

El testing como parte del proceso de calidad del software

Introducción al testing unitario

Contenido



- Testing ad-hoc y sistematización
- Testing unitario

Testing “ad hoc”

Todo programador está habituado al testing.

En muchos casos, la forma en la que hacemos testing es “ad hoc”, es decir, no sistemática.

- Ej: supongamos que queremos testear la siguiente aplicación:

Testing “ad hoc”

```
public class maxApplication {  
    public static void main(String[] args) {  
        if (args.length != 2) {  
            System.out.println("uso: maxApplication <int> <int>");  
        }  
        else {  
            int a = Integer.parseInt(args[0]);  
            int b = Integer.parseInt(args[1]);  
            if (a>b) {  
                System.out.println(a);  
            }  
            else {  
                System.out.println(b);  
            }  
        }  
    }  
}
```

Testing “ad hoc”

- En este caso, podemos probar la aplicación directamente desde la línea de comandos:

```
$ java maxApplication 2 3  
3  
$
```

Testing “ad hoc”

- En este caso, podemos probar la aplicación directamente desde la línea de comandos:

```
$ java maxApplication 2 3  
3  
  
$ java maxApplication 12345 987394032  
987394032  
  
$
```

Testing “ad hoc”

- En este caso, podemos probar la aplicación directamente desde la línea de comandos:

```
$ java maxApplication 2 3  
3  
  
$ java maxApplication 12345 987394032  
987394032  
  
$ java maxApplication -453 -4  
-453  
  
$
```

Testing “ad hoc”

- En este caso, podemos probar la aplicación directamente desde la línea de comandos:

```
$ java maxApplication 2 3  
3  
  
$ java maxApplication 12345 987394032  
987394032  
  
$ java maxApplication -453 -4  
-453  
  
$ java maxApplication 23 5 3  
uso: maxApplication <int> <int>  
  
$
```

Testing “ad hoc”

- Ej: En caso que lo que tengamos no sea una aplicación, sino una **función**:

```
public static int max(int a, int b) {  
    if (a>b)  
        return a;  
    else  
        return b;  
}
```

- el testing “ad hoc” se vuelve más difícil: tenemos que programar un arnés para el método (i.e., un “main” que lo invoque).

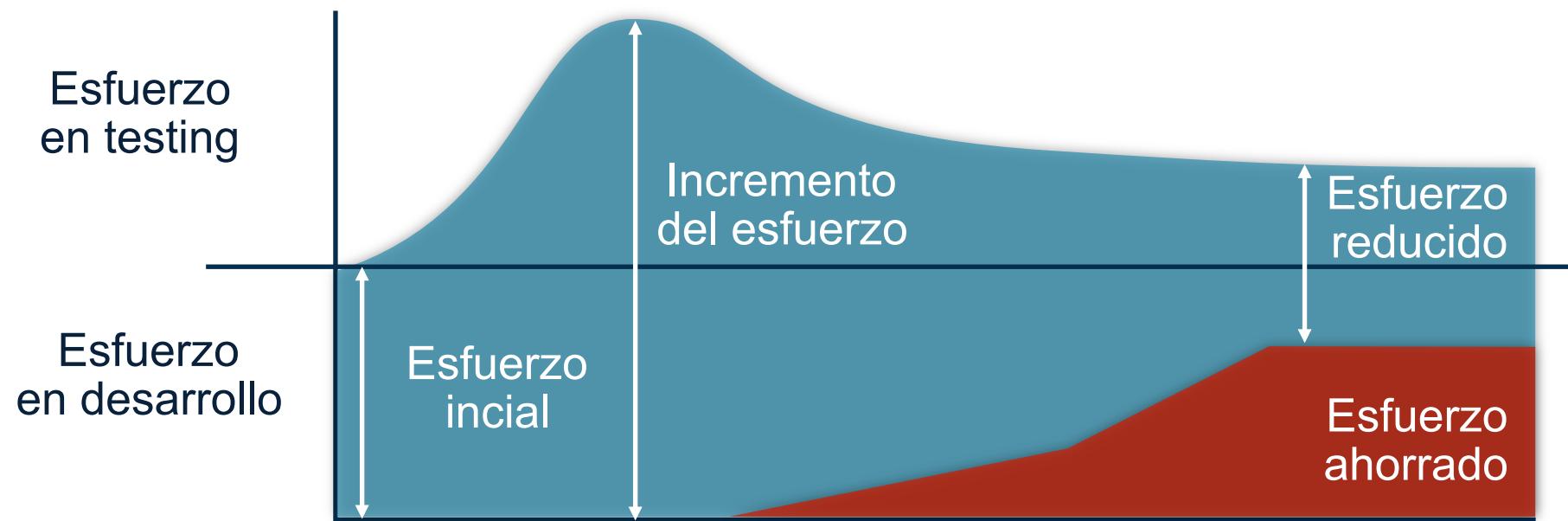
Dificultades del testing “ad hoc”

- Es simple para testear aplicaciones,
 - pero no almacena los tests para pruebas futuras.
- Requiere la construcción manual de “arneses” para la prueba de funcionalidades “internas” (no directamente invocables desde la interfaz de la aplicación).
- Requiere la inspección humana de la salida de los tests:
 - se decide manualmente si el test pasó o no.

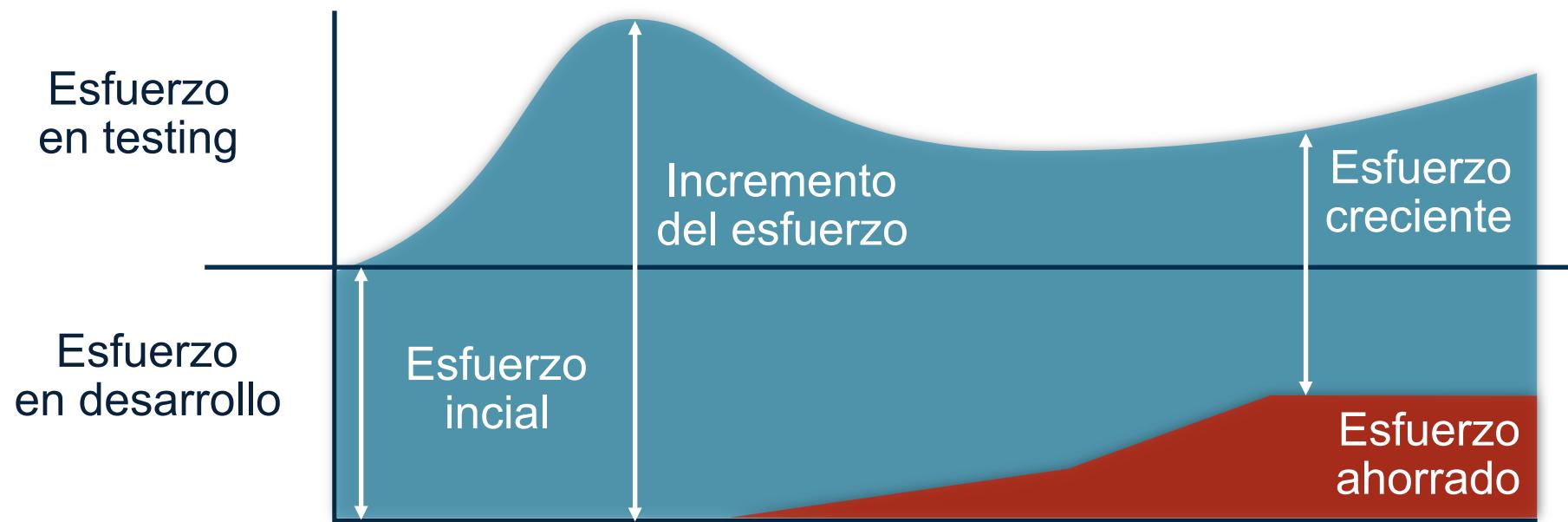
Automatización de testing

- Cuando el testing se vuelve **complejo y caro**
 - como el **testing** es “**opcional**”, hay **una tendencia** a abandonarlo;
 - o se lo intenta simplificar a través de su **automatización**.
- El **costo** de realizar y mantener un testing automatizado **debe ser menor** que el costo ahorrado por la reparación de defectos encontrados tardíamente.

Economía de la automatización



Economía de la automatización



Objetivos de la automatización

- Aumentar la calidad:
 - utilizando tests como especificaciones;
 - previniendo la reintroducción de defectos (en lugar de repararlos);
 - localizando los defectos (ubicación y causa).
- Comprender el objeto bajo prueba:
 - utilizando los tests como documentación.
- Reducir (y no introducir) riesgos:
 - utilizando los tests como “red de seguridad”;
 - sin introducir cambios en el objeto bajo prueba.

Requisitos para la automatización

- Ser sencillos de ejecutar:
 - completamente automáticos.
 - self-checking.
 - repetibles.
- Ser sencillos de escribir y mantener:
 - simplicidad.
 - expresividad.
 - separación de incumbencias.
- Requerir mantenimiento mínimo: robustez.

Problemas de la automatización

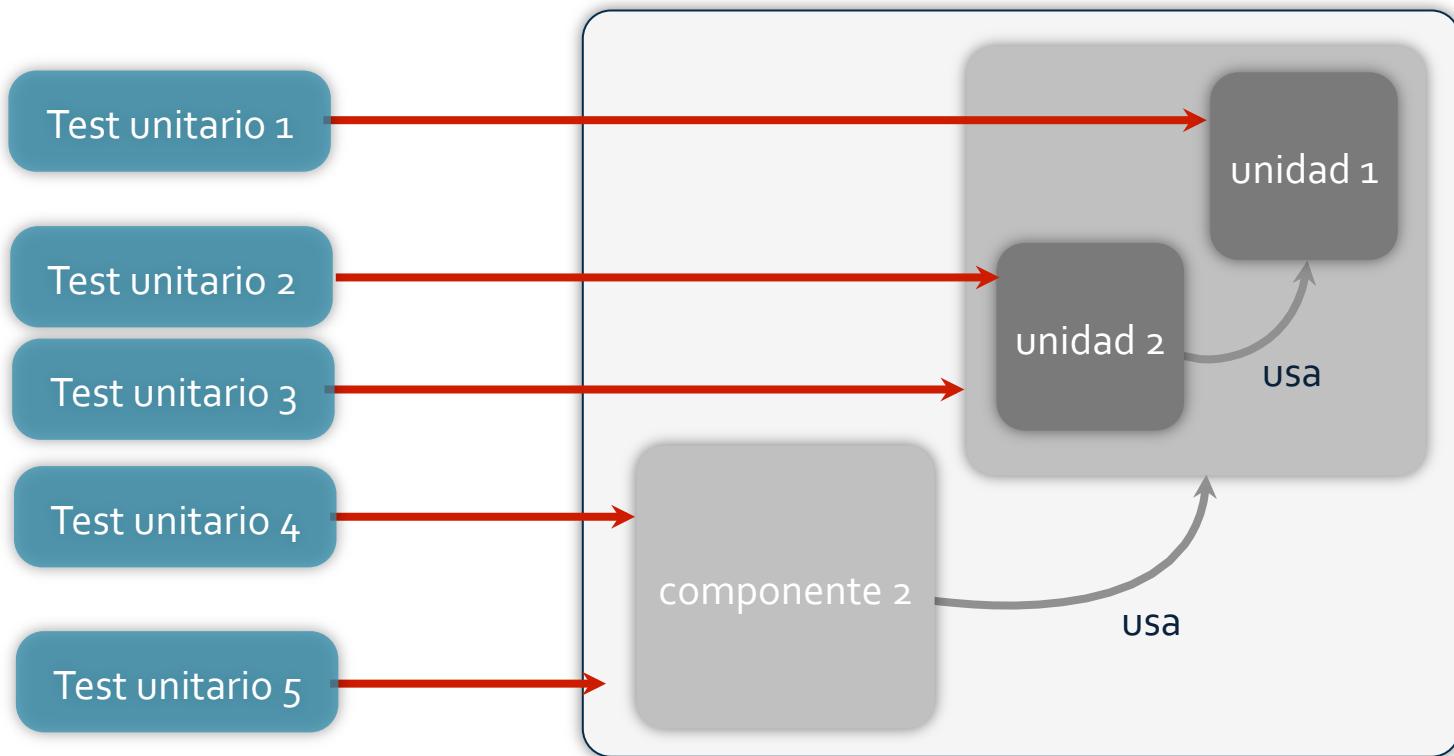
- Algunas herramientas ofrecen **automatizar pruebas que interactuan con el objeto de prueba a través de su interfaz de usuario** (metáfora **record and playback**)
- Estos test acarrean los siguientes **problemas**:
 - Sensibilidad al **comportamiento**
 - Sensibilidad a la **interfaz**
 - Sensibilidad a los **datos**
 - Sensibilidad al **contexto**

Testing unitario

- Es una metodología para probar **unidades individuales** de código (**SUT**: Software under test), preferentemente **de forma aislada** de sus dependencias (**DOC**: Depend-on component).
- Las unidades pueden ser de **distinta granularidad**: porción de código, método, clase, librería, etc.
- Las pruebas son **creadas por los mismos programadores** durante el proceso de desarrollo.
- Se utiliza un **framework** para su sistematización y automatización.

Testing unitario

Normalmente las unidades son las **partes más pequeñas** de un sistema: funciones o procedimientos.



Sistematización del testing unitario

- Una **framework** de apoyo al **testing unitario** ayuda a **sistematizar y automatizar** parte de **las tareas manuales** asociadas al testing:
- Define la **estructura básica** de un test:
 - Inicialización | Ejercitación | Verificación | Demolición
 - Permite **almacenar** los tests como “scripts”.
 - Permite definir la **salida esperada** como parte del script.

Sistematización del testing unitario

- Permite organizar tests en suites:
 - Algunas librerías encuentran todos los test definidos y construyen la suite automáticamente.
 - Otras requieren una construcción explícita.
 - Las suites suelen organizarse jerárquicamente.

Sistematización del testing unitario

- Ofrece entornos para la ejecución de tests y suites:
 - El *test runner* permite ejecutar automáticamente
 - todos los test o un subconjunto de ellos,
 - de manera independiente.
 - El *test runner* permite repetir un test con diferentes datos de entrada.

Sistematización del testing unitario

- Reporta información detallada sobre las pruebas:
 - El *test runner* genera un reporte estadístico de la ejecución de las suites,
 - El resultado puede incluir datos de benchmarking y cobertura.

JUnit

JUnit es una librería de apoyo al testing unitario para Java:

- Define la estructura básica de un test.
- Permite organizar tests en suites.
- Ofrece entornos para la ejecución de tests y suites.
- Reporta información detallada sobre las pruebas,
 - en especial sobre las fallas.

Estructura de una prueba

```
import static org.junit.Assert.*;  
import org.junit.Test;  
  
public class MaxTest {  
    @Test  
    public void maxOnRight() {  
        int a = 1;  
        int b = 3;  
        int res = SampleStaticRoutines.max(a,b);  
        assertTrue(res == 3);  
    }  
}
```

Inicialización: se preparan los datos para alimentar al programa.

Ejecución: se ejecuta el programa con los datos construidos.

Verificación: se evalúa si los resultados obtenidos se corresponden con lo esperado.

Demo

Max

Aserciones

- `assertTrue(<expr>)`: verifica que `<expr>` evalúe a true.
- `assertFalse(<expr>)`: verifica que `<expr>` evalúe a false.
- `assertEquals(<expr1>, <expr2>)`: verifica que `<expr1>` y `<expr2>` evalúen al mismo valor.
- `assertArrayEquals(<array1>, <array2>)`: verifica que `<array1>` y `<array2>` sean iguales, elemento a elemento.
- `assertNotNull(<object>)`: verifica que `<object>` no sea null.
- `assertNull(<object>)`: verifica que `<object>` sea null.
- `assertNotSame(<object1>, <object2>)`: verifica que `<object1>` y `<object2>` no sean el mismo objeto.
- `assertSame(<object1>, <object2>)`: verifica que `<object1>` y `<object2>` sean el mismo objeto.

Aserciones

- `assertThat(<expr>, <matcher>)`: verifica que `<expr>` satisfaga `<matcher>`.
 - `assertThat(x, is(3));`
 - `assertThat(x, is(not(4)));`
 - `assertThat(responseString, either(containsString("color")).or(containsString("colour")));`
 - `assertThat(myList, hasItem("3"));`
- **Matchers:**
 - <http://junit-team.github.com/junit/javadoc/latest/org/junit/Matchers.html>
 - <http://junit-team.github.io/junit/javadoc/latest/org/hamcrest/core/package-summary.html>

Pruebas negativas

- Son aquellos en los que **probamos que el SUT falla** al ejecutar cierta porción de código bajo ciertas entradas.
- Podemos capturar los **tests negativos** con indicando que se espera cierta excepcion:

```
@Test(expected=IOException.class)
public void yourTestMethod() throws Exception {
    throw new IOException();
}
```

Demo

Max genérico

Suites de pruebas

- Cuando el SUT está compuesto por varias clases, o simplemente varios métodos de una clase, resulta claro que debemos organizar los tests.
- Los tests se organizan en test suites:
 - un conjunto de tests.
- En JUnit, todo conjunto de tests definidos en una misma clase/archivo es una test suite.

Ejercicio

- Utilizando la clase *Point*:
 1. Implementá el método *equals*.
 2. Construí una suite de test para la clase, incluyendo al menos un test por método.
 3. ¿Descubriste algún *bug*? Corregilo.

Suites de pruebas y datos compartidos

- Generalmente los tests de una suite **comparten los datos** (fixture) que manipulan.
 - Suele ser útil organizar los tests definiendo **procesos comunes de inicialización** (o arrangement, o setUp) para todos los tests.
 - Se pueden definir **procesos comunes de destrucción** (o tearDown), que se ejecuten luego de cada test.

Datos compartidos: ejemplo

Consideremos un ejemplo de testing de una clase Minefield, que representa el estado de un campo minado en el juego “Buscaminas”:

```
public class Minefield {  
    private Mine[][] field;  
    ...  
    public int minedNeighbours(int x, int y) {  
        ...  
    }  
}  
  
public class Mine {  
    private boolean isMined;  
    private boolean isMarked;  
    private boolean isOpened;  
    ...  
}
```

Para poder testear `minedNeighbours`, debemos crear el `minefield`, y ubicar minas en lugares específicos.

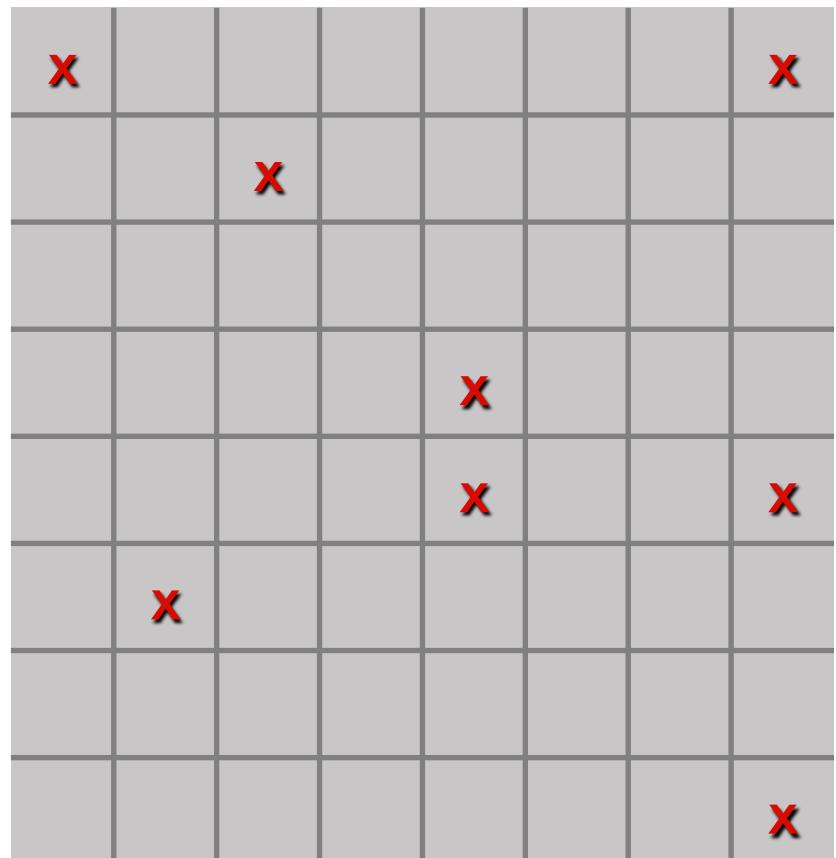
Datos compartidos: ejemplo

- El método `@Before` crea el escenario adecuado para la ejecución de tests de `minedNeighbours`.
- Se **ejecuta antes** de cada test.

```
public class MinefieldTest {  
    private Minefield field;  
  
    @Before  
    public void setUp() {  
        if (field == null) {  
            field = new Minefield();  
        }  
  
        field.putMine(0, 0);  
        field.putMine(3, 4);  
        field.putMine(4, 3);  
        field.putMine(2, 2);  
        field.putMine(0, 7);  
        field.putMine(7, 7);  
        field.putMine(5, 1);  
        field.putMine(4, 7);  
    }  
}
```

Datos compartidos: ejemplo

- El método `@Before` crea el escenario adecuado para la ejecución de tests de `minedNeighbours`.
- Se **ejecuta antes** de cada test.

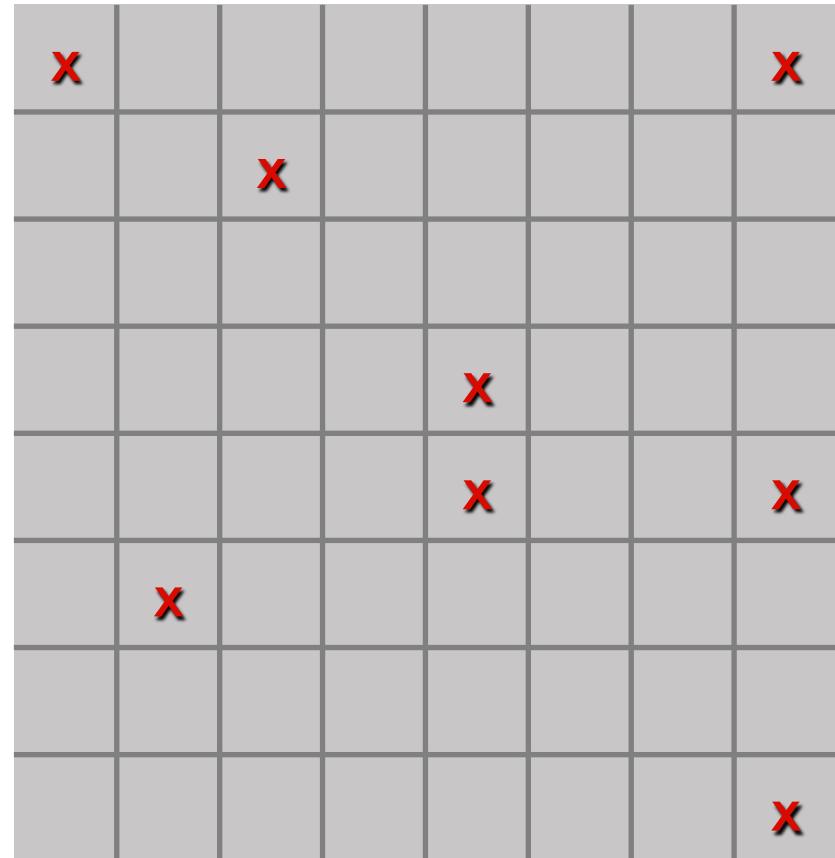


Datos compartidos: ejemplo

```
@Test  
public void minedNeighbours1() {  
    int n;  
    n = field.minedNeighbours(1, 0);  
    assertTrue(n == 1);  
}
```

```
@Test  
public void minedNeighbours2() {  
    int n;  
    n = field.minedNeighbours(7, 7);  
    assertTrue(n == 0);  
}
```

```
@Test  
public void minedNeighbours3() {  
    int n;  
    field.putMine(7, 6);  
    n = field.minedNeighbours(7, 7);  
    assertTrue(n == 1);  
}
```



Datos compartidos: ejemplo

- El método `@After` limpia el escenario creado para la ejecución de tests de `minedNeighbours`.
- Se ejecuta después de cada test.

```
public class MinefieldTest {  
    private Minefield field;  
  
    @After  
    public void tearDown() {  
        for (int i=0; i<8; i++) {  
            for (int j=0; j<8; j++) {  
                field.removeMine(i, j);  
                field.unmark(i, j);  
                field.close(i, j);  
            }  
        }  
    }  
}
```

@Before y @After

- Es importante que **no haya dependencias** entre tests diferentes.
 - No hay **ninguna garantía sobre el orden** en el que se ejecutan los tests
- Las rutinas **@Before** y **@After** ayudan a **configurar el fixture** en los diferentes tests.

```
MinefieldTest.setUp();
MinefieldTest.minedNeighbours3;
MinefieldTest.tearDown();
MinefieldTest.setUp();
MinefieldTest.minedNeighbours1;
MinefieldTest.tearDown();...
```

Fixture e independencia

La importancia del setUp (y tearDown) para mantener la independencia entre tests se ve en el siguiente ejemplo:

```
public class MineTest {  
    private Mine mine;  
  
    @Test  
    public void test1() {  
        mine = new Mine();  
        mine.setMarked(true);  
        assertTrue(!mine.isOpened());  
    }  
  
    @Test  
    public void test2() {  
        mine.setOpened(true);  
        assertTrue(!mine.isMarked());  
    }  
}
```

Fixture e independencia

La importancia del setUp (y tearDown) para mantener la independencia entre tests se ve en el siguiente ejemplo:

```
public class MineTest {  
    private Mine mine;  
  
    @Before  
    public void setUp() {  
        mine = new Mine();  
    }  
  
    @Test  
    public void test1() {  
        mine.setMarked(true);  
        assertTrue(mine.isMarked() &&  
                   !mine.isOpened());  
    }  
  
    @Test  
    public void test2() {  
        mine.setOpened(true);  
        assertTrue(!mine.isMarked() &&  
                   mine.isOpened());  
    }  
}
```

Ejercicio

- Utilizando las clases *Minefield* y *MinefieldTest*:
 1. Agregá a *MinefieldTest* más casos de prueba para testear todos los métodos de *Minefield*.
 2. Implementá el método *minedNeighbours* en la clase *Minefield* y extendé el test correspondiente.

Test paramétricos

- Varios tests pueden poseer exactamente la misma estructura, pero difieren en los datos que se utilizan para la *inicialización* del test
 - JUnit ofrece una forma de organizar los tests, separando su estructura de los datos que manipulan.
- La claves son:
 - definir un test como paramétrico.
 - definir un generador de parámetros para la suite, que se usará para instanciar los datos para los tests.

Test paramétricos: ejemplo

```
@RunWith(Parameterized.class)
public class LargestTest {

    private Integer [] array;
    private Integer res;

    public LargestTest(Object [] array, Object res) {
        this.array = (Integer[]) array;
        this.res = (Integer) res;
    }

    @Parameters
    public static Collection<Object[]> firstValues() {
        return Arrays.asList(new Object[][] {
            {new Integer [] { 1,2,3 }, 3 },
            {new Integer [] { 2,1,3 }, 3 },
            {new Integer [] { 3,1,2 }, 3 },});
    }

    @Test
    public void testFirst() {
        int max = SampleStaticRoutines.largest((Integer[]) array);
        assertTrue(res == max);
    }
}
```

Indica que la suite está formada por tests paramétricos.

Productor de parámetros

Test genérico

Ejercicio

- Considerando los métodos definidos en *staticRoutines*:
 1. Extendé el test de *largest* para incluir más casos.
 2. Construí una suite paramétrica para testear *bubbleSort*.
 3. Si encontrás *bugs*, corregilos.

Beneficios

- El testing unitario **sistematizado**:
 - **Facilita los cambios**: los test permiten comprobar que la unidad continúa funcionando correctamente a pesar de cualquier tipo de cambio (refactoring o nuevas funcionalidades); su sistematización facilita el testing de regresión.
 - **Simplifica la integración**: los test reducen la incertidumbre sobre las componentes individuales, facilitando la integración y su verificación, ya sea *bottom-up* o *top-down* (con la integración de *mocks*)

Beneficios

- El testing unitario **sistematizado**:
 - Sirve como documentación: los test especifican las partes criticas del comportamiento de la unidad, ya sea su uso apropiado o inapropiado; sirven como ejemplo y descripción siempre actualizada de la API.
 - Contribuye al diseño: los test sirven como ejemplo de uso y dan *feedback* temprano; fuerzan a enfocarse en el **qué** y en el **cómo**, encontrando defectos de diseño y aportando a su replanteamiento.

Introducción al testing unitario

