


# Desarrollo de aplicaciones Multiplataforma

# &

# Desarrollo de aplicaciones Web

## Documentación, refactorización y pruebas

Linkia FP

## Actividad

Realizar casos de pruebas unitarias en Java con JUnit de eclipse y a partir de un código realizar el grafo, la complejidad ciclomática y obtener los caminos independientes.

Aplicar la refactorización a un código y documentar un código en Java con JavaDoc en eclipse

## Objetivos

- Diseñar casos de prueba con Junit y Eclipse.
- Dado un código, realizar el grafo, la complejidad ciclomática y obtener los caminos independientes.
- Aplicar patrones de refactorización para optimizar el código Java.
- Realizar la documentación de una clase y generar un JavaDoc con Eclipse.

Actividad 1

***Patrones de refactorización***

En esta actividad aplicaremos algunos de los **patrones de refactorización** vistos en clase para optimizar el código.

## ¿Qué has de hacer en esta actividad?

**Para cada bloque de código** adjunto abajo:

1. **Crea una nueva clase** Java con el código (adjunta la evidencia)
2. Refactoriza el código
3. Adjunta el código una vez refactorizado (adjuntar evidencia)
4. **Explica** a continuación del código refactorizado **qué patrón has usado** y **comenta los cambios** que se han hecho**.**

# Bloques de código a utilizar en la actividad:

*Bloque 1:*

Tenemos una clase que tiene un atributo que necesita información y funciones propias. Qué patrón de refactorización tenemos que usar para añadir otra información del cliente como edad, dni, etc.

```
public class Pedido {
   private int id;
   private Cliente cliente;

   public Pedido(int id,Cliente cliente) {
      this.id = id;
      this.cliente = cliente;
   }
}
```

*Bloque 2:*

Qué patrón tenemos que usar para refactorizar una clase que hace el trabajo que debería ser hecho por dos clases.

```
public class Cliente {

   private String nombre;
   private String telefonoTrabajo;
   private String prefijoTelefonoTrabajo;
   private String telefonoCasa;
   private String prefijoTelefonoCasa;

   public String getTelefonoTrabajo() {
      return telefonoTrabajo;
   }
   public String getTelefonoCasa() {
      return telefonoCasa;
```

```
    }
}
```

## Solución de la actividad 1:

## Bloque 1:

```
public class Pedido {
   private int id;
   private String cliente;

   public Pedido(int id,String cliente) {
      this.id = id;
      this.cliente = cliente;
   }
}

public class Cliente {

        private String nombre;
        private int edad;
        private String dni;

        public Cliente() {

        }

        /**
         * @return the nombre
         */
        public String getNombre() {
                return nombre;
        }

        /**
         * @param nombre the nombre to set
         */
        public void setNombre(String nombre) {
                this.nombre = nombre;
        }

        /**
         * @return the edad
         */
        public int getEdad() {
                return edad;
        }

        /**
         * @param edad the edad to set
         */
        public void setEdad(int edad) {
                this.edad = edad;
        }

        /**
         * @return the dni
         */
        public String getDni() {
                return dni;
        }

        /**
         * @param dni the dni to set
         */
        public void setDni(String dni) {
                this.dni = dni;
```

```
        }

        public static void main(String[] args) {

                Cliente cliente = new Cliente();
                cliente.setNombre("roberto");
                cliente.setEdad(38);
                cliente.setDni("12345678T");

        }

}
```

Para realizar esta factorización hemos usado el patrón de refactorización Reemplazar datos y valores por objetos.

Bloque 2:

```
public class Cliente {

   private String nombre;
   private Telefono telefonoTrabajo;
   private Telefono telefonoCasa;

   public Telefono getTelefonoTrabajo() {
      return telefonoTrabajo;
   }
   public Telefono getTelefonoCasa() {
      return telefonoCasa;
   }
}

public class Telefono {

   private String numero;
   private String prefijo;

   public String getNumero() {
      return numero;
   }
   public String getPrefijo() {
      return prefijo;
   }
    public void setNumero(String numero) {
     this.numero = numero;
   }
   public void setPrefijo(String prefijo) {
      this.prefijo = prefijo
   }
}
```

Para realizar esta factorización hemos usado el patrón de refactorización Extraer Clase.

Actividad 2

***Documentación de una clase en Eclipse***

En esta actividad, realizaremos la **documentación de código Java**, usando las etiquetas (tags) del JavaDoc y la herramienta de generación de documentación automática JavaDoc de Eclipse.

## *¿Qué has de hacer en esta actividad?*

Dado el código adjunto abajo, los pasos a realizar son los siguientes:

1. **Crear la clase Stack** (tienes el código abajo)
2. **Comprobar que no tenga errores**. En el caso que tenga, toca solucionarlos**. Comenta que has hecho para solucionarlos** y adjunta una evidencia.
3. **Documentar**:
    a. la clase
    b. cada uno de los campos y métodos
    c. las partes principales del código

    Utiliza las etiquetas y comentarios apropiados. En este enlace podéis encontrar un resumen de los tipos de comentarios Java:

    http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html#comments

    Adjunta evidencia del código comentado.
    Es necesario que se vea vuestro nombre como autor del código.

4. **Generad la documentación JavaDoc** con Eclipse, adjuntar evidencia.
5. **Adjuntar** en la entrega el **código Java** correctamente documentado, así como los **documentos JavaDoc generados**

Bloque de código a utilizar en la actividad:

```
public class Stack {

	private int tamaño;
	private Vector<Integer> elementos;

	public Stack() {

		elementos = new Vector<Integer>();
		tamaño = 0;
	}

	public boolean stackVacio () {
		if (tamaño==0) {
			return true;
		}
		return false;
	}
```

```
        public void apilar ( Integer o ) {
                elementos.add(tamaño, o);
                tamaño++;
        }

        public Integer desapilar () {
                try {
                        if(stackVacio())
                                throw new ErrorStackVacio();
                        else {
                                return elementos.get(--tamaño);
                        }
                } catch(ErrorStackVacio error) {
                        System.out.println("ERROR: el stack está vacio");
                        return null;
                }
        }

        public int getNumElements() {
                return tamaño;
        }

        @SuppressWarnings("serial")
        class ErrorStackVacio extends Exception {
                public ErrorStackVacio() {
                        super();
                }
        }

}
```

## Solución de la actividad 2:

### Punto 1:

La clase Stack es correcta, solo se tiene que importar las librerías Java para poder usar la clase Vector.

### Punto 2-3:

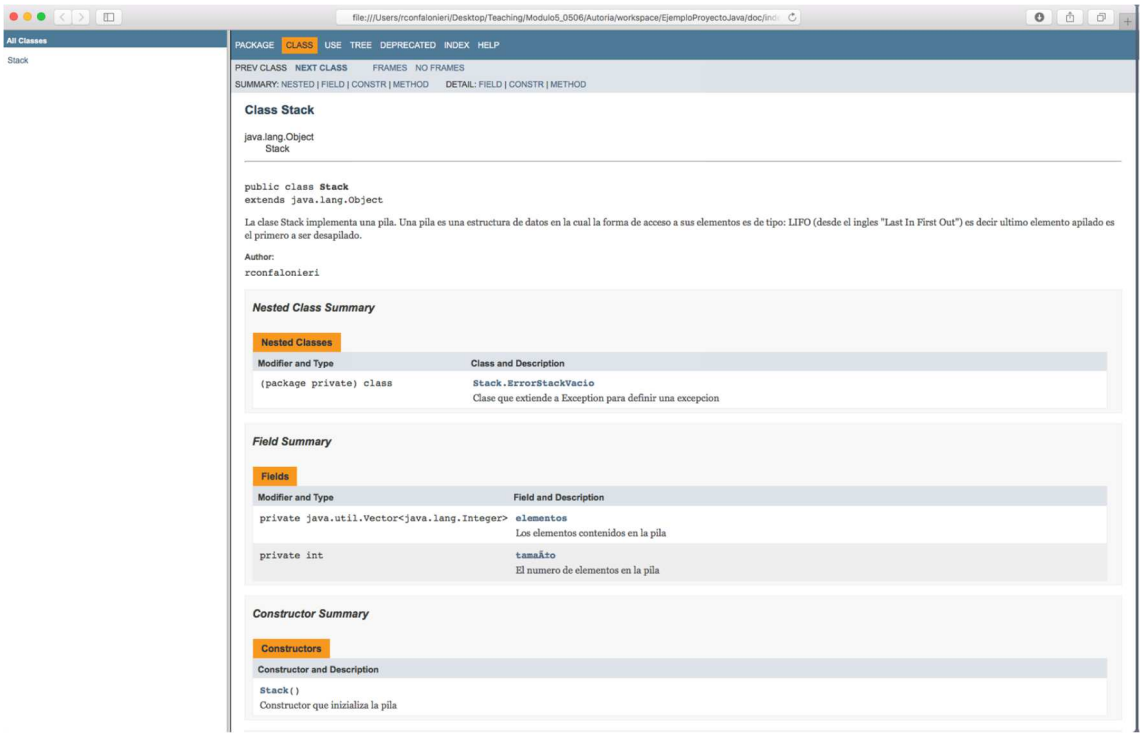
```
import java.util.Vector;
/**
* La clase @class{Stack}implementa una pila. Una pila es una estructura de datos en la
* cual la forma de acceso a sus elementos es de tipo:
* LIFO ("Last In First Out") es decir último elemento apilado es el primero a ser *desapilado.
*/
public class Stack {
         /**
         * El número de elementos en la pila
         */
         private int tamaño;
         /**
         * Los elementos contenidos en la pila
         */
         private Vector<Integer> elementos;
         /**
         * Constructor que inizializa la pila
         */

         public Stack() {

         elementos = new Vector<Integer>();
         tamaño = 0;
         }
         /**
         * Este método comprueba si la pila está vacia
         * @return boolean
         */
         public boolean stackVacio () {
                  if (tamaño==0) {
                  return true;
                  }
                  return false;
         }
         /**
         * Método para agregar un elemento a la pila
         * @param o entero
         */
         public void apilar ( Integer o ) {
                  elementos.add(tamaño, o);
                  tamaño++;
         }
```

```
	/**
	* Método para recuperar el último elemento apilado
	* @return int
	*/
	public Integer desapilar () {
	try {
		//si no hay elementos en la pila devuelvo una excepcion
		if(stackVacio())
			throw new ErrorStackVacio();
		//sino devuelvo el ultimo elemento apilado
		else {
			return elementos.get(--tamaño);
		}
		//gestion de la exception
		} catch(ErrorStackVacio error) {
			System.out .println("ERROR: el stack está vacio");
			return null;
		}
	}
	/**
	* Método que devuelve el numero de elementos en la pila
	* @return int
	*/
	public int getNumElements() {
		return tamaño;
			}
	/**
	* Clase que extiende a Exception para definir una excepcion
	*/
	@SuppressWarnings("serial")
		class ErrorStackVacio extends Exception {
		public ErrorStackVacio() {
		super();
		}
	}
}
```

## Punto 4:



All Classes

Stack

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

**Class Stack**

java.lang.Object
Stack

```
public class Stack
extends java.lang.Object
```

La clase Stack implementa una pila. Una pila es una estructura de datos en la cual la forma de acceso a sus elementos es de tipo: LIFO (desde el ingles "Last In First Out") es decir ultimo elemento apilado es el primero a ser desapilado.

Author:
rconfalonieri

***Nested Class Summary***

**Nested Classes**

| Modifier and Type | Class and Description |
|---|---|
| (package private) class | Stack.ErrorStackVacio<br>Clase que extiende a Exception para definir una excepcion |

***Field Summary***

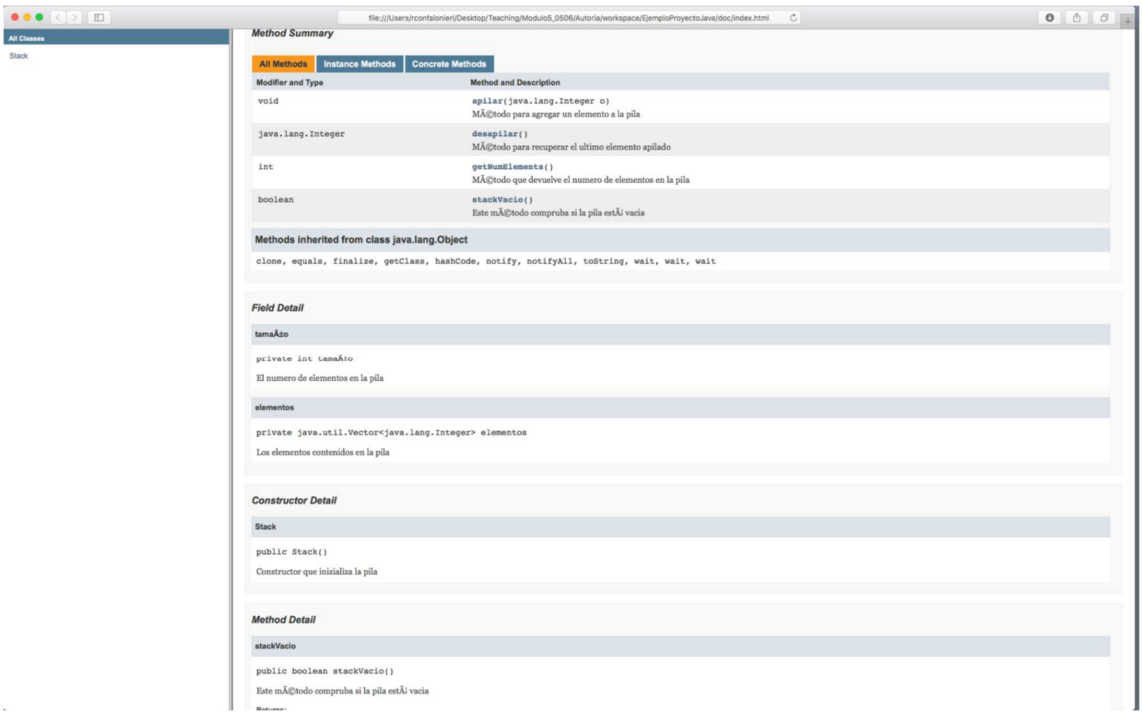
**Fields**

| Modifier and Type | Field and Description |
|---|---|
| private java.util.Vector<java.lang.Integer> | elementos<br>Los elementos contenidos en la pila |
| private int | tamaÃ±o<br>El numero de elementos en la pila |

***Constructor Summary***

**Constructors**

| Constructor and Description |
|---|
| Stack()<br>Constructor que inizializa la pila |



All Classes

Stack

***Method Summary***

**All Methods** | **Instance Methods** | **Concrete Methods**

| Modifier and Type | Method and Description |
|---|---|
| void | apilar(java.lang.Integer o)<br>MÃ©todo para agregar un elemento a la pila |
| java.lang.Integer | desapilar()<br>MÃ©todo para recuperar el ultimo elemento apilado |
| int | getNumElements()<br>MÃ©todo que devuelve el numero de elementos en la pila |
| boolean | stackVacio()<br>Este mÃ©todo compruba si la pila estÃ¡ vacia |

**Methods inherited from class java.lang.Object**

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

***Field Detail***

**tamaÃ±o**

```
private int tamaÃ±o
```

El numero de elementos en la pila

**elementos**

```
private java.util.Vector<java.lang.Integer> elementos
```

Los elementos contenidos en la pila

***Constructor Detail***

**Stack**

```
public Stack()
```

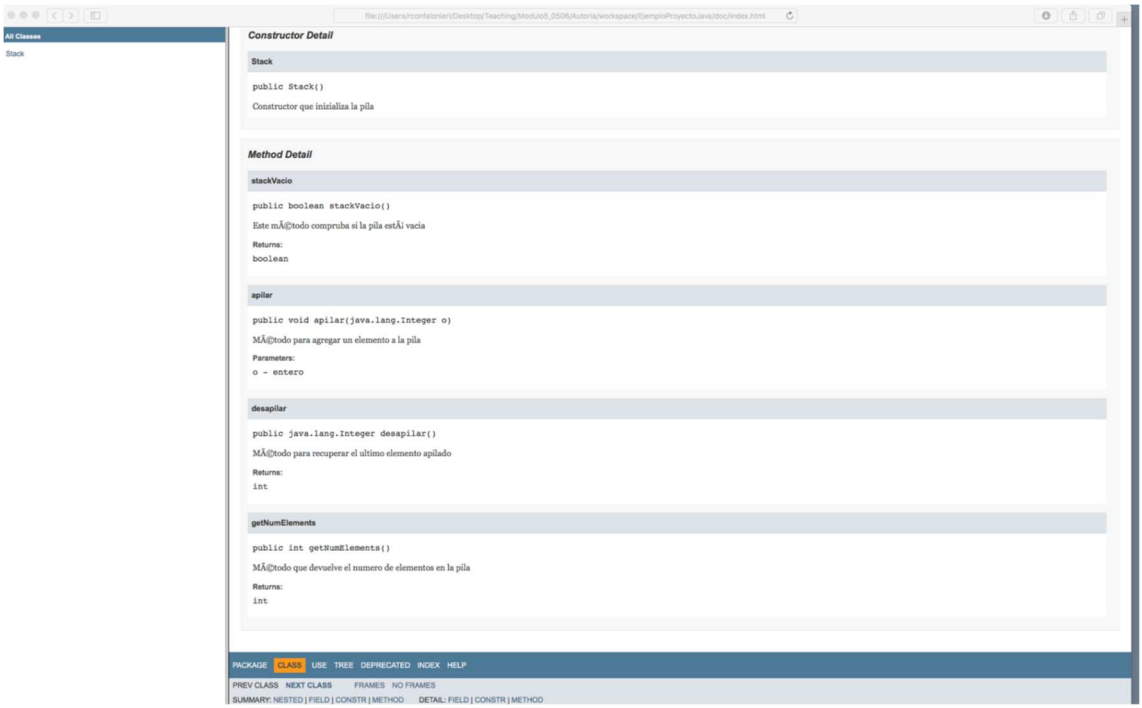
Constructor que inizializa la pila

***Method Detail***

**stackVacio**

```
public boolean stackVacio()
```

Este mÃ©todo compruba si la pila estÃ¡ vacia


All Classes
Stack
Constructor Detail
Stack
public Stack()
Constructor que inizializa la pila
Method Detail
stackVacio
public boolean stackVacio()
Este mÃ©todo compruba si la pila estÃ¡ vacia
Returns:
boolean
apilar
public void apilar(java.lang.Integer o)
MÃ©todo para agregar un elemento a la pila
Parameters:
o - entero
desapilar
public java.lang.Integer desapilar()
MÃ©todo para recuperar el ultimo elemento apilado
Returns:
int
getNumElements
public int getNumElements()
MÃ©todo que devuelve el numero de elementos en la pila
Returns:
int
PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Actividad 3

***Complejidad ciclomática***

En esta actividad, calcularemos la complejidad ciclomática de un programa.

## *¿Qué has de hacer en esta actividad?*

1. Crea el grafo de flujo del código adjunto abajo (adjunta evidencia).
    a. Tenéis herramientas online en internet para crear grafos de flujo
2. Calcula la complejidad ciclomática del código.
3. Identificar los caminos independientes del código.

Bloque de código a utilizar en la actividad:

```
ArrayList<Integer> vector = new ArrayList<Integer>();
int nr_par = 0;
int nr_impar = 0;
int i = 0;
       while (i < vector.size()) {

              if (vector.get(i) % 2 == 0) {
                     nr_par++;
              }
              else
                     nr_impar++;
              i++;

       }
System.out.println("nr_par "+nr_par);
System.out.println("nr_impar "+nr_par);
```

### Solución de la actividad 3:

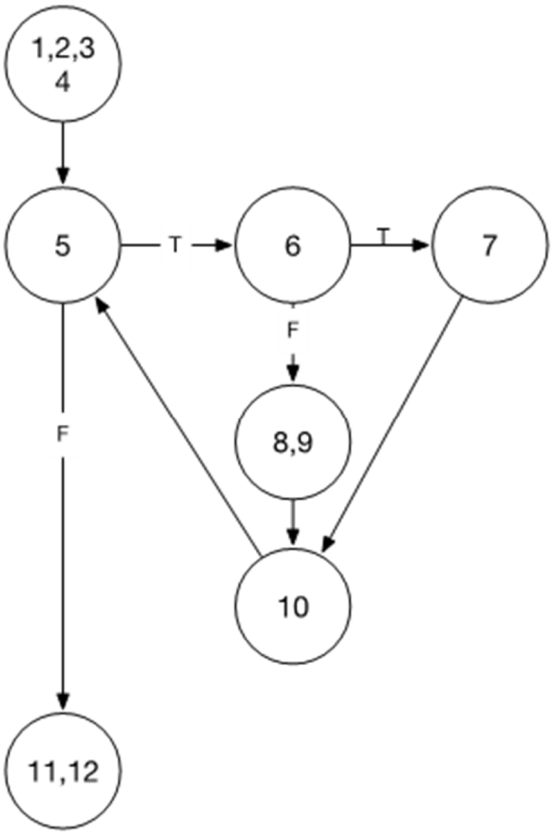
### 1. Para crear el grafo de flujo, antes de todo, enumeramos las instrucciones de nuestro codigo:

```
1. ArrayList<Integer> vector = new ArrayList<Integer>();
2. int nr_par = 0;
3. int nr_impar = 0;
4. int i = 0;
5. while (i < vector.size()) {

6. if (vector.get(i) % 2 == 0) {
7. nr_par++;
              }
8. else
9. nr_impar++;
10. i++;

       }
11. System.out.println("nr_par "+nr_par);
12. System.out.println("nr_impar "+nr_par);
```

Agrupando las instrucciones el grafo de flujo nos queda así:




El número do nodos es = 7
El número de aristas es = 8

2. La complejidad ciclomática del programa es:

V(G) = V – N + 2 = 8 -7 + 2 = 3

3. Los caminos independientes son 4:

- Camino 1: (1,2,3,4) – 5 –6—7—10—5—(11,12)
- Caminos 2: (1,2,3,4) – 5 –6—(8,9)—10—5—(11,12)
- Caminos 3: (1,2,3,4) – 5 – (11,12)

Actividad 4

***Implementación de pruebas***

Dado el código Java de la actividad 3, implementar las pruebas unitarias que permiten comprobar el correcto funcionamiento de la clase Stack. Para esto, usaremos Eclipse y la librería jUnit.

## *¿Qué has de hacer en esta actividad?*

1. Crear una clase de prueba (New JUnit Test Case) llamada TestStack (un caso de prueba JUnit) sobre la clase Stack usando el código adjunto abajo.
2. Implementar los diferentes métodos de la clase TestStack.
    a. Añade al menos un comentario en cada método explicando brevemente que hace la prueba.
    b. Adjunta evidencia del código
3. Ejecutar el test
    a. comprobar que no hay errores en tus pruebas.
    b. Adjuntar una captura de pantalla de la ejecución de los tests implementados en jUnit.

Bloque de código a utilizar en la actividad:

```
public class TestStack {

	@SuppressWarnings("unused")
	private Stack stackConElementos;
	@SuppressWarnings("unused")
	private Stack stackSinElementos;

	@Before
	public void setUp() throws Exception {
		//configurar el test
	}

	@Test
	public void testStackVacio() {
		fail("Not yet implemented");

	}

	@Test
	public void testApilar() {
		fail("Not yet implemented");
	}

	@Test
	public void testDesapilarStackSinElementos() {
		fail("Not yet implemented");
	}
```

```
	@Test
	public void testDesapilarStackConElementos() {
		fail("Not yet implemented");
	}

	@Test
	public void testGetNumElements() {
		fail("Not yet implemented");
	}

}
```

## Solución de la actividad 4:

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
public class TestStack {
private Stack stackConElementos;
private Stack stackSinElementos;
@Before
public void setUp() throws Exception {
	stackConElementos = new Stack();
	stackConElementos.apilar(1);
	stackConElementos.apilar(2);
	stackSinElementos = new Stack();
	}
	@Test
	public void testStackVacio() {
	//fail("Not yet implemented");
	assertEquals(stackSinElementos.getNumElements(),0);
	}
	@Test
	public void testApilar() {
	//fail("Not yet implemented");
	int i = 5;
	int dim = stackConElementos.getNumElements();
	stackConElementos.apilar(i);
	assertEquals(stackConElementos.getNumElements(),dim+1);
	}
	@Test
	public void testDesapilarStackSinElementos() {
	//fail("Not yet implemented");
	int dim = stackSinElementos.getNumElements();
	Integer i = stackSinElementos.desapilar();
	assertEquals (i, null);
	assertEquals(stackSinElementos.getNumElements(),dim);
	}
	@Test
	public void testDesapilarStackConElementos() {
	//fail("Not yet implemented");
	int mida = stackConElementos.getNumElements();
	int i = stackConElementos.desapilar();
	assertEquals (i, 2);
	assertEquals(stackConElementos.getNumElements(),mida-1);
	}
	@Test
	public void testGetNumElements() {
	//fail("Not yet implemented");
	int valor = 0;
	valor =stackConElementos.getNumElements();
	assertEquals(valor,2);
	}
}
```