paso es crear un método factoría que genere el conjunto de elementos a partir de la lista.

```
private Set<C> createSetFromListUsingComparatorForC(List<C> listOfC)
        {Set<C> setOfC = new TreeSet<C>(new ComparatorForC());
        setOfC.addAll(listOfC);
        return setOfC;
}
```

El segundo paso es mover el assert de la segunda prueba a un método que indique con más claridad qué estamos probando:

```
void assertOneElementIsRepeated(List<C> listOfC, Set<C> setOfC) {
        assertEquals(listOfC.size()-1, setOfC.size());
}
```

El código de las pruebas quedaría de esta manera.

```
@Test
public void testDetectaListasErroneas() {
        List<C> listOfC = Arrays.asList(new C("A"), new C("B"), new C("C"), new C("A"));

        Set<C> setOfC = createSetFromListUsingComparatorForC(listOfC);

        assertOneElementIsRepeated(listOfC, setOfC);
}
```