Ejercicios resueltos de programación

Mariano Fernández López
Escuela Politécnica Superior, Universidad San Pablo CEU
18 de marzo de 2015

Índice general

Capítulo 1

Implementación de un método recursivo

1.1. Enunciado

Siguiendo la técnica de desarrollo dirigido por pruebas, implemente un método recursivo que permita calcular la suma de los n primeros números naturales.

1.2. Resolución

En primer lugar se escribe la clase donde va a estar ubicado el método a implementar:

```
package numeros;

public class Numero {

}
```

A continuación, se escribe un test para forzar la creación del método:

```
package numeros;

import org.junit.Test;
import static org.junit.Assert.*;

public class NumeroTest {

    @Test
    public void testSumaHasta0() {
        assertEquals(0, Numero.sumaNPrimeros(0));
    }

}
```

El código incial del método tiene lo mínimo para permitir la compilación del test, y para verlo fallar. Recuérdese que, para garantizar que el test es útil, es importante verlo fallar al menos una vez.

```
static Object sumaNPrimeros(int i) {
    throw new UnsupportedOperationException("Not supported yet."); //To change body c
}
```

A continuación, se escribe el código mínimo en el método para que pase el test. De hecho, se devuelve una constante:

```
public static int sumaNPrimeros(int i) {
    return 0;
}
```

Se lleva ahora a cabo la triangulación mediante un test que obliga a generalizar el código del método.

```
@Test
public void testSumaHasta3() {
    assertEquals(6, Numero.sumaNPrimeros(3));
}
```

Este test lo vemos fallar con el código actual del método.

Para generalizar el código del método se utiliza la técnica vista en clase para los métodos recursivos. En primer lugar, se escribe el caso básico y, para el caso complejo, se asume que hay un duende que resuelve el problema con la entrada reducida.

```
public static int sumaNPrimeros(int n) {
    if (n == 0) return 0;
    else return n + sumaNPrimeros(n - 1);
}
```

Luego, se añade el siguiente test para tratar los casos anómalos:

```
@Test(expected = ArithmeticException.class)
public void testSumaHastaMenos3() throws ArithmeticException {
    Numero.sumaNPrimeros(-3);
}
```

La versión actual del método permite ver fallar el test.

A continuación, hay que modificar el método para que pase el test:

```
public static int sumaNPrimeros(int n) throws ArithmeticException {
    if (n < 0) throw new ArithmeticException("Este método no permite "
            + "números menores que 0");
    if (n == 0) return 0;
    else return n + sumaNPrimeros(n - 1);
}
```

Por último, se introduce otro test para dotar de más robustez al código realizando una comprobación sobre un número de varios órdenes de magnitud mayor que los de las pruebas anteriores:

```
@Test
public void testSumaHasta10000() throws ArithmeticException {
    assertEquals(50005000, Numero.sumaNPrimeros(10000));
}
```

Para asegurarse de que este test es útil, hay que hacerlo fallar modificando el código del método a implementar, aunque luego, obviamente, hay que restaurarlo.

Capítulo 2

Implementación de una cola

2.1. Enunciado

De acuerdo con la técnica de desarrollo dirigido por pruebas, implemente una estructura de cola con las siguientes operaciones:

- Está vacía: devuelve *verdadero* si la cola está vacía, y *falso* en otro caso.
- Insertar: introduce un elemento al principio de la cola.
- Borrar: elimina un elemento del final de la cola.
- Primero: obtiene el primer elemento de la cola.
- Tamaño: devuelve el número de elementos que tiene la cola.

Para entender esta estructura de datos podemos pensar en la cola de un determinado servicio. Van llegando individuos a la cola (insertar en la cola), y se van atendiendo individuos (eliminar de la cola). La política de esta estructura de datos es FIFO: el primero que entra es el primero que sale, es decir, el primero que llega a la cola es el primero en ser atendido.

2.2. Implementación de la cola con un *array*

2.2.1. Primera versión de la clase *Cola* y de los métodos *estaVacia* e *insertar*

En un proyecto que ya tengamos de estructuras de datos o en un proyecto nuevo, empezamos creando la clase *Cola* en el paquete *cola*:

```
package colas;

public class Cola {
}
```

A continuación, vamos creando los tests que necesitamos para ir dotando de contenido a la clase *Cola*. El primer test, que permite comprobar si una cola recién creada está vacía, nos fuerza a crear el método *estaVacia*. La clase *ColaTest*, con el método *testEstaVaciaColaRecienCreada*, se muestra a continuación:

```
package colas;

import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author mariano
 */
public class ColaTest {

    Cola cola;

    public ColaTest() {
    }

    @Before
    public void setUp() {
        cola = new Cola();
    }

    @Test
    public void testEstaVaciaColaRecienCreada() {
        assertEquals(true, cola.estaVacia());
    }

}
```

Tal y como se puede comprobar, se crea siempre una cola antes de cada test (véase el código de *setUp*).

El método *estaVacia*, de la clase *Cola*, que generamos automáticamente con Netbeans es el siguiente:

```
    Object estaVacia() {
throw new UnsupportedOperationException("Not supported yet.");
}
```

Como se puede comprobar, el método *estaVacia* todavía no tiene contenido, y su tipo de retorno no es el adecuado. El único propósito del código anterior es que el test compile y lo veamos fallar. Ahora modificamos *estaVacia* para hacer funcionar el test:

```
    public boolean estaVacia() {
        return true;
    }
```

A continuación, se procede a la triangulación para generalizar el método *estaVacia*, con el siguiente test:

```
@Test
public void testNoEstaVaciaColaConUnUnElemento()
{
    cola.insertar("Juan");
    assertEquals(false, cola.estaVacia());
}
```

Con el código que teníamos antes del método *estaVacia*, vemos fallar el test.

Para hacerlo funcionar, tenemos que pensar en un código general para la clase *Cola*:

```
public class Cola {
    int indiceUltimo, indicePrimero = -1;
    int tamMax = 100;
    Object[] array = new Object[tamMax];

    public boolean estaVacia() {
        return indicePrimero==-1;
    }

    public void insertar(Object elemento) {
        array[indiceUltimo++] = elemento;
        indicePrimero++;
    }
}
```

2.2.2. Primera versión del método *borrar*

Seguimos escribiendo tests para implementar el método *borrar*. El primero de ellos comprueba si, después y de insertar y eliminar en una cola, la cola se queda vacía.

```
@Test
public void testInsertarYBorrarDejaLaColaVacia()
{
    cola.insertar("Juan");
    cola.borrar();
    assertEquals(true, cola.estaVacia());
}
```

El código de *borrar* que generamos automáticamente con Netbeans es el que se muestra a continuación:

```
void borrar() {
    throw new UnsupportedOperationException("Not supported yet.");
}
```

Con este código conseguimos compilar y ver fallar el test. Ahora, creamos un código que permite hacer funcionar el test:

```
public void borrar() {
    indicePrimero--;
}
```

2.2.3. Primera versión del método *primero*

A continuación, escribimos un test para provocar la creación del método *primero*:

```
@Test
public void testPrimeroDeUnaColaConUnElemento()
{
    cola.insertar("Juan");
    assertEquals(0, ((String) cola.primero()).compareTo("Juan"));
}
```

Con el siguiente código conseguimos que compile el test y verlo fallar:

```
Object primero() {
    throw new UnsupportedOperationException("Not supported yet.");
}
```

Ahora, corregimos el código de *primero* para hacerlo funcionar.

```
public Object primero() {
    return array[indicePrimero];
}
```

2.2.4. Refinamiento de *indicePrimero* e *indiceUltimo*

En este momento surge un problema: ¿cómo puede ser que en el método de insertar siempre incrementemos el *indicePrimero* si lo que debe ocurrir es que, salvo para el primer individuo que entra en la cola, sólo se incremente el *indiceUltimo*?

Para forzar la corrección de este código, combinamos dos inserciones y la recuperación del primero de la cola mediante el siguiente test:

```
@Test
public void testPrimeroDeUnaColaConDosElementos()
{
    cola.insertar("Juan");
    cola.insertar("Miguel");
    assertEquals(0, ((String) cola.primero()).compareTo("Juan"));
}
```

Para que el test funcione, tenemos que modificar el código de insertar para que quede de la siguiente manera:

```
public void insertar(Object elemento) {
    array[indiceUltimo++] = elemento;
    if (estaVacia()) indicePrimero=0;
}
```

Obsérvese que la modificación del índice del primero sólo se produce cuando se inserta en una cola vacía.

No obstante, el tratamiento del *indicePrimero* sigue sin estar afinado. No puede ser que sólo cambie cuando se inserta la primera vez. Cuando se realicen borrados, también hay que modificar este índice. De hecho, se puede comprobar que el test que se muestra a continuación:

```
@Test
public void testPrimeroDeUnaColaDespuesDeAnnadir2EltosYBorrarUno()
{
    cola.insertar("Juan");
    cola.insertar("Miguel");
    cola.borrar();
    assertEquals(0, ((String) cola.primero()).compareTo("Miguel"));
}
```

donde se insertan dos elementos y se borra uno, falla para la versión actual del código de la clase *Cola*.

Para que el test funcione, es necesario que el borrado modifique *indicePrimero*, tal y como se muestra en el siguiente código:

```
public void borrar() {
    indicePrimero++;
}
```

Sin embargo, falla el test de insertar un elemento y borrar otro, porque no hay constancia de que la cola quede vacía. Para conseguir que todos los tests funcionen, incluido este último, es necesario realizar cambios en los siguientes componentes de la clase *Cola*:

1. La inicialización de los índices primero y último.
2. La condición de cola vacía.
3. La inserción de nuevos elementos.

El código queda tal y como se muestra a continuación:

```
public class Cola {
    int indiceUltimo =-1, indicePrimero = 0;
    int tamMax = 100;
    Object[] array = new Object[tamMax];

    public boolean estaVacia() {
        return indicePrimero > indiceUltimo;
    }
```

```
    public void insertar(Object elemento) {
        array[++indiceUltimo] = elemento;
    }

    public void borrar() {
        indicePrimero++;
    }

    public Object primero() {
        return array[indicePrimero];
    }
}
```

2.2.5. Primera versión del método *tamanno*

A continuación, pasamos a la escritura del primer test para forzar la creación del método de obtención del tamaño de la cola:

```
    @Test
    public void testTamannoColaVacia()
    {
        assertEquals(0, cola.tamanno());
    }
```

El código que generamos automáticamente con Netbeans para *tamanno* es el siguiente:

```
    Object tamanno() {
        throw new UnsupportedOperationException("Not supported yet.");
    }
```

Para hacer funcionar el test, como de costumbre, en la primera versión del método a implementar se devuelve una constante:

```
    public int tamanno() {
        return 0;
    }
```

Se procede ahora a la triangulación escribiendo un test para comprobar el tamaño de una cola con un solo elemento:

```
    @Test
    public void testTamannoColaUnElemento()
    {
        cola.insertar("Juan");
        assertEquals(1, cola.tamanno());
    }
```

La versión actual del método *tamanno* nos permite ver fallar el test.

La forma más fácil de modificar el código para que pase el test es creando un atributo *tam* que se incremente cada vez que se inserta un nuevo elemento, y se decremente cada vez que se borre. El código de la clase *Cola* queda ahora tal y como se muestra a continuación:

```
public class Cola {
    int indiceUltimo =-1, indicePrimero = 0, tam = 0;
    int tamMax = 100;
    Object[] array = new Object[tamMax];

    public boolean estaVacia() {
        return indicePrimero > indiceUltimo;
    }

    public void insertar(Object elemento) {
        array[++indiceUltimo] = elemento;
        tam++;
    }

    public void borrar() {
        indicePrimero++;
        tam--;
    }

    public Object primero() {
        return array[indicePrimero];
    }

    public int tamanno() {
        return tam;
    }

}
```

2.3. Tratamiento de excepciones

Para llegar al tratamiento de excepciones, se codifican tests para los casos extremos. Vamos a empezar con el caso de intentar obtener el primer elemento de una cola vacía. Dado que pretendemos lanzar una excepción de cola vacía, debemos crear la clase de excepción correspondiente en una paquete llamado *excepciones*:

```
package excepciones;

public class ExcepcionDeColaVacia extends Exception {
    public ExcepcionDeColaVacia(String descripcion)
    {
        super(descripcion);
    }
```

```
}
```

El test que espera la excepción al intentar acceder al primer elemento de una cola vacía es el que se muestra a continuación:

```
@Test(expected = ExcepcionDeColaVacia.class)
public void testPrimeroColaVacia() throws ExcepcionDeColaVacia
{
    cola.primero();
}
```

Dado que esta excepción no se lanza todavía en el código de *primero*, el código actual de *Cola* nos permite ver fallar el test.

Para hacer funcionar el test, modificamos el código de *primero* para que quede de la siguiente manera:

```
public Object primero() throws ExcepcionDeColaVacia {
    if (estaVacia()) throw new ExcepcionDeColaVacia("Intenta obtener el primer "
            + "elemento de una cola vacía");
    return array[indicePrimero];
}
```

Esto obliga a tratar la excepción en aquellos tests donde hay una invocación a *primero*, por ejemplo, en el que se muestra a continuación:

```
@Test
public void testPrimeroDeUnaColaDespuesDeAnnadir2EltosYBorrarUno()
{
    try {
        cola.insertar("Juan");
        cola.insertar("Miguel");
        cola.borrar();
        assertEquals(0, ((String) cola.primero()).compareTo("Miguel"));
    } catch (ExcepcionDeColaVacia ex) {
        Logger.getLogger(ColaTest.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Si no nos fiamos de la transformación de estos tests, podemos verlos fallar modificando el código de *primero*, por ejemplo, tal y como se puede ver en el siguiente código:

```
    public Object primero() throws ExcepcionDeColaVacia {
//        if (estaVacia()) throw new ExcepcionDeColaVacia("Intenta obtener el primer "
//                + "elemento de una cola vacía");
//        return array[indicePrimero];
        return null;
    }
```

Si volvemos al código que teníamos antes de modificar *primero*, ya vemos que sí se pasan todos los tests:

```
public Object primero() throws ExcepcionDeColaVacia {
    if (estaVacia()) throw new ExcepcionDeColaVacia("Intenta obtener el primer "
            + "elemento de una cola vacía");
    return array[indicePrimero];
}
```

Con el borrado se procede de forma análoga al método *primero*. Empezamos creando el test que espera la excepción cuando se borra en una cola vacía:

```
@Test(expected = ExcepcionDeColaVacia.class)
public void testBorrarEnColaVacia() throws ExcepcionDeColaVacia
{
    cola.borrar();
}
```

que podemos ver fallar con la versión actual de la clase *Cola*.
Para que funcione el test, tenemos que modificar el método *borrar*:

```
public void borrar() throws ExcepcionDeColaVacia {
    if (estaVacia()) throw new ExcepcionDeColaVacia("Intenta borrar en una "
            + "cola vacía");
    indicePrimero++;
    tam--;
}
```

El test lo vemos fallar con la versión del código actual. Sin embargo, este otro código lo hace funcionar:

```
public void borrar() throws ExcepcionDeColaVacia {
    if (estaVacia()) throw new ExcepcionDeColaVacia("Intenta borrar en una "
            + "cola vacía");
    indicePrimero++;
    tam--;
}
```

De nuevo, hay que incluir el tratamiento de esta excepción en un test que hace referencia al borrado y que no hace referencia a la excepción de pila vacía. En los demás métodos con borrado sí se ha incluido la excepción, porque invocan al método *primero*, que obliga al tratamiento de esta excepción.

```
@Test
public void testInsertarYBorrarDejaLaColaVacia()
{
    try {
        cola.insertar("Juan");
```

```
            cola.borrar();
            assertEquals(true, cola.estaVacia());
        } catch (ExcepcionDeColaVacia ex) {
            Logger.getLogger(ColaTest.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
```

2.4. La cola con un *array* móvil

A continuación, realizamos los cambios oportunos en *Cola* para que, cada vez que se llene el *array*, realice una copia en un *array* más grande.

Esto también lo forzamos mediante desarrollo dirigido por pruebas. De hecho, el siguiente test, hace fallar el código actual al desbordar la cola:

```
    @Test
    public void testColaGrande()
    {
        try {
            for(int i = 1; i <= tamMax+4; i++)
            {
                cola.insertar(i);
            }

            assertEquals(1, cola.primero());
        } catch (ExcepcionDeColaVacia ex) {
            Logger.getLogger(ColaTest.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
```

Para evitar que falle, al intentar insertar en una cola que está llena, se crea un *array* el doble de grande que el actual y se copia el contenido actual de cola.

Creamos ahora un test para forzar el tener un constructor donde el tamaña máximo se introduzca como parámetro.

```
    @Test
    public void testConstructorParametro()
    {
        try {
            Cola cola2 = new Cola(10);
            cola2.insertar("Juan");
            assertEquals(0, ((String) cola2.primero()).compareTo("Juan"));
        } catch (ExcepcionDeColaVacia ex) {
            Logger.getLogger(ColaTest.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
```

Para que compile, tenemos que añadir el siguiente código en la clase *Cola*:

```
    public Cola(){}

    Cola(int i) {
        throw new UnsupportedOperationException("Not supported yet."); //To change body c
    }
```

Obsérves que, además del constructor con parámetro, ha habido que crear un constructor sin parámetros. Con este código, vemos fallar el test.

Para hacerlo funcionar, dotamos de contenido al constructor con parámetro.

```
    Cola(int tamMax) {
        this.tamMax = tamMax;
    }
```

2.5. La cola con un *array* circular

Ahora, el problema que tiene esta estructura es de eficiencia. En concreto, no se aprovechan los huecos que van quedando en el array conforme van saliendo elementos de la cola. Para aprovecharlos, podemos tratar el *array* como un *array* circular utilizando aritmética modular. Esto obliga a cambiar el código donde se modifiquen *indicePrimero* e *indiceUltimo*. Por ejemplo, donde antes se estaba escrito $indiceUltimo++$, ahora hay que escribir $indiceUltimo = (indiceUltimo + 1)\,\%tamMax$;. Tampoco vale la versión actual del método *estaVacia*.

El código de la clase *Cola* queda como se muestra a continuación:

```
package colas;

import excepciones.ExcepcionDeColaVacia;

public class Cola {

    int indiceUltimo = -1, indicePrimero = 0, tam = 0;
    int tamMax = 100;
    Object[] array = new Object[tamMax];

    public Cola(){}

    Cola(int tamMax) {
        this.tamMax = tamMax;
    }

    public boolean estaVacia() {
        return (indiceUltimo + 1) % tamMax == indicePrimero;
    }

    public void insertar(Object elemento) {
        if (tam == tamMax) {
```

```
            tamMax = tamMax * 2;
            Object[] arrayAux = new Object[tamMax];
            System.arraycopy(array, 0, arrayAux, 0, array.length);
            array = arrayAux;
        }

        indiceUltimo = (indiceUltimo + 1) % tamMax;
        array[indiceUltimo] = elemento;
        tam++;
    }

    public void borrar() throws ExcepcionDeColaVacia {
        if (estaVacia()) {
            throw new ExcepcionDeColaVacia("Intenta borrar en una "
                    + "cola vacía");
        }
        indicePrimero = (indicePrimero + 1) % tamMax;
        tam--;
    }

    public Object primero() throws ExcepcionDeColaVacia {
        if (estaVacia()) {
            throw new ExcepcionDeColaVacia("Intenta obtener el primer "
                    + "elemento de una cola vacía");
        }
        return array[indicePrimero];
    }

    public int tamanno() {
        return tam;
    }

}
```

2.6. Añadido de pruebas para hacer más robusto el software

Una vez que se tiene la software que se tenía que construir, hay que comprobar que pruebas quedan por realizar para tener ciertas garantías de que es suficientemente robusto.

Se puede comprobar que el método *tamanno* sólo aparece en dos tests. Por tanto, habría que incluirlo en algún otro test. Por ejemplo, se puede comprobar si el tamaño sigue comprobándose bien cuando se copia el *array* después de llenarse el primero.

```
    @Test
    public void testTamannoColaGrande() {
        for (int i = 1; i <= tamMax + 4; i++) {
            cola.insertar(i);
```

```
        }

        assertEquals(tamMax + 4, cola.tamanno());
    }
```

Para ver fallar el test, se puede manimpular el código de *tamanno* para que devuelva siempre 0. Una vez hemos visto fallar el test, recuperamos el código anterior de *tamanno*.

También sería conveniente comprobar que los elementos que creemos haber copiado de un *array* al otro siguen estando ahí. Podemos, una vez que hemos forzado la copia llenando el *array* inicial, volver a borrar elementos y comprobar que el primero de la cola es uno de los que estaba en el *array* que habíamos llenado inicialmente:

```
    @Test
    public void testCopiaCorrectaEnColaGrande() throws ExcepcionDeColaVacia {
        for (int i = 1; i <= tamMax + 4; i++) {
            cola.insertar(i);
        }

        System.out.println();

        for (int i = 1; i <= 10; i++) {
            cola.borrar();
        }

        assertEquals(11, cola.primero());
    }
```

También podemos comprobar que, al borrar hasta llegar a los elementos que han entrado nuevos en el array más grande, también se recupera el primero de la cola correctamente:

```
    @Test
    public void testCopiaCorrectaYMuchoBorradoEnColaGrande() throws ExcepcionDeColaVacia
        for (int i = 1; i <= tamMax + 4; i++) {
            cola.insertar(i);
        }

        System.out.println();

        for (int i = 1; i <= tamMax + 2; i++) {
            cola.borrar();
        }

        assertEquals(tamMax + 3, cola.primero());
    }
```

Por último, habría que ver fallar los dos tests anteriores manipulando el método *primero*. Luego, habría que recuperar su código inicial.

2.7. Código final con los tests y la cola

A modo de síntesis, se muestra el código escrito tanto para las pruebas, como para la estructura de datos en sí.

```
package colas; //en la zona de paquetes de pruebas

import excepciones.ExcepcionDeColaVacia;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class ColaTest {

    int tamMax = 100;

    Cola cola;

    public ColaTest() {
    }

    @Before
    public void setUp() {
        cola = new Cola();
    }

    @Test
    public void testEstaVaciaColaRecienCreada() {
        assertEquals(true, cola.estaVacia());
    }

    @Test
    public void testNoEstaVaciaColaConUnUnElemento() {
        cola.insertar("Juan");
        assertEquals(false, cola.estaVacia());
    }

    @Test
    public void testInsertarYBorrarDejaLaColaVacia() {
        try {
            cola.insertar("Juan");
            cola.borrar();
            assertEquals(true, cola.estaVacia());
        } catch (ExcepcionDeColaVacia ex) {
            Logger.getLogger(ColaTest.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @Test
    public void testPrimeroDeUnaColaConUnElemento() {
        try {
            cola.insertar("Juan");
```

```
            assertEquals(0, ((String) cola.primero()).compareTo("Juan"));
        } catch (ExcepcionDeColaVacia ex) {
            Logger.getLogger(ColaTest.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @Test
    public void testPrimeroDeUnaColaConDosElementos() {
        try {
            cola.insertar("Juan");
            cola.insertar("Miguel");
            assertEquals(0, ((String) cola.primero()).compareTo("Juan"));
        } catch (ExcepcionDeColaVacia ex) {
            Logger.getLogger(ColaTest.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @Test
    public void testPrimeroDeUnaColaDespuesDeAnnadir2EltosYBorrarUno() {
        try {
            cola.insertar("Juan");
            cola.insertar("Miguel");
            cola.borrar();
            assertEquals(0, ((String) cola.primero()).compareTo("Miguel"));
        } catch (ExcepcionDeColaVacia ex) {
            Logger.getLogger(ColaTest.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    @Test
    public void testTamannoColaVacia() {
        assertEquals(0, cola.tamanno());
    }

    @Test
    public void testTamannoColaUnElemento() {
        cola.insertar("Juan");
        assertEquals(1, cola.tamanno());
    }

    @Test(expected = ExcepcionDeColaVacia.class)
    public void testPrimeroColaVacia() throws ExcepcionDeColaVacia {
        cola.primero();
    }

    @Test(expected = ExcepcionDeColaVacia.class)
    public void testBorrarEnColaVacia() throws ExcepcionDeColaVacia {
        cola.borrar();
    }

    @Test
    public void testColaGrande() {
        try {
            for (int i = 1; i <= tamMax + 4; i++) {
```

```
            cola.insertar(i);
        }

        assertEquals(1, cola.primero());
    } catch (ExcepcionDeColaVacia ex) {
        Logger.getLogger(ColaTest.class.getName()).log(Level.SEVERE, null, ex);
    }
}

@Test
public void testTamannoColaGrande() {
    for (int i = 1; i <= tamMax + 4; i++) {
        cola.insertar(i);
    }

    assertEquals(tamMax + 4, cola.tamanno());
}

@Test
public void testCopiaCorrectaEnColaGrande() throws ExcepcionDeColaVacia {
    for (int i = 1; i <= tamMax + 4; i++) {
        cola.insertar(i);
    }

    System.out.println();

    for (int i = 1; i <= 10; i++) {
        cola.borrar();
    }

    assertEquals(11, cola.primero());
}

@Test
public void testCopiaCorrectaYMuchoBorradoEnColaGrande() throws ExcepcionDeColaVacia
    for (int i = 1; i <= tamMax + 4; i++) {
        cola.insertar(i);
    }

    System.out.println();

    for (int i = 1; i <= tamMax + 2; i++) {
        cola.borrar();
    }

    assertEquals(tamMax + 3, cola.primero());
}


@Test
public void testConstructorParametro() {
    try {
        Cola cola2 = new Cola(10);
        cola2.insertar("Juan");
```

```
            assertEquals(0, ((String) cola2.primero()).compareTo("Juan"));
        } catch (ExcepcionDeColaVacia ex) {
            Logger.getLogger(ColaTest.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

}
```

La estructura de *Cola* es la siguiente:

```
package colas;

import excepciones.ExcepcionDeColaVacia;

public class Cola {

    int indiceUltimo = -1, indicePrimero = 0, tam = 0;
    int tamMax = 100;
    Object[] array = new Object[tamMax];

    public Cola(){}

    Cola(int tamMax) {
        this.tamMax = tamMax;
    }

    public boolean estaVacia() {
        return (indiceUltimo + 1) % tamMax == indicePrimero;
    }

    public void insertar(Object elemento) {
        if (tam == tamMax) {
            tamMax = tamMax * 2;
            Object[] arrayAux = new Object[tamMax];
            System.arraycopy(array, 0, arrayAux, 0, array.length);
            array = arrayAux;
        }

        indiceUltimo = (indiceUltimo + 1) % tamMax;
        array[indiceUltimo] = elemento;
        tam++;
    }

    public void borrar() throws ExcepcionDeColaVacia {
        if (estaVacia()) {
            throw new ExcepcionDeColaVacia("Intenta borrar en una "
                    + "cola vacía");
        }
        indicePrimero = (indicePrimero + 1) % tamMax;
        tam--;
    }

    public Object primero() throws ExcepcionDeColaVacia {
```

```
        if (estaVacia()) {
            throw new ExcepcionDeColaVacia("Intenta obtener el primer "
                    + "elemento de una cola vacía");
        }
        return array[indicePrimero];
    }

    public int tamanno() {
        return tam;
    }

}
```