JUNIT. Pruebas Unitarias

Dpto. de Ingeniería de Sistemas Telemáticos
http://www.lab.dit.upm.es/~lprg

Introducción

- Un **programa** es **aceptable** cuando:
  - Hace lo que se acordó que debía hacer en las especificaciones.
  - No hace lo que no debe hacer.
- "Un programador **jamás** debería **entregar un programa sin haberlo probado**. Igualmente, quien recibe un programa de otro jamás debería aceptarlo sin haberlo probado.
- Para aprobar una práctica ésta debe pasar las pruebas funcionales.
- Cualquier **funcionalidad** de un programa **sin** una **prueba** automatizada, simplemente **no existe**" (Extreme Programming Explained", de Kent Beck)

Pruebas Unitarias Sistemáticas (1): Conceptos

- **Prueba unitaria**: una prueba individual de un método o clase.
- **Prueba unitaria ad-hoc**: por ejemplo, cuando creamos un objeto de cierta clase con BlueJ e invocamos manualmente un método del mismo con distintas entradas para ver si funciona.
- Sin embargo, con este tipo de pruebas no se puede trabajar eficiente y sistemáticamente.
- Cada vez que cambiamos algo en el método o clase tendríamos que volver a pasar todas las pruebas para asegurarnos de que "nada se ha descabalado". Es decir, realizar **pruebas de regresión**.
- Para ello, vendría muy bien algo que nos permitiera **definir sistemáticamente** una serie de **pruebas y ejecutarlas automáticamente**, tantas veces como necesitáramos.
- **JUNIT** nos permite hacer esto. JUNIT + BlueJ todavía mejor.

Pruebas Unitarias Sistemáticas (2): Procedimiento

- Antes de implementar una determinada funcionalidad, piensa cómo deberías probarla para verificar que se comporta correctamente. Esto permite desarrollar la funcionalidad teniendo las ideas muy claras de lo que debería hacer.
- Escribe el código que implementa la funcionalidad deseada.
- Escribe el código de las pruebas inmediatamente después.
- Ejecuta las pruebas que hiciste.
- Corrige la unidad de código que implementa la funcionalidad deseada **hasta que pase todas y cada una de las pruebas**.
- **Al añadir una nueva funcionalidad, repite el ciclo**: piensa en cómo probarla, codifica la funcionalidad, codifica las pruebas, ejecuta todas las pruebas que hiciste (nuevas y viejas). No sigas hasta que el código pase **absolutamente todas** las pruebas.
- Así una y otra vez para cada nueva funcionalidad que implementes. Lo vemos con un ejemplo.

Ejemplo Procedimiento Prueba con JUNIT (1)

- Desarrollar un método estático que tome un *array* de enteros como argumento y devuelva el mayor valor encontrado en el *array*.

```
public class MayorNumero {
   /**
     * Devuelve el elemento de mayor valor de una lista
     * @param    list    Un array de enteros
     * @return           El entero de mayor valor de la lista
   */
   public static int obt_mayorNumero(int lista[]) {
     return 0;   // para que compile hasta que desarrollemos el método
   }
}
```


Ejemplo Procedimiento Prueba con JUNIT (2)

- ¿Qué pruebas se te ocurren para el método **obt_mayorNumero**?
  - ¿Qué ocurre para un *array* con valores cualesquiera (el caso normal)?

    [3, 7, 9, 8] → 9
  - ¿Qué ocurre si el mayor elemento se encuentra al principio, en medio o al final del *array*?

    [9, 7, 8] → 9; [7, 9, 8] → 9; [7, 8, 9] → 9
  - ¿Y si el mayor elemento se encuentra duplicado en el *array*?

    [9, 7, 9, 8] → 9
  - ¿Y si sólo hay un elemento en el *array*?

    [7] → 7
  - ¿Y si el *array* está compuesto por elementos negativos?

    [-4, -6, -7, -22] → -4

Ejemplo Procedimiento Prueba con JUNIT (3)

- Escribimos el código del método **obt_mayorNumero**:

```
/**
 * Devuelve el elemento de mayor valor de una lista
 *
 * @param  list   Un array de enteros
 * @return        El entero de mayor valor de la lista
*/
public static int obt_mayorNumero(int lista[]) {
    int indice, max = Integer.MAX_VALUE;
    for (indice = 0; indice < lista.length-1; indice++) {
      if (lista[indice] > max) {
                max = lista[indice];
      }
    }
    return max;
}
```


Ejemplo Procedimiento Prueba con JUNIT (4)

- Escribimos el código de las pruebas (1):

```
import junit.framework.*;

public class TestMayorNumero extends TestCase {
   public TestMayorNumero() {
   }

   public void testSimple() {
      assertEquals(9, MayorNumero.obt_mayorNumero(new int[] {3, 7, 9, 8}));
   }

   public void testOrden() {
      assertEquals(9, MayorNumero.obt_mayorNumero(new int[] {9, 7, 8}));
      assertEquals(9, MayorNumero.obt_mayorNumero(new int[] {7, 9, 8}));
      assertEquals(9, MayorNumero.obt_mayorNumero(new int[] {7, 8, 9}));
   }

// Sigue en la próxima transparencia
```

Ejemplo Procedimiento Prueba con JUNIT (5)

- Escribimos el código de las pruebas (2):

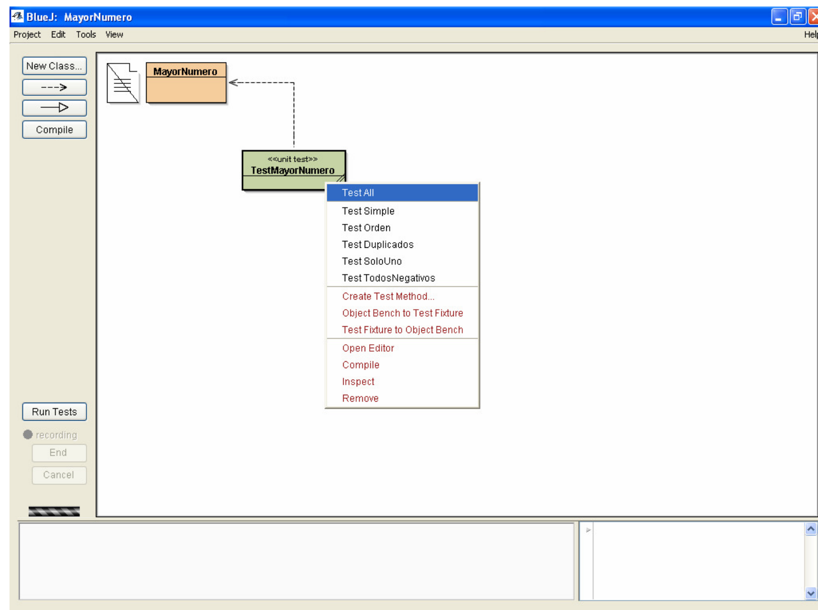
```
// Sigue de la transparencia anterior

public void testDuplicados() {
    assertEquals(9, MayorNumero.obt_mayorNumero(new int[] {9, 7, 9, 8}));
  }
  public void testSoloUno() {
    assertEquals(7, MayorNumero.obt_mayorNumero(new int[] {7}));
  }
  public void testTodosNegativos() {
    assertEquals(-4, MayorNumero.obt_mayorNumero(new int[] {-4, -6, -7, 22}));
  }

} // Fin de la clase
```

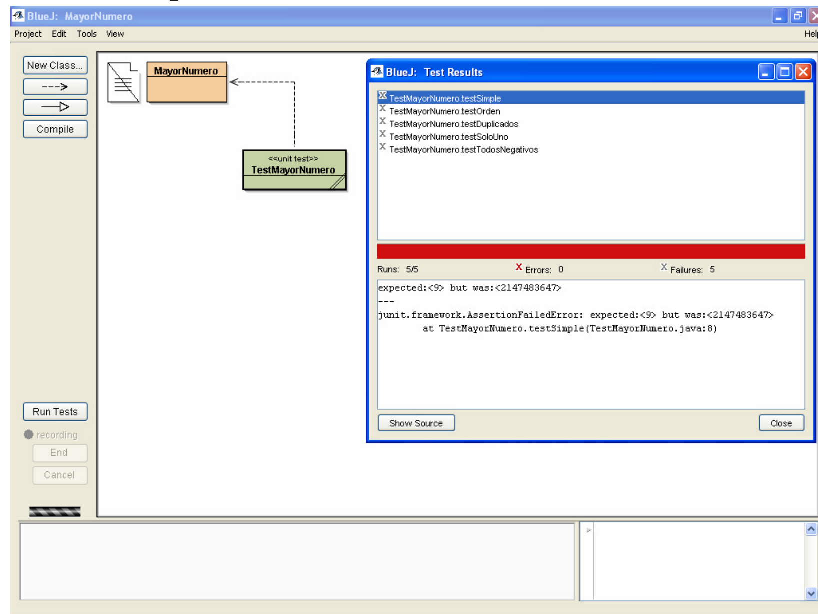

Ejemplo Procedimiento Prueba con JUNIT (6)

- Ejecutamos las pruebas:



Ejemplo Procedimiento Prueba con JUNIT (7)

- Ejecutamos las pruebas:



Ejemplo Procedimiento Prueba con JUNIT (8)

- ¿Qué está mal?: PREGUNTAR A ALUMNOS

```
/**
 * Devuelve el elemento de mayor valor de una lista
 *
 * @param  list   Un array de enteros
 * @return        El entero de mayor valor de la lista
*/
public static int obt_mayorNumero(int lista[]) {
    int indice, max = Integer.MAX_VALUE;
    for (indice = 0; indice < lista.length-1; indice++) {
      if (lista[indice] > max) {
                max = lista[indice];
      }
    }
    return max;
}
```

Ejemplo Procedimiento Prueba con JUNIT (9)

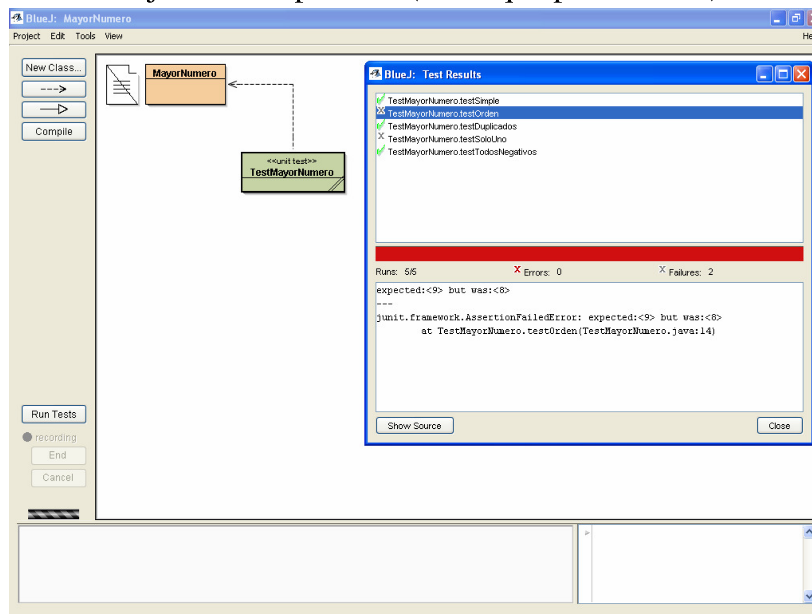
- ¿Qué está mal?:

```
/**
 * Devuelve el elemento de mayor valor de una lista
 *
 * @param  list   Un array de enteros
 * @return        El entero de mayor valor de la lista
*/
public static int obt_mayorNumero(int lista[]) {
    int indice, max = Integer.MAX_VALUE;
    for (indice = 0; indice < lista.length-1; indice++) {
       if (lista[indice] > max) {
                max = lista[indice];
       }
    }
    return max;
}
```

Integer.MIN_VALUE

Ejemplo Procedimiento Prueba con JUNIT (10)

- Volvemos a ejecutar las pruebas (hasta que pasen todas):



Ejemplo Procedimiento Prueba con JUNIT (11)

- ¿Qué está mal?:

Falla en este "assert", devuelve el 8 y no el 9

```
public void testOrden() {
    assertEquals(9, MayorNumero.obt_mayorNumero(new int[] {9, 7, 8}));
    assertEquals(9, MayorNumero.obt_mayorNumero(new int[] {7, 9, 8}));
    assertEquals(9, MayorNumero.obt_mayorNumero(new int[] {7, 8, 9}));
}

public static int obt_mayorNumero(int lista[]) {
    int indice, max = Integer.MAX_VALUE;
    for (indice = 0; indice < lista.length-1; indice++) {
        if (lista[indice] > max) {
            max = lista[indice];
        }
    }
    return max;
}
```

PREGUNTAR A ALUMNOS
ejemplo de que siempre hay
que analizar los extremos

Ejemplo Procedimiento Prueba con JUNIT (12)

- ¿Qué está mal?:

Falla en este "assert", devuelve el 8 y no el 9

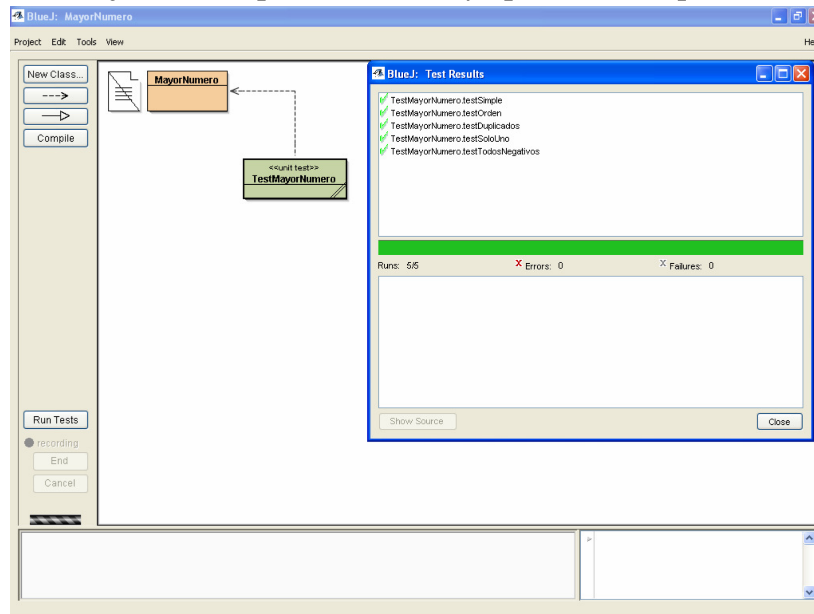
```
public void testOrden() {
    assertEquals(9, MayorNumero.obt_mayorNumero(new int[] {9, 7, 8}));
    assertEquals(9, MayorNumero.obt_mayorNumero(new int[] {7, 9, 8}));
    assertEquals(9, MayorNumero.obt_mayorNumero(new int[] {7, 8, 9}));
}

public static int obt_mayorNumero(int lista[]) {
    int indice, max = Integer.MAX_VALUE;
    for (indice = 0; indice < lista.length-1; indice++) {
        if (lista[indice] > max) {
            max = lista[indice];
        }
    }
    return max;
}
```

lista.length

Ejemplo Procedimiento Prueba con JUNIT (13)

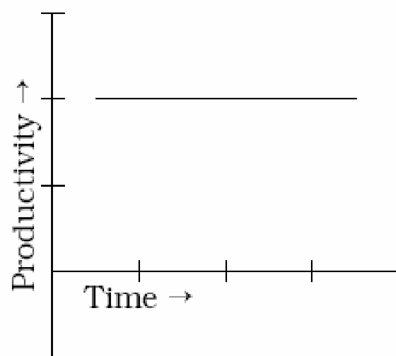
- Otra vez a ejecutar las pruebas (ahora ya pasan todas, podemos seguir):



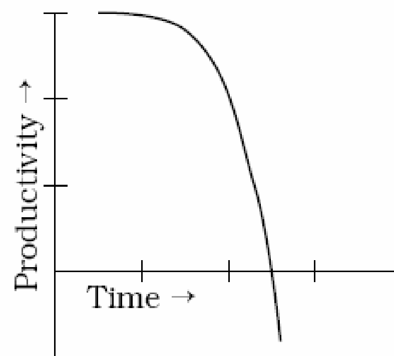
Ventajas de Probar Progresivamente

- Este método puede parecer extremo (casi es *Extreme Programming*), pero normalmente ahorra tiempo y esfuerzo:

Si pruebas mientras desarrollas



Si desarrollas todo y al final pruebas



- Lo mejor es un término medio: cada funcionalidad significativa que desarrolles pruébala antes de seguir.

Pruebas con JUNIT: Explicación Formal (1)

Marco para desarrollar pruebas unitarias (1)

```
línea 1     import junit.framework.*;
-
línea 3     public class Pruebas extends TestCase {
-             // variables_privadas
-
-             public Pruebas ( )  {
-               // inicialización
-             }
-
línea 10      public void testXXX ( )  {
-               // código para verificar que el método a probar funciona correctamente
-             }
-
-             public static void main (String[ ]  args) {
línea 15        junit.textui.TestRunner.run(Pruebas.class);   // ejecución consola texto
-               // junit.swingui.TestRunner.run(Pruebas.class);   // ejecución ventana
-             }
-           }  // fin de clase
```


Pruebas con JUNIT: Explicación Formal (2)

Marco para desarrollar pruebas unitarias (2):

a) (línea 1). **Importar las clases de JUNIT** necesarias.

b) (línea 3). Toda clase que contenga pruebas ha de **heredar de la clase "TestCase"**. Ésta contiene la mayoría de funcionalidad necesaria para realizar pruebas, incluyendo los "assert".

c) (línea 10). **Todo método que empiece por "test" en la clase será ejecutado automáticamente** por JUNIT. Así han de empezar todos los métodos con casos de prueba.

Otros detalles:

a) Cada método que sólo trate un caso o aspecto a probar.

b) (línea 15). Si no usas BlueJ puedes ejecutar las pruebas por consola.

Pruebas con JUNIT: Explicación Formal (3)

Métodos de comprobación (1):

assertEquals (valor_esperado, valor_real)

- Comprueba si “valor_esperado” y “valor_real” son iguales.
- “valor_real” es la salida que genera el método bajo prueba, “valor_esperado” es lo que debería generar el método bajo prueba si funciona correctamente.
- Estos valores pueden ser de cualquier tipo nativo, de tipo String y Object (hay distintos métodos para cada caso). Ojo, que si pasas arrays no los compara elemento a elemento, sólo la referencia.

assertTrue (boolean condición)

- Comprueba que la condición es cierta.

assertFalse (boolean condición)

- Comprueba que la condición es falsa.

assertSame (Objeto esperado, Objeto obtenido)

- Comprueba que “obtenido” y “esperado” se refieren al mismo objeto.

Pruebas con JUNIT: Explicación Formal (4)

Métodos de comprobación (2):

assertNotSame (Objeto esperado, Objeto obtenido)

- Comprueba que “obtenido” y “esperado” **no** se refieren al mismo objeto.

assertNull (Objeto objeto)

- Comprueba que el objeto indicado es nulo (referencia a *null*)

assertNotNull (Objeto objeto)

- Comprueba que el objeto pasado como parámetro no sea nulo (que referencia a un objeto inicializado en memoria).

fail (String mensaje)

- Si en el código se alcanza y ejecuta esta sentencia, la prueba fallará.
- Se usa para marcar secciones de código que no deberían ejecutarse si todo funcionara correctamente.
- Por ejemplo, justo después de una excepción para comprobar si ésta se lanza correctamente.

Pruebas con JUNIT: Explicación Formal (5)

Uso de los Métodos de comprobación:

- Puedes poner en una función de prueba ("testXXX") tantos métodos de comprobación como necesites para implementar el caso de prueba concreto.
- A la hora de ejecutar la función prueba ("testXXX"), en cuanto falle uno de los métodos de comprobación se para la ejecución. No se ejecutan el resto de métodos de comprobación tras el que falló.
- En ese caso, antes de seguir tienes que corregir el fallo que se ha producido. No sigas hasta entonces: "No vivas con ventanas rotas".

Pruebas con JUNIT: Explicación Formal (6)

JUNIT y Excepciones (1)

- En general hay que comprobar que un método lanza todas las excepciones que se han declarado en el mismo cuando debe. Y que no las lanza cuando no hay motivo para ello. Esta es la utilidad del método fail.

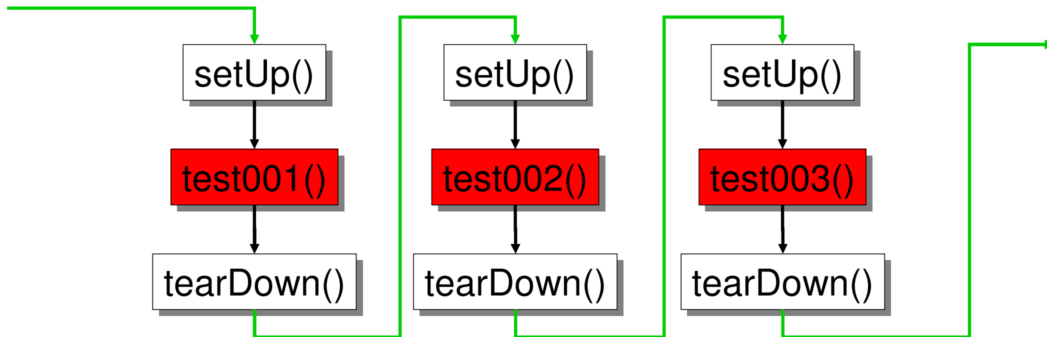
```
public void testExcepcionOrdenarListaNula( ) {
    try {
        ordena_lista(null);
        fail("Debería haber lanzado una excepción");
    } catch (RuntimeException e) { }
}
```

- Si ordena_lista está correctamente programado, debería lanzar una excepción al intentar ordenar una lista nula y entonces el método "fail" no debería ejecutarse nunca.

setUp & tearDown

- Se pueden poner métodos para envolver las pruebas:

```
public void setUp() { ...; }
public void tearDown() { ...; }
```



¿Qué hay que probar? (1)

- Comprobar que se obtienen los resultados esperados. En caso de que el código funcione correctamente, ¿cómo lo sabré?
- Prueba en los extremos pues es donde "viven" la mayoría de los errores.
- Algunos síntomas de que hay un error en los extremos pueden ser: valores a null o vacíos cuando no debería ser así, valores muy superiores a lo esperado, elementos duplicados en listas o que desaparecen, listas desordenadas cuando deberían estar ordenadas, acciones que ocurren fuera de orden, etc.
- **C onformance**. ¿Se muestran los valores con el formato esperado?
- **O rdering**. ¿Se presentan los valores apropiadamente ordenados o desordenados?
- **R ange**. ¿Los valores generados están dentro del rango esperado [max, min]?
- **R eference**. ¿Se refiere el código a algo externo que no está bajo su control?
- **E xistence**. ¿Existe el valor (es no nulo), está presente en un array, etc?
- **C ardinality**. ¿Se obtiene el número deseado de elementos o valores?
- **T ime**. ¿Ocurren las cosas en orden?

¿Qué hay que probar? (2)

- Una forma de probar que algo funciona es hacer las cosas de dos formas distintas y ver si dan el mismo resultado.
- También es interesante forzar a que se den determinados errores para ver si el sistema los trata adecuadamente.
- “Probar un programa es ejercitarlo con la peor intención a fin de encontrarle fallos”.

Prueba de Programas

- **Tipos de pruebas**:
  - Pruebas de caja blanca: analizar el propio código.
  - Pruebas de caja negra: sin ver el código, probar la funcionalidad según especificaciones.
  - Pruebas de integración: probar cómo funciona el sistema completo, compuesto por módulos distintos.
  - Pruebas de aceptación: realizadas por el cliente para ver si se le entrega lo que pidió.
  - Pruebas de regresión: tras añadir algo nuevo, para ver que no se ha descabalado la funcionalidad que ya había.
  - Pruebas de robustez o solidez, de aguante, de prestaciones, etc.
- **Cobertura de las pruebas**: porcentaje de código (caja blanca) o de funcionalidades y casos (caja negra) que hemos probado del total posible.

Caja Negra y Caja Blanca

- De caja negra
  - cuando conocemos lo que tiene que hacer el código ejercitamos todo lo que tiene que hacer
  - pruebas funcionales
- De caja blanca
  - cuando conocemos el código fuente forzamos la ejecución de todo el código
  - pruebas estructurales

caja negra

Casos de prueba

- ¿Qué hay que probar?
  - todo lo que dice la especificación:
    - manual
    - instrucciones
    - otra documentación
- Cobertura del 100%
  - si va tachando lo que se va probando, 100% es cuando todo está tachado
    - funcionamiento correcto
    - detección y reporte de errores

caja negra

Datos de prueba

- ¿Con qué datos se prueba?
  1. divida el espacio de datos en clases de equivalencia
     - clase de equivalencia:
       datos que provocan el "mismo comportamiento"
     - no parece que deban provocar comportamientos diferentes
  2. elija un dato "normal" de clase de equivalencia
  3. pruebe con todos los datos "frontera":
     valores extremos de la clase de equivalencia

caja negra

Datos de prueba

- Añada aquellos casos en los que sospeche que el programador puede haberse equivocado
- Pruebe todas las combinaciones
  { datos × comportamiento }

caja negra

Ejercicio e1

- Constructor
  - clave normal
  - clave null
  - clave “”
  - clave con 1 carácter
  - clave con caracteres fuera del alfabeto
  - ...

caja negra

Ejercicio e1

- int codifica (char c)
  - caracteres normales en el alfabeto
  - caracteres en los extremos del alfabeto
  - caracteres fuera del alfabeto
  - caracteres sospechosos
  - ...
- char descodifica(int x)
- String cifra(String texto)
- String descifra(String criptograma)

caja blanca

Casos de prueba

- ¿Qué hay que probar?
  - ejecutar al menos una vez cada sentencia
    - cobertura de sentencias
  - ejecutar al menos una vez cada condición con resultado cierto y falso
    - cobertura de ramas
- Método
  - si va tachando el código probado 100% es cuando todo esté tachado

caja blanca

Casos de prueba

- if (...)
  - si T, si F
- switch (...)
  - cada ‘case’ + ‘default’
- cobertura de bucles
  - for -> 3 pruebas: 0 veces, 1 vez, n>1 veces
  - repeat -> 2 pruebas: 1 vez, n>1 veces
  - while -> 3 pruebas: 0 veces, 1 vez, n>1 veces

caja blanca

Ejercicio e1

- String cifra(String texto)
  - texto “”
  - texto con 1 carácter en el alfabeto
  - texto con menos caracteres que la clave
  - texto con tantos caracteres como la clave
  - texto con 1 carácter más que la clave
  - texto con más caracteres que la clave
  - texto con caracteres fuera del alfabeto
  - ...

Estudiar:

- En la Web de la asignatura se encuentra esta documentación dada en clase y otros apuntes adicionales.
- Como tareas a realizar esta semana, leerse toda esta documentación y navegar por la Web de la asignatura.
- Leerse las normas del laboratorio publicadas en la Web: cómo pedir cuenta, organización del laboratorio, uso de los puestos, uso del entorno de ventanas del laboratorio, etc.
- Quien tenga ordenador en casa, descargarse el JDK 6 y el entorno de programación BlueJ. Instalarlo siguiendo las instrucciones en el laboratorio y documentación de cada programa.

CifradorTest

Class Edit Tools Options

Compile | Undo | Cut | Copy | Paste | Find... | Find Next | Close — Implementation

```java
package lprg.el.cifrador;

/**
 * Clase para probar la funcionalidad de la clase CifradorReferencia, que implementa un algoritmo de cifrado según el método de Vigenere.
 * Se realizan los siguientes casos de prueba:
 *
 * - testCodificacionAlfabeto   Para probar que el cifrador funciona correctamente en caso de solicitarse la codificación de caracteres normales.
 * - testXXX
 * ...
 * - testZZZ
 *
 * @author  Luis Enrique García Fernández
 * @version Versión 1.0  18/02/2007
 */
public class CifradorTest extends junit.framework.TestCase
{
    /**
     * Método que devuelve un cifrador con una clave concreta.
     * @param clave  Una cadena con la clave que vamos a usar para cifrar
     * @return Un objeto de la clase CifradorReferencia para cifrar textos según la clave que se indicó como parámetro de entrada.
     */
    private Vigenere getCifrador(String clave)
     {
         return new CifradorReferencia(clave);
        // return new MiCifrador(clave);
    }


    /**
     *  Test que emplearemos para probar que el cifrador funciona correctamente en caso de solicitarse la codificación de caracteres normales.
     */
    public void testCodificacionAlfabeto()
    {
        Vigenere cifrador = getCifrador(new String("A"));
        assertEquals(cifrador.codifica('A'), 0);
        assertEquals(cifrador.codifica('B'), 1);
        assertEquals(cifrador.codifica('C'), 2);
    }

```

Class compiled - no syntax errors

saved