Ministerio de Industria
Presidencia de la Nación

INTI

Automatización de pruebas unitarias y de integración en el desarrollo de Software

Automatización de pruebas unitarias y de integración en el desarrollo de Software

Disertante:
Ing. Federico S. Bobbio
Investigador
Laboratorio de Software
Centro INTI Córdoba
fbobbio@inti.gob.ar

1. Introducción
2. Calidad
3. Testing unitario
4. Dobles de pruebas
5. Criterios de cobertura
6. Integración continua

Introducción

Laboratorio de Software

Centro INTI Córdoba

Introducción y Calidad

Verificación y Validación

- **Verificación**: el software debería realizar lo que su especificación indica: ¿Construimos el producto correctamente?
- **Validación**: el software debería hacer lo que el usuario requiere de él: ¿Construimos el producto correcto?

El Proceso de V y V

Dos objetivos centrales:

- Descubrir defectos
- Medir si el sistema es "usable" en una situación de operación del mismo

Análisis

Dos maneras:

- Análisis dinámico
- Análisis estático
  - Inspección manual

¿Qué es Testing?

- No es vs. Si es
- Definiciones
- Misión del Testing

No es vs. Si es

- No es:
  - Depurar código
  - Verificar que las funciones del software se implementan
  - Demostrar que no hay defectos
- Si es:
  - Proceso destructivo
  - Encontrar defectos

Algunas definiciones

- Glenford Myers
- IEEE 829-1998
- ANSI – 1990 – Std 610.12
- Rick Craig y Stefan Jaskiel

Misión del Testing

- Generar información
- Forma de medir la calidad

Principios del Testing

- No es exhaustivo
- Testing temprano
- El Testing debe validar al cliente
- Contexto

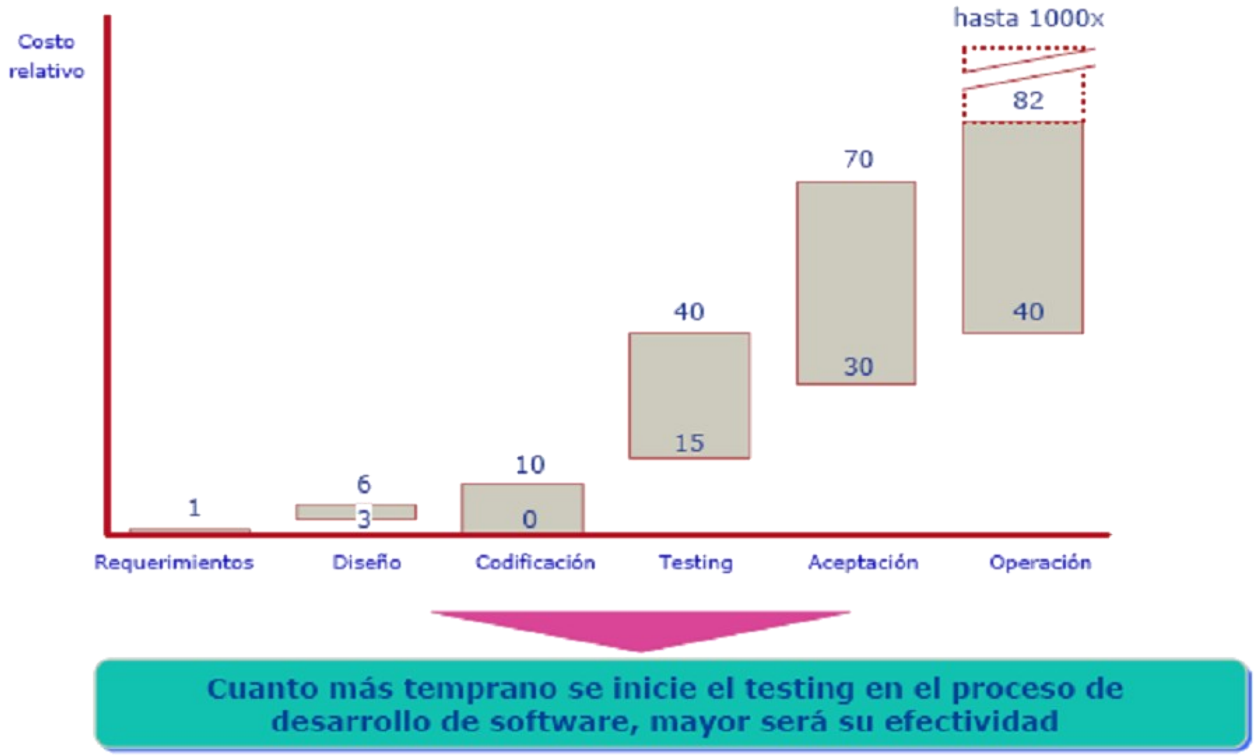
Testing e inspecciones

- Ambas son complementarias entre si
- Inspecciones son más efectivas en Verificación
- El testing es parte explícita del proceso de desarrollo

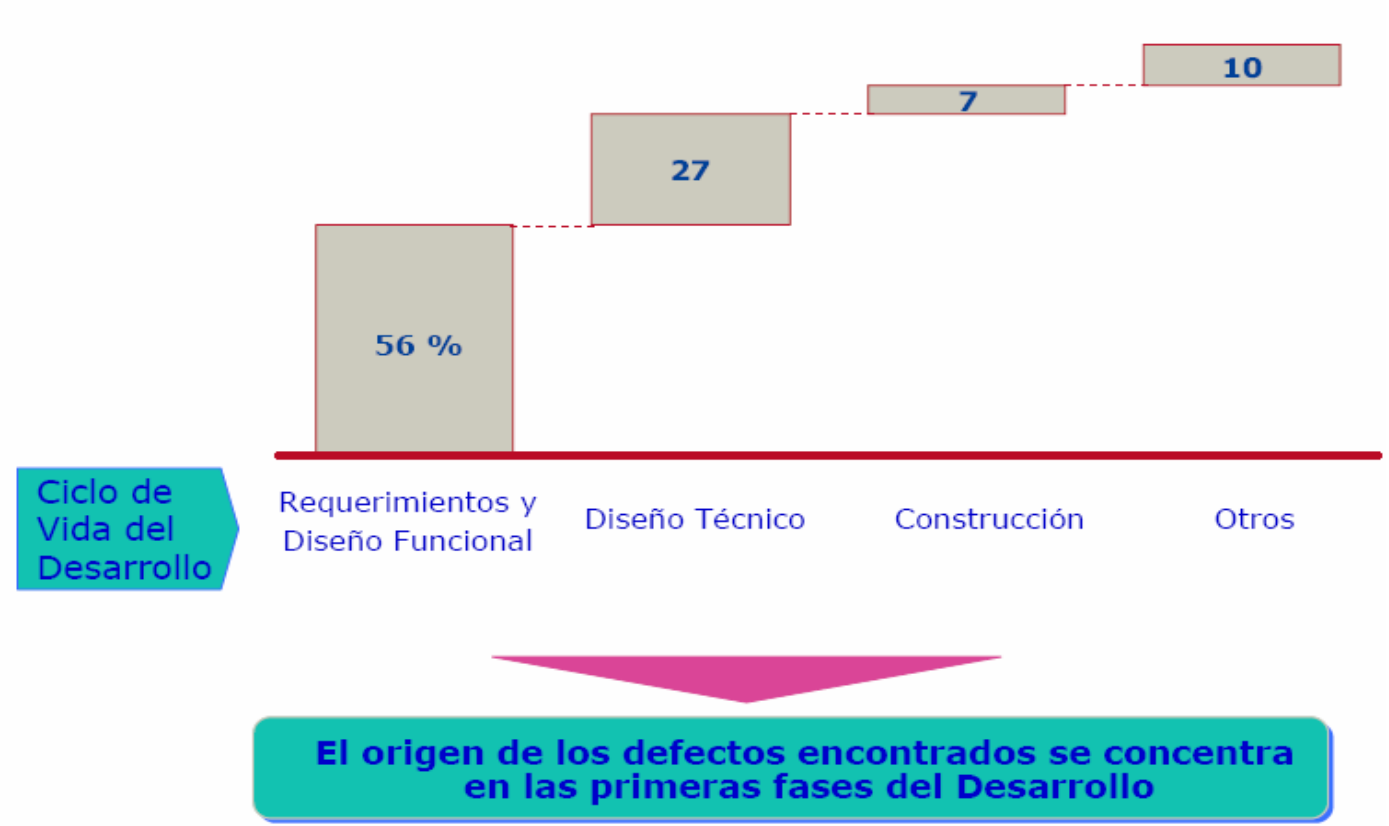
Los procesos en la historia

- Desarrollo en cascada
- Desarrollo en V
- RUP
- Ágiles
- TDD

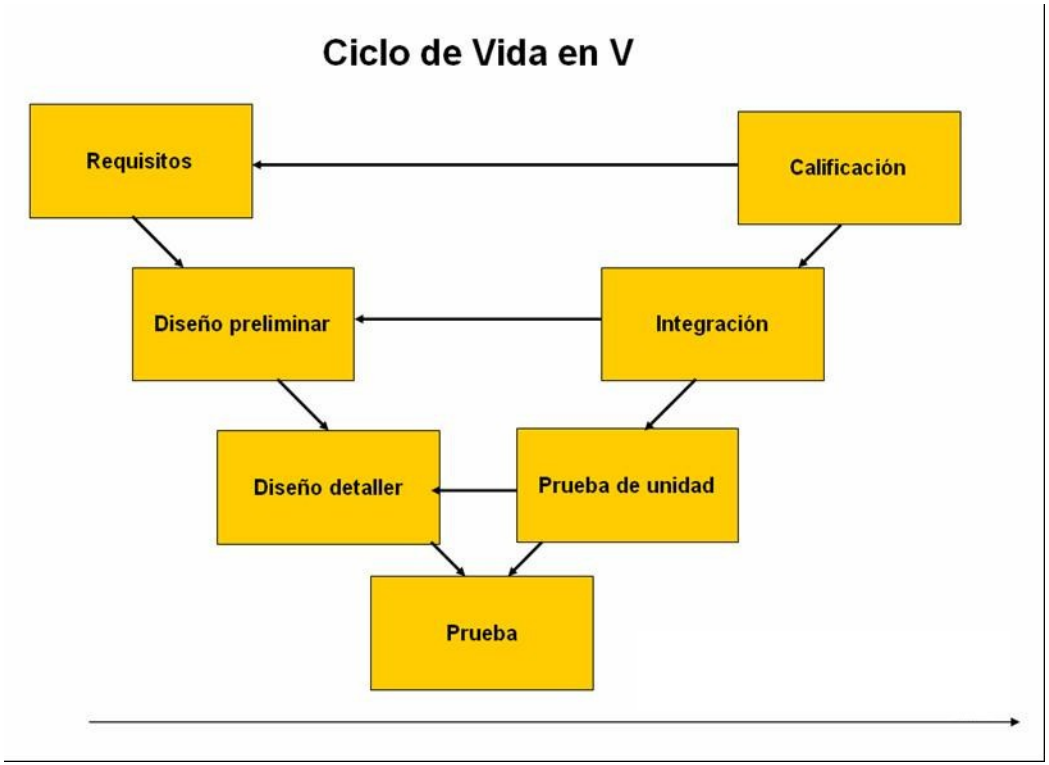
Desarrollo en cascada



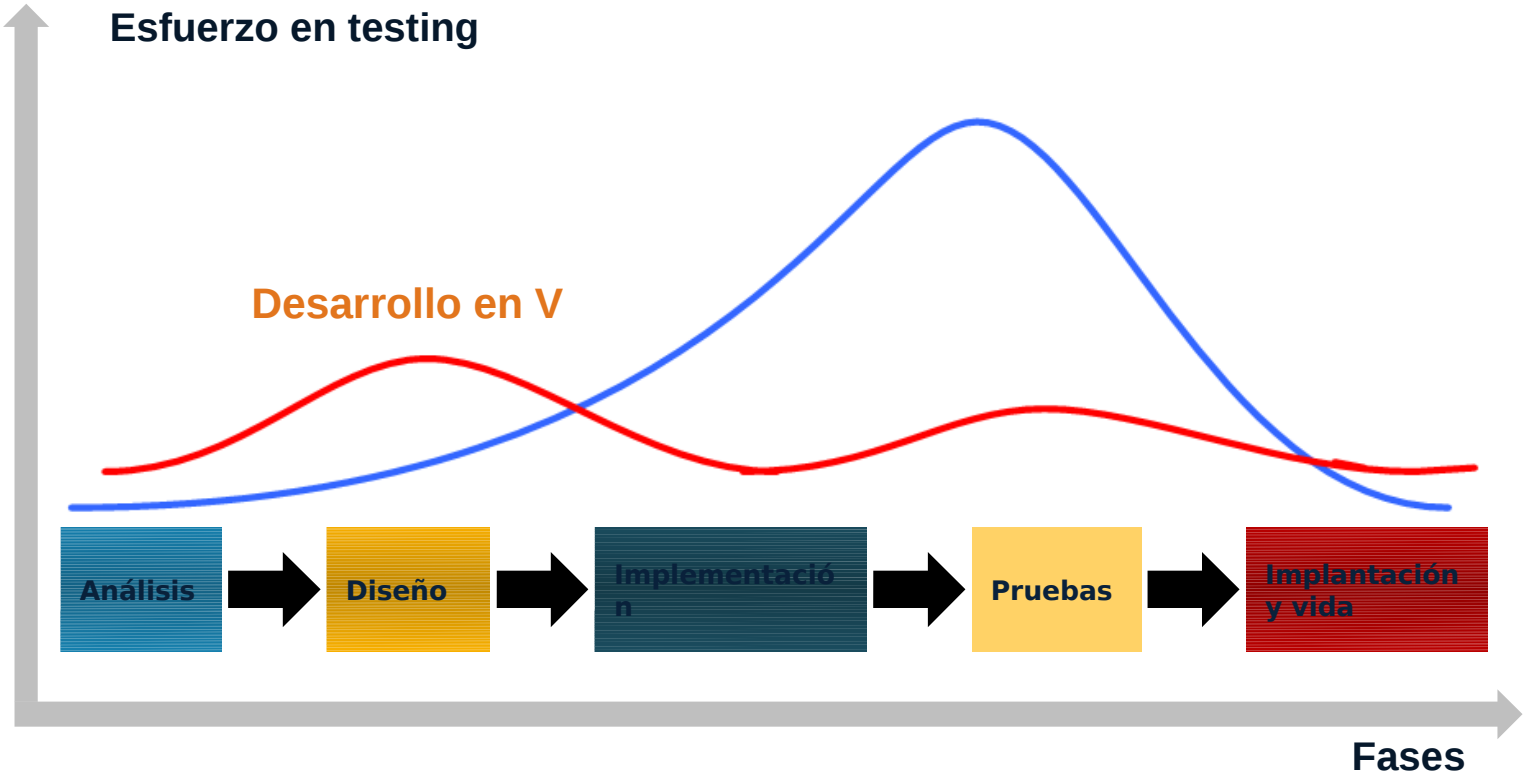
Desarrollo en cascada



Desarrollo en V

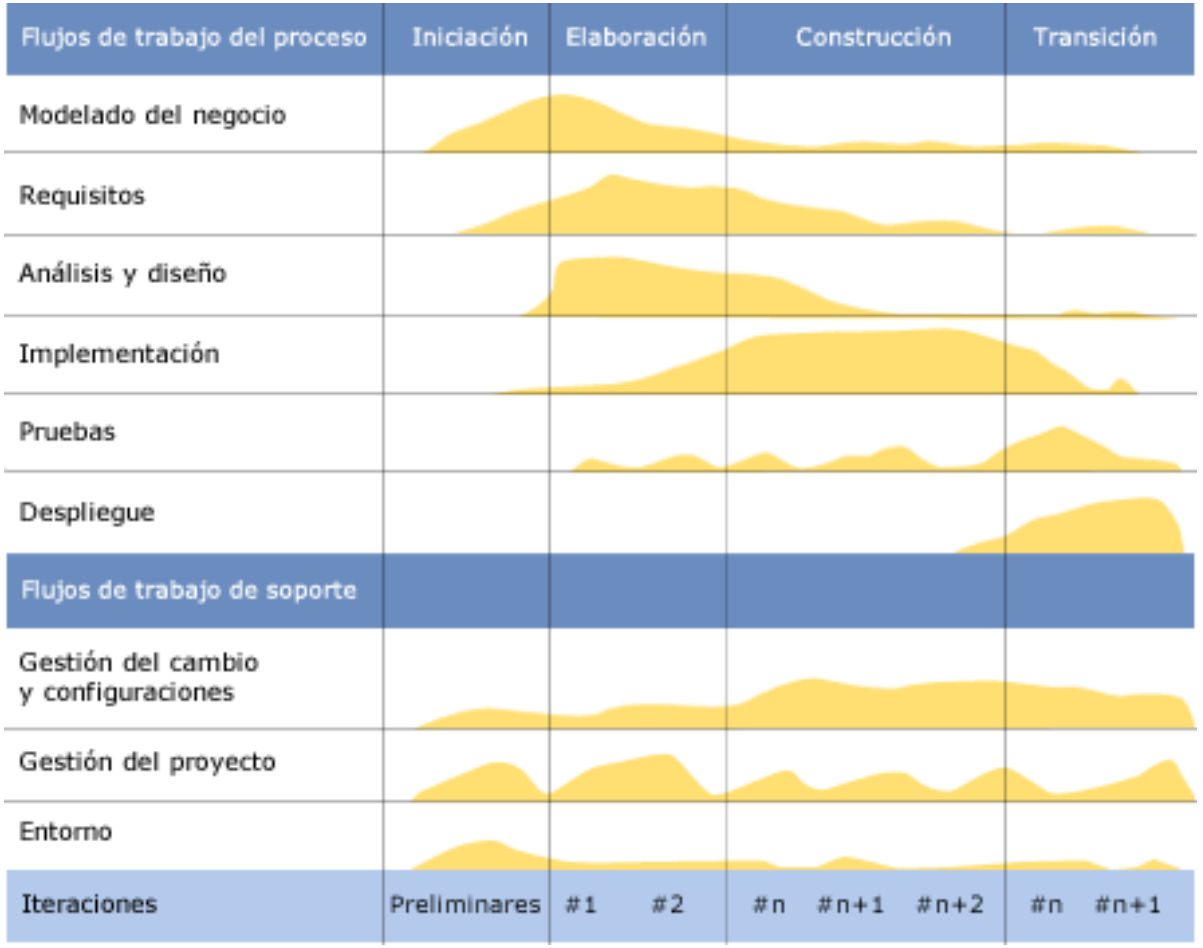


Desarrollo en V



Introducción y Calidad > ¿Qué es Testing? > Procesos

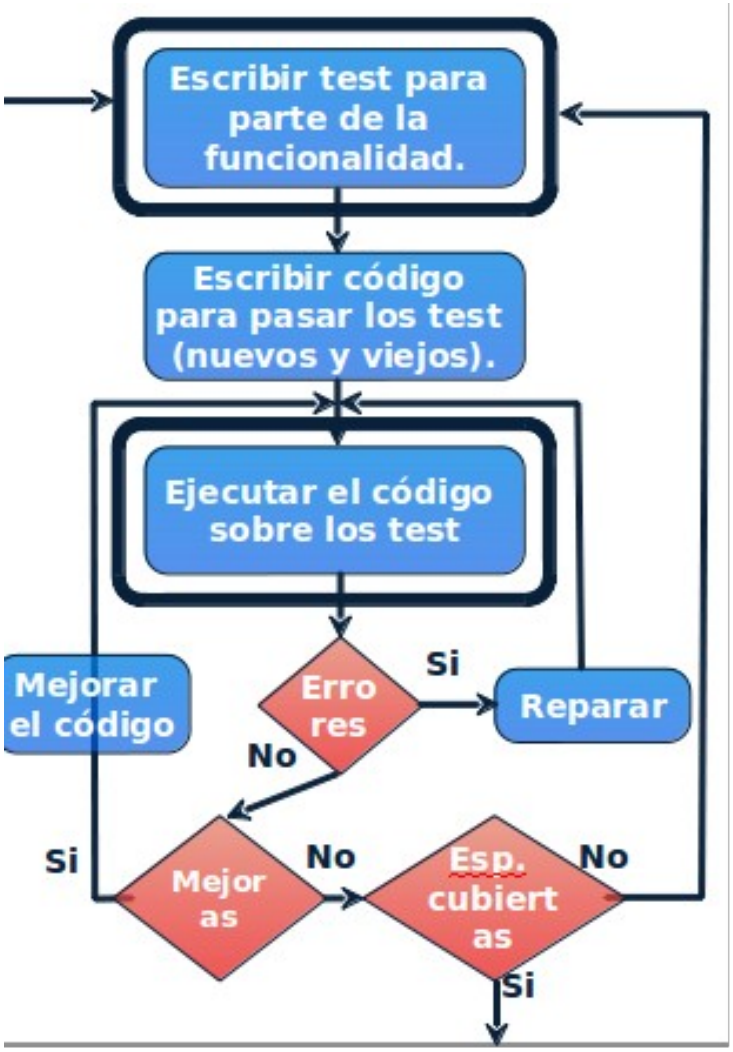
RUP



Ágiles


Escribir código para implementar alguna funcionalidad
Mejorar los test para probar la nueva funcionalidad.
Ejecutar el test
Error es
Si
Reparar
No
No
Esp. cubiert as
Si

TDD


Escribir test para parte de la funcionalidad.
Escribir código para pasar los test (nuevos y viejos).
Ejecutar el código sobre los test
Erro res
Si
Reparar
Mejorar el código
No
Si
Mejor as
No
Esp. cubiert as
No
Si

Terminología básica

- Error
- Defecto
- Falla
- Ejemplo

Ejemplo

```
int doblar(int param)
{
    int res;
    res = param * param;
    return(res);
}
```

Niveles de prueba

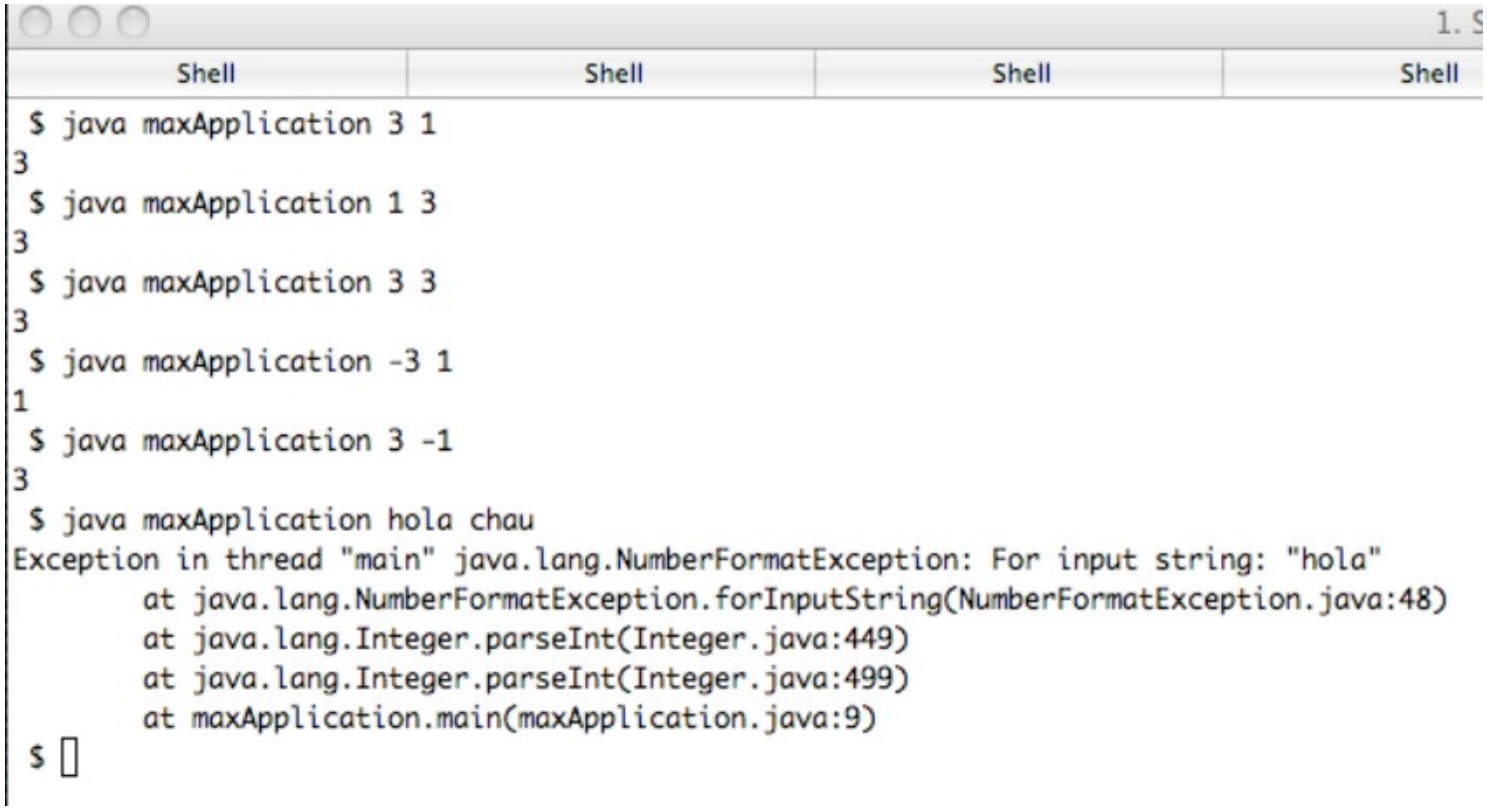
- De componente o unitarios
- De integración
- De sistema
- De aceptación

Testing unitario

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



Dificultades

- No almacena los tests
- Requiere del desarrollo de “arneses”
- Requiere inspección humana en la salida de los tests

Sistematización

Las librerías de apoyo al testing sistematizan las tareas manuales:

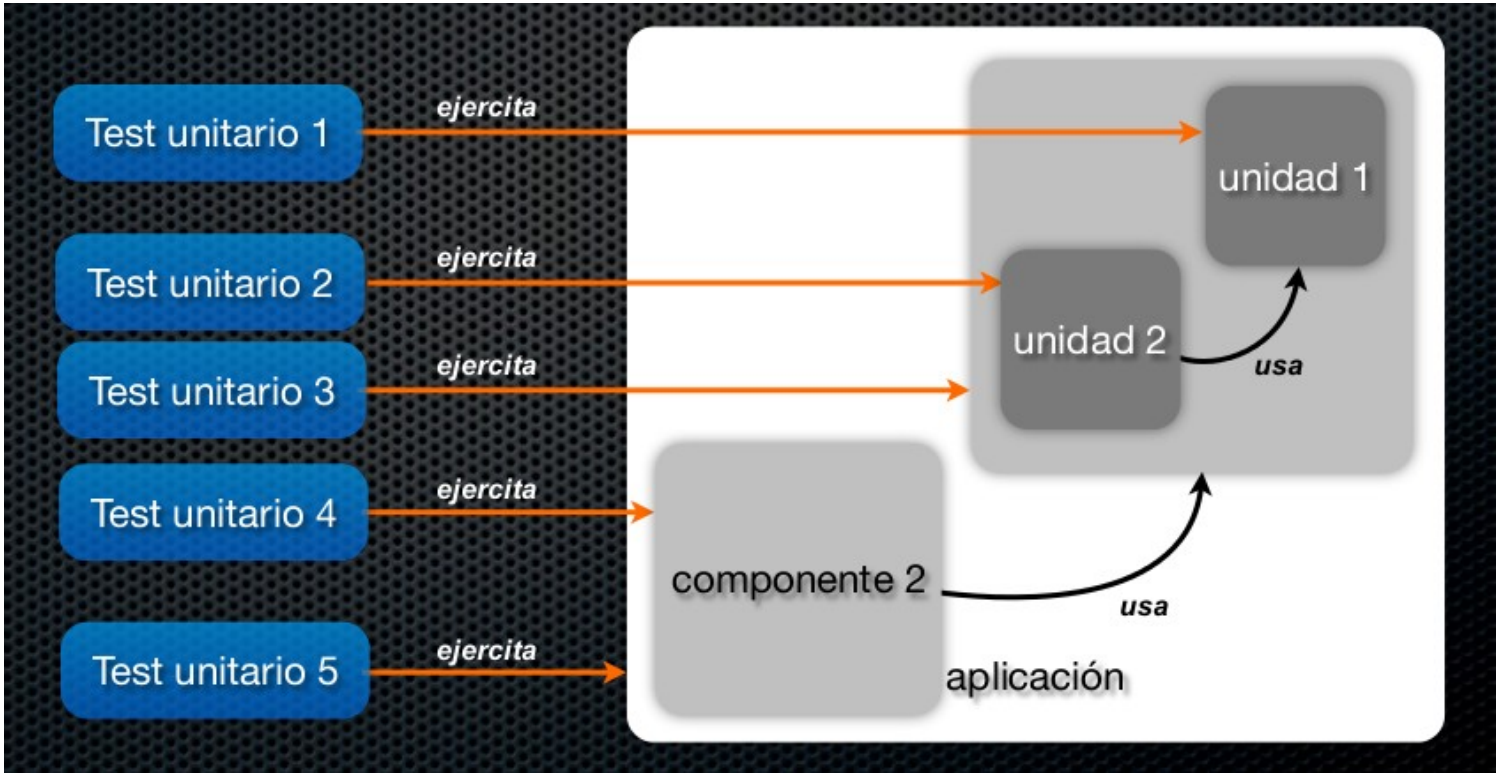
- Almacenar tests como "scripts"
- Esquema estándar para scripts de tests
- Definen la salida esperada como parte del script
- Construyen automáticamente arneses para la ejecución de los tests.

Testing unitario

El uso de esta metodología:

- Facilita los cambios
- Simplifica la integración de componentes
- Sirve como documentación de código
- Contribuye al diseño del sistema
- Ofrece una granularidad variable

Testing unitario

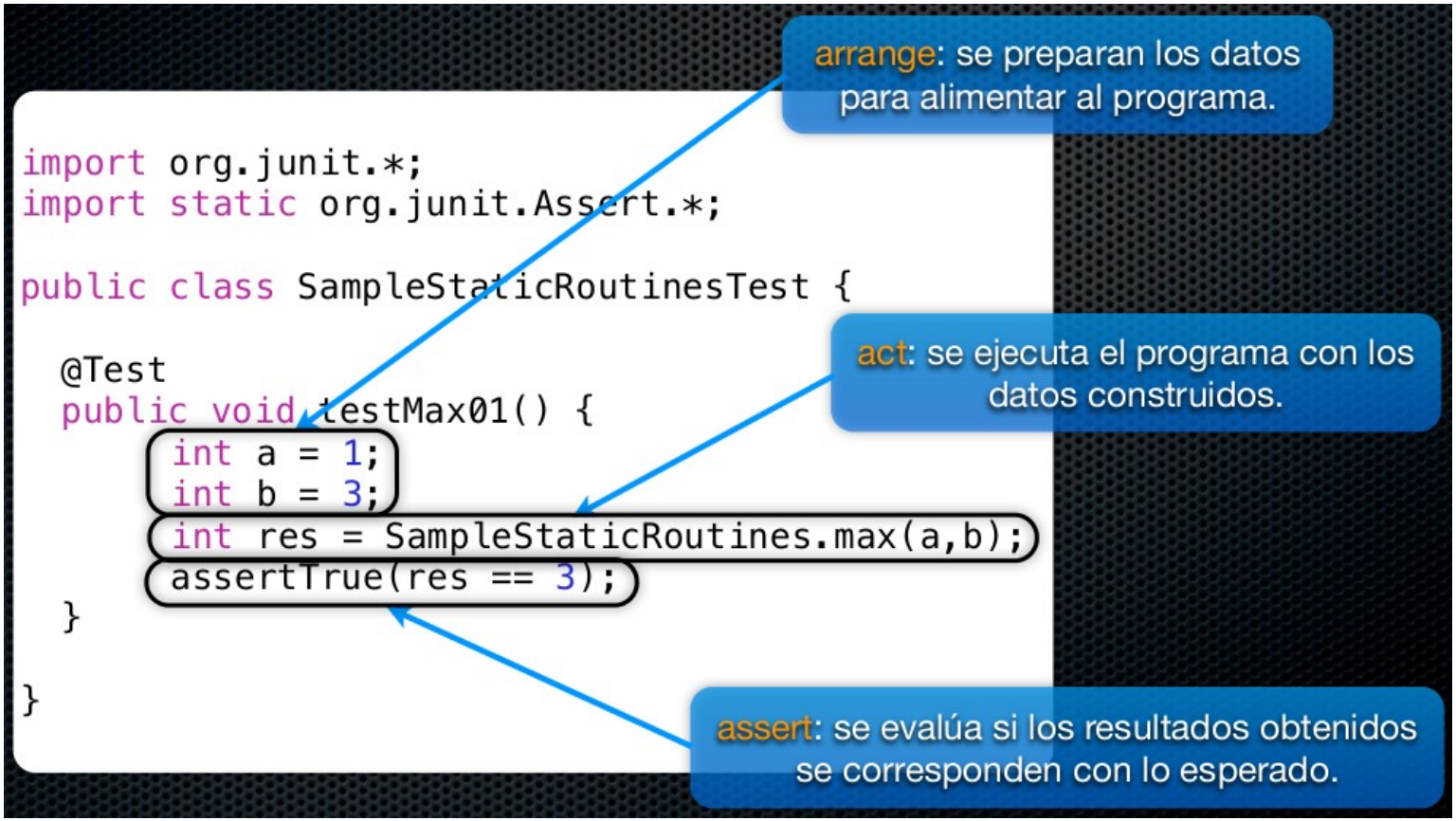


Testing unitario con JUnit

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


arrange: se preparan los datos para alimentar al programa.
act: se ejecuta el programa con los datos construidos.
assert: se evalúa si los resultados obtenidos se corresponden con lo esperado.
import org.junit.*;
import static org.junit.Assert.*;
public class SampleStaticRoutinesTest {
@Test
public void testMax01() {
int a = 1;
int b = 3;
int res = SampleStaticRoutines.max(a,b);
assertTrue(res == 3);
}
}

Aserciones

Funciones propias ofrecidas por el framework de test unitario que permiten:

- Evaluar y contrastar los datos generados en el test contra los resultados esperados
- Registrar la información de esa evaluación

Test negativos

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

   throw new IOException();
}
```

Testing unitario con PHPUNit

PHPUnit es una librería de apoyo al testing unitario para PHP que trabaja de manera similar a Junit pero marcando diferencias en:

- Particularidades del lenguaje PHP
- Paradigmas que rigen a PHP
- Entornos en los que se maneja el código

Estructura

Example 2.1. Testing array operations with PHPUnit

```php
<?php
class StackTest extends PHPUnit_Framework_TestCase
{
    public function testPushAndPop()
    {
        $stack = array();
        $this->assertEquals(0, count($stack));

        array_push($stack, 'foo');
        $this->assertEquals('foo', $stack[count($stack)-1]);
        $this->assertEquals(1, count($stack));

        $this->assertEquals('foo', array_pop($stack));
        $this->assertEquals(0, count($stack));
    }
}
?>
```

Suites de test

Los tests se organizan en test suites:

- Una test suite es simplemente un conjunto de tests
- En JUnit, todo conjunto de tests definidos en una misma clase/archivo es una test suite
- En PHPUnit un conjunto de tests puede definirse dentro de una suite desde el sistema de archivos o desde una archivo XML de configuración

Suites de test y datos compartidos

En muchos casos, los tests de una suite comparten los datos que manipulan:

- SetUp: procesos comunes de fixture para todos los tests
- TearDown: procesos comunes de destrucción que se ejecutan luego de cada test

Ejemplo: MineField

```
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;

    ...
}
```

```
public class MinefieldTest {

   private Minefield testingField;

   @Before
   public void setUp() throws Exception {
       if (testingField == null) {
           testingField = new Minefield();
       }
       for (int i=0; i<8; i++) {
           for (int j=0; j<8; j++) {
               testingField.removeMine(i, j);
               testingField.unmark(i, j);
               testingField.close(i, j);
           }
       }
       testingField.putMine(0, 0);
       testingField.putMine(3, 4);
       testingField.putMine(4, 3);
       testingField.putMine(2, 2);
       testingField.putMine(0, 7);
       testingField.putMine(7, 7);
       testingField.putMine(5, 1);
       testingField.putMine(4, 7);
   }

    ...
}
```

Ejemplo en PHP

Example 4.2. Example showing all template methods available

```php
<?php
class TemplateMethodsTest extends PHPUnit_Framework_TestCase
{
    public static function setUpBeforeClass()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    protected function setUp()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    protected function tearDown()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    public static function tearDownAfterClass()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
    }

    public function testTwo()
    {
        fwrite(STDOUT, __METHOD__ . "\n");
        $this->assertTrue(FALSE);
    }
```

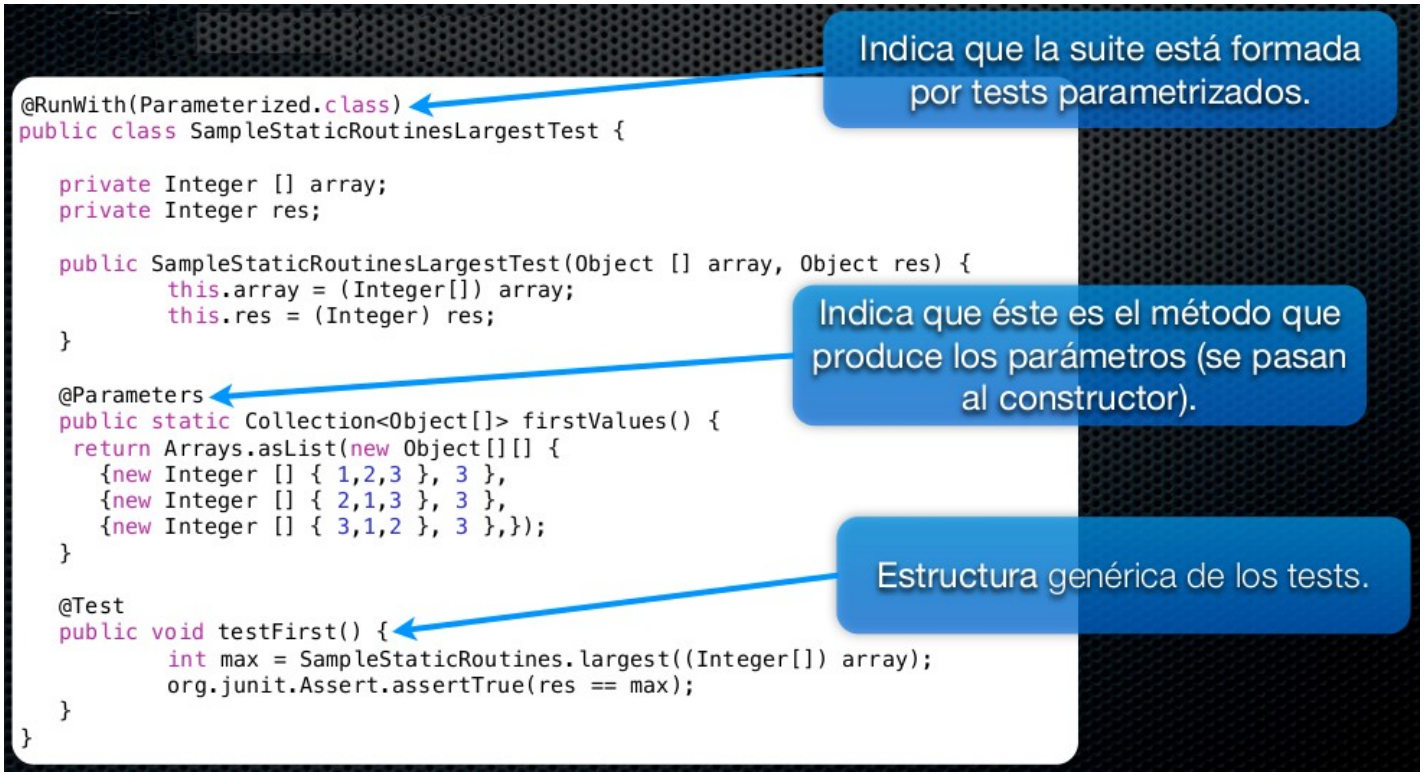
Sobre setUp y tearDown

Ya que no hay ninguna garantía sobre el orden en el que se ejecutan los tests , las rutinas setUp y tearDown ayudan a garantizar la ausencia de dependencias entre diferentes tests

Sobre setUp y tearDown

- JAVA:
  - @Before: métodos que se ejecutan antes de cada test
  - @After: métodos que se ejecutan después de cada test
- PHP: métodos heradados del template
  - setUp(), setUpBeforeClass(), assertPreConditions()
  - tearDown(), tearDownBeforeClass(), AssertPostConditions()

Test parametrizados por datos




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

   private Integer [] array;
   private Integer res;

   public SampleStaticRoutinesLargestTest(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);
           org.junit.Assert.assertTrue(res == max);
   }
}
```

Test parametrizados por datos

Example 2.5: Using a data provider that returns an array of arrays

```php
<?php
class DataTest extends PHPUnit_Framework_TestCase
{
    /**
     * @dataProvider provider
     */
    public function testAdd($a, $b, $c)
    {
        $this->assertEquals($c, $a + $b);
    }

    public function provider()
    {
        return array(
          array(0, 0, 0),
          array(0, 1, 1),
          array(1, 0, 1),
          array(1, 1, 3)
        );
    }
}
?>
```

Dobles de prueba

Dobles de prueba

Normalmente el funcionamiento del SUT depende de otros componentes, por dos vías:

- **Entrada indirecta**: es un valor obtenido por invocaciones a un método de un DOC.
- **Salida indirecta**: es una potencial modificación al estado de un DOC.

Dobles de prueba

Un doble de prueba reemplaza un DOC, aislando el SUT cuando:

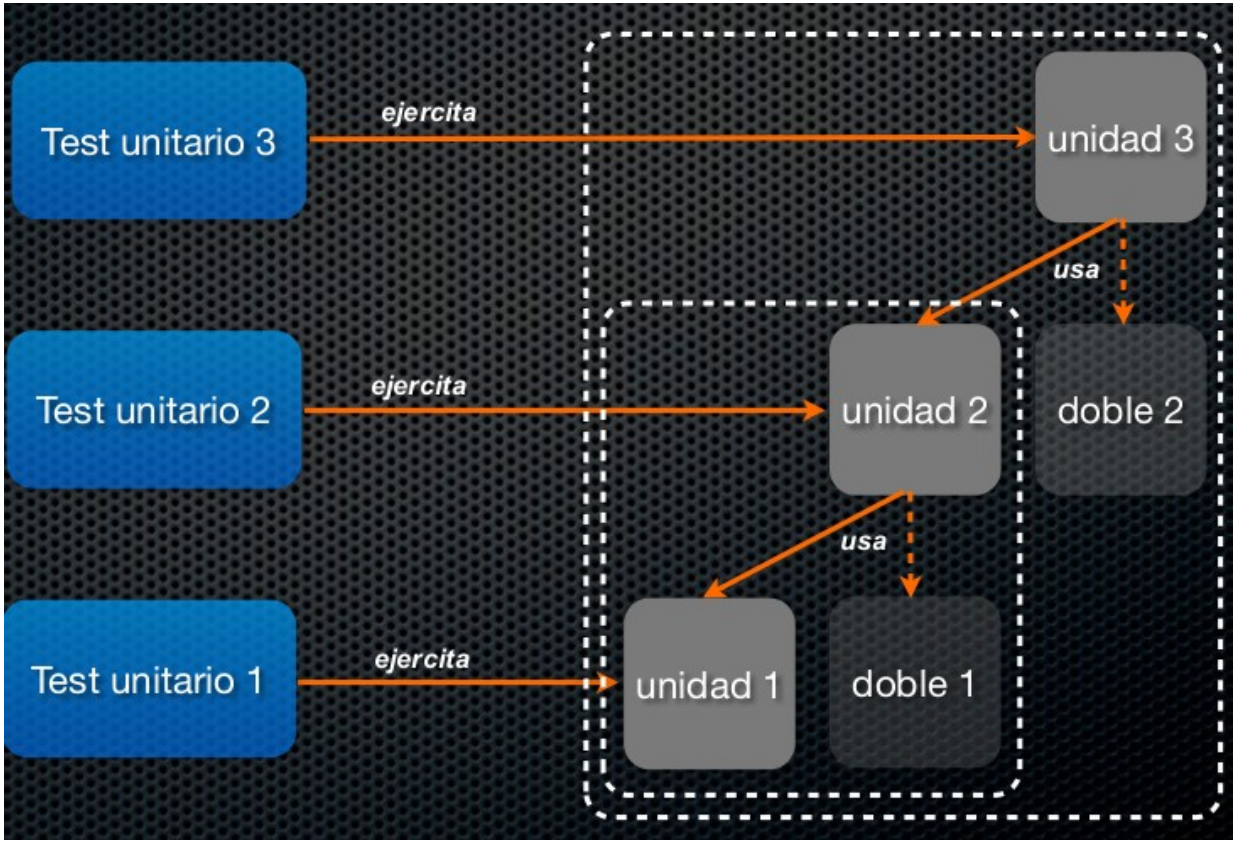
- Es necesario controlar las entradas indirectas, para manejar el hilo de ejecución que se desea ejercitar.
- Es necesario monitorear las salidas indirectas, que son consecuencia del funcionamiento del SUT.

Estrategias de integración

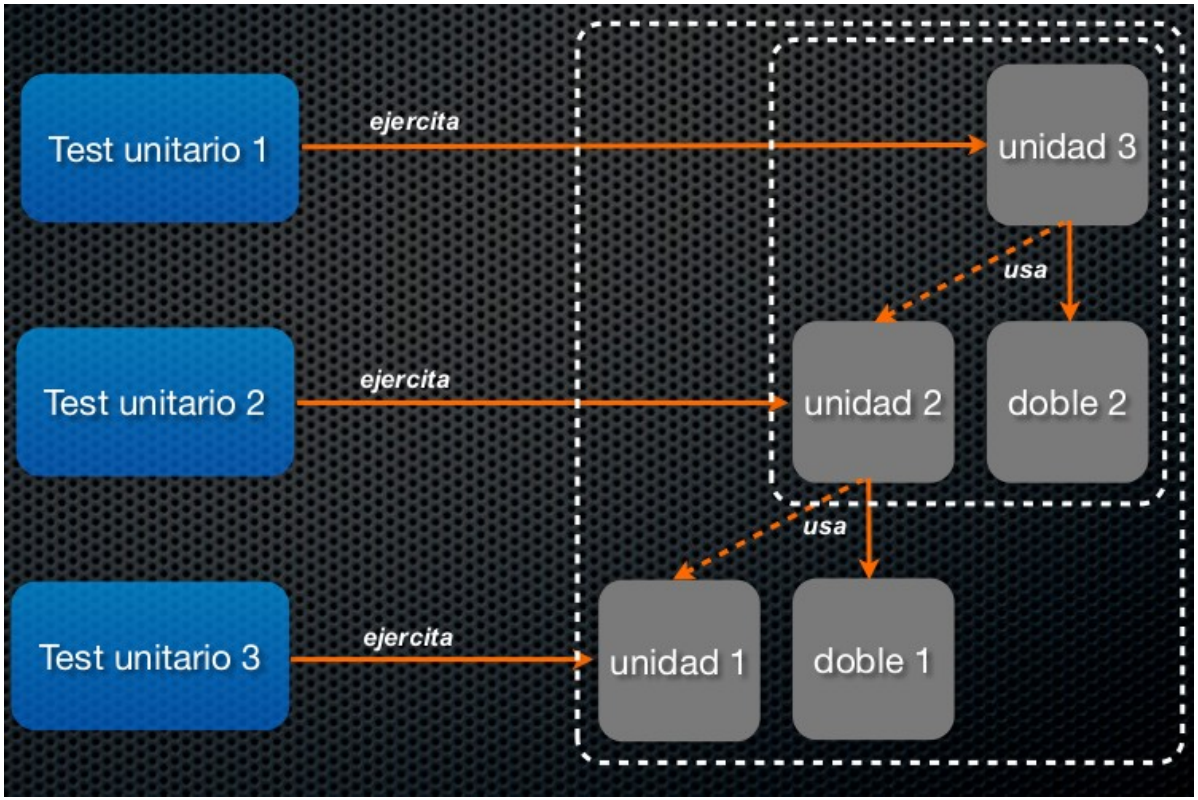
Se identifican dos estrategias distintas de integración:

- Adentro hacia afuera
- Afuera hacia adentro

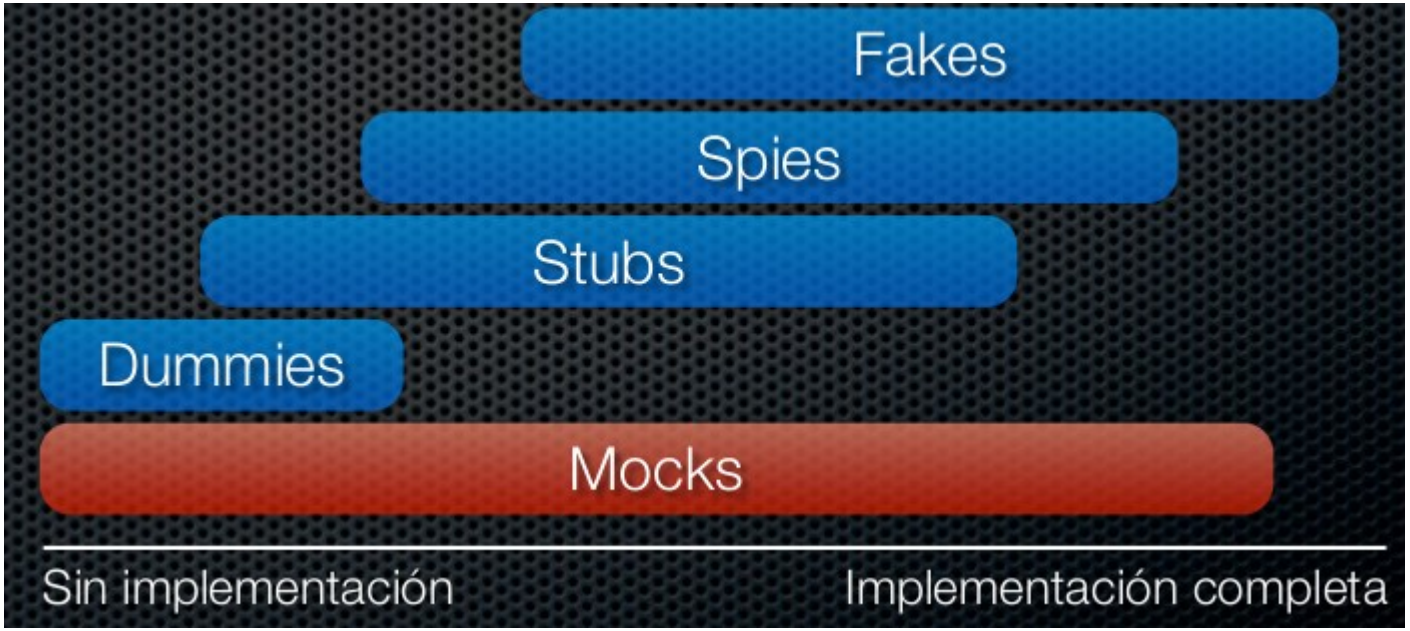
Adentro hacia afuera



Afuera hacia adentro



Tipos de dobles



Dummy

- Es el más simple y primitivo
- Son derivaciones de interfaces que no contienen ninguna implementación
- Se utilizan normalmente como valores para parámetros que nunca se utilizan

Ejemplo

Sólo satisface dependencias formales y es suficiente siempre que la interfaz no sea ejercitada:

```
public class DummyShopDataAccess implements IShopDataAccess {

   public double getProductPrice(int productId) {
        throw new Exception("The method or operation is not implemented.");
   }

   public void save(int orderId, Order o) {
        throw new Exception("The method or operation is not implemented.");
   }

}
```

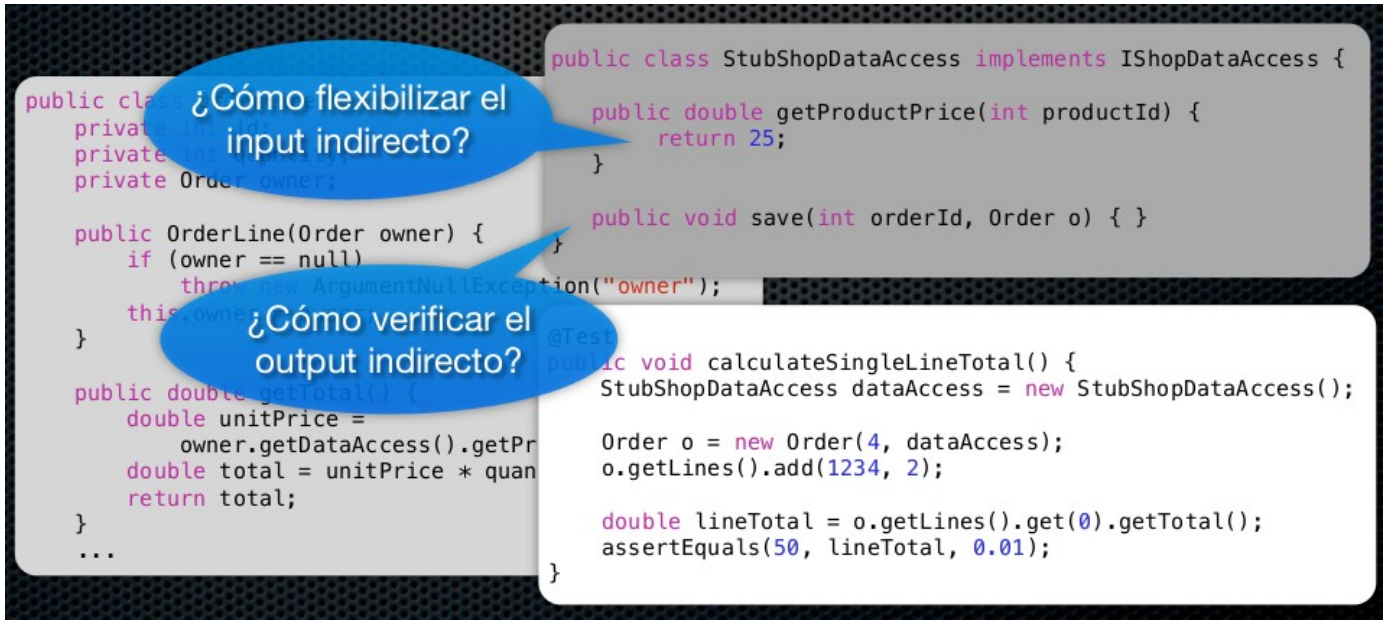
```
@Test
public void createOrder() {
    DummyShopDataAccess dataAccess = new DummyShopDataAccess();

    Order o = new Order(2, dataAccess);
    o.getLines().add(1234, 1);
    o.getLines().add(4321, 3);

    assertEquals(2, o.getLines().size());
}
```

Stub

Este caso extiende al Dummy ya que interpreta que la interfaz es ejercitada, permitiendo la ejecución de sus métodos

Ejemplo

Funciona en casos de input y output indirecto:



Spy

- Utilizado para verificar el output indirecto de algún método que deba ser ejecutado desde la unidad a probar
- Nos indicará si al realizar ciertos pasos se alcanzó la ejecución o no de dicho output indirecto

Ejemplo

- Verificar el output indirecto requiere registrar las invocaciones y sus parámetros:

```
public class SpyShopDataAccess implements IShopDataAccess {
    private boolean saveWasInvoked;

   ...

    public boolean getSaveWasInvoked() {
       return this.saveWasInvoked;

}
```

Fake

- Utilizado para flexibilizar el input indirecto de algún método que inyecte valores al código a probar
- Implica crear una implementación "falsa" de lo que debería hacer el método del cual depende nuestro código bajo prueba

Ejemplo

- Flexibilizar el input indirecto implica aproximarse a una implementación de producción:

```
public class FakeShopDataAccess implements IShopDataAccess {
    private ProductCollection products;

    public FakeShopDataAccess() {
        this.products = new ProductCollection();
    }

    public double getProductPrice(int productId) {
        if (this.products.contains(productId)) {
            return this.products.get(productId).getUnitPrice();
        }
        throw new ArgumentOutOfRangeException("productId");
    }

    List<Product> getProducts() {
        return this.products;
    }

    public void save() {
        ...
    }
    ...
```

```
@Test
public void calculateLineTotalsUsingFake() {
    FakeShopDataAccess dataAccess = new FakeShopDataAccess();
    dataAccess.getProducts().add(new Product(1234, 45));
    dataAccess.getProducts().add(new Product(2345, 15));

    Order o = new Order(6, dataAccess);
    o.getLines().add(1234, 3);
    o.getLines().add(2345, 2);

    assertEquals(135, o.getLines().get(0).getTotal(), 0.01);
    assertEquals(30, o.getLines().get(1).getTotal(), 0.01);
}
```

Mock

Mock es una denominación general para dobles que controlan entrada y salida indirectas.
Los mocks se crean en tiempo de ejecución con la ayuda de una librería específica, verificada y validada, que permite:

- Especificar el comportamiento esperado
- Crear el objeto cuyos métodos serán invocados
- Verificar el comportamiento ejercitado respecto al especificado

EasyMock

Una de las librerías específicas para creación "on the fly" the mocks es la conocida como EasyMock escrita en Java:

```
@Test
public void saveOrderAndVerifyExpectations() {
    IShopDataAccess dataAccess = createMock(IShopDataAccess.class);

    Order o = new Order(6, dataAccess);
    o.getLines().add(1234, 1);
    o.getLines().add(4321, 3);

    // Record expectations
    dataAccess.save(6, o);
    replay(dataAccess);

    o.save();

    verify(dataAccess);
}
```

Criterios de cobertura

Criterios de cobertura

- **Regularidad**: Un criterio es regular si todos los conjuntos de casos test que satisfacen el criterio detectan en general los mismos errores

- **Validez**: Un criterio es válido si para cualquier error en el programa hay un conjunto de casos de test que satisfagan tal criterio y detecten el error

Técnicas

Existen principalmente dos ramas para definir criterios de testing:

- **Caja negra** (funcional)
- **Caja blanca** (estructural)

Técnicas de caja negra

- Caso de uso
- Particionando en clases de equivalencia
- Valores frontera
- Árbol de clasificación
- Pruebas de pares
- Manejo de datos
- Tabla de decisión
- Diagrama de transición

Técnicas de caja blanca

- Basadas en el flujo de control
  - Cobertura de sentencias
  - Cobertura de decisiones
  - Cobertura de condiciones
  - Cobertura de caminos
- CodeCover
- Mutación
- Jumble

Cobertura de sentencias

Con la entrada **{ a = 2008 }** nos alcanza para conseguir cumplir con cobertura de sentencias:

```
public static boolean bisiesto(int a) {
    boolean b = false;
    if ((a%4==0) && (a%100!= 0))
        b = true;
    return b;
}
```

Cobertura de decisiones

Con la entrada **{ list = "neuquen", list = "pero"}** nos alcanza para conseguir cumplir con cobertura de decisiones:

```
public static boolean capicua(char[] list) {
    int index = 0;
    int l = list.length;
    while (index<(l-1)) {
        if (list[index] != list[(l-index)-1]) {
            return false;
        }
        index++;
    }
    return true;
}
```

Cobertura de condiciones

La satisfacción de esta técnica se dará cuando cada condición (de cada decisión) es ejecutada por verdadero y por falso por al menos un test de la suite, lo que no implica que sean todas las combinaciones:

```
if (index < count(list) && !found …
```

Cobertura de caminos

Con la entrada **{ {a=1, b=0}, {a=1, b=1}, {a=2, b=1}, {a=1, b=2} }** nos alcanza para conseguir cumplir con cobertura de caminos:

```
public static int mcd(int x, int y) {
    int a = x;
    int b = y;
    while (b != 0) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

CodeCover

Es una herramienta para medir cobertura de una test suite, de acuerdo a algunos criterios de caja blanca:

- Diseñada para Java+JUnit
- Se puede instalar como un plugin de Eclipse
- Muestra visualmente el código cubierto + estadísticas de cobertura.

PHPUnit + PHP_CodeCoverage + Xdebug

Dentro de la Suite de PHPUnit es posible hacer uso del componente PHP_CodeCoverage junto con Xdebug para ejecutar métricas de cobertura de código

Mutación

```
public static boolean bisiesto(int a) {
    boolean b = false;
    if ((a%4==0) && (a%100!= 0))
        b = true;
    return b;
}
```

```
public static boolean bisiesto(int a) {
    boolean b = false;
    if ((a%4==0) || (a%100!= 0))
        b = true;
    return b;
}
```

Jumble

Es una herramienta para evaluar suites de acuerdo a mutación:

- Diseñada para Java+JUnit
- Se puede instalar como un plugin de Eclipse
- Crea mutantes a partir del código original:
  - A nivel de bytecode
  - Mutantes: cambios en operadores booleanos y aritméticos, etc
  - Corre suites JUnit y mide score (% de mutantes muertos)
  - Reporta los mutantes que quedaron vivos
    - útil para mejorar la suite!

Otras

- JAVA:
  - PITEST: http://pitest.org/
  - JUDY: http://mutationtest.com/
- PHP
  - Mutagenesis – Mutate me: https://github.com/padraic/mutagenesis

Integración continua

- Consiste en hacer *integraciones automáticas* de un proyecto lo más a menudo posible para así poder detectar fallos cuanto antes
- Entendemos por integración la compilación y ejecución de tests de todo un proyecto
- El proceso suele ser, cada cierto tiempo, descargarse las fuentes desde el gestor de versiones, compilarlo, ejecutar tests y generar informes

Ventajas

- Detectar y solucionar problemas de integración de forma continua
- Disponibilidad constante de una build para pruebas, demos o lanzamientos anticipados
- Ejecución inmediata de las pruebas unitarias
- Monitorización continua de las métricas de calidad del proyecto

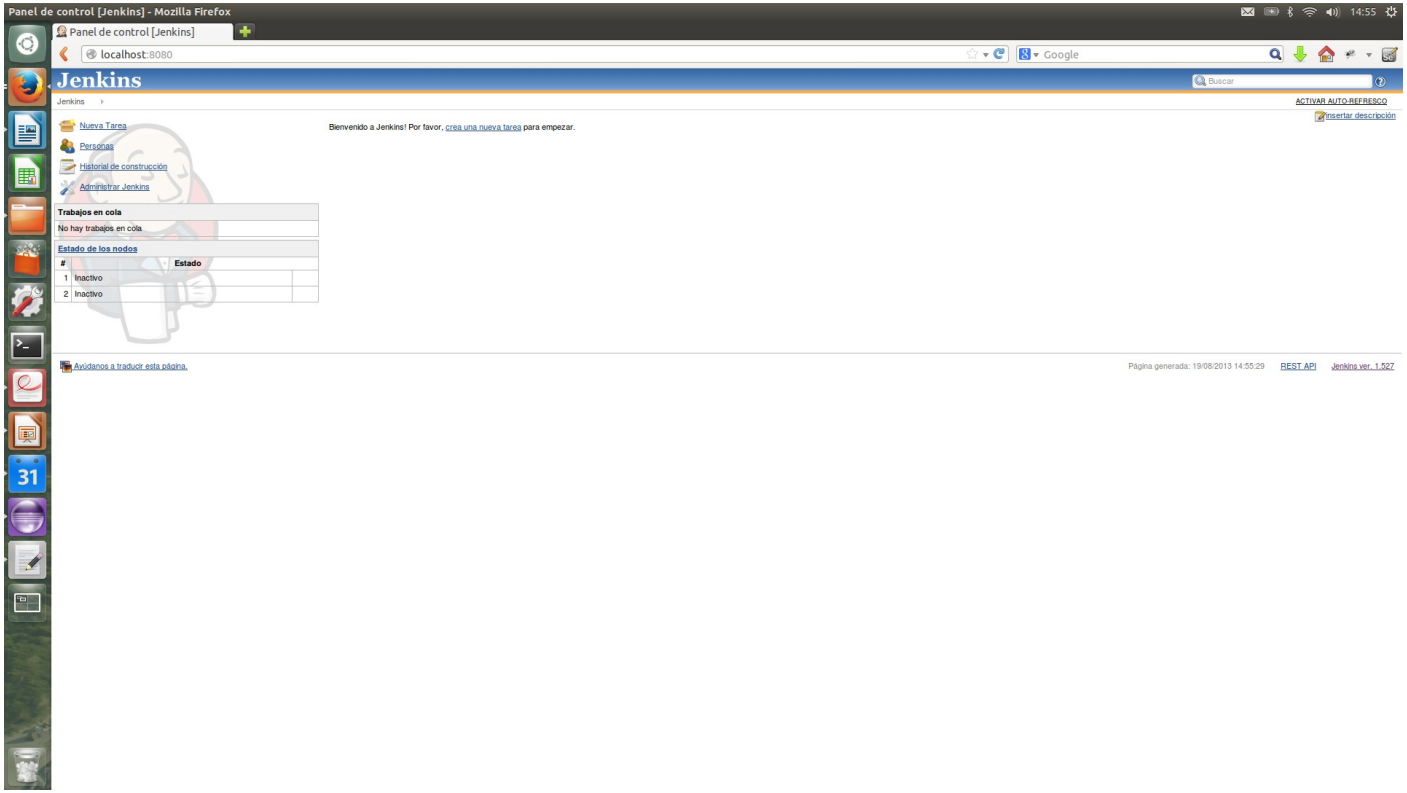
Jenkins

Jenkins

- Es un software de Integración continua open source escrito en Java.
- Proporciona integración continua para el desarrollo de software
- Es un sistema corriendo en un servidor que es un contenedor de servlets, como Apache Tomcat
- Funciona para un gran número de lenguajes

Integración continua

Jenkins



Cierre

Ministerio de Industria
Presidencia de la Nación

INTI

Av. Vélez Sarsfield 1561
(5000) Córdoba Capital
Córdoba, Argentina
+549 (0351) 4603974 Int. 112
fbobbio@inti.gob.ar

Abril de 2014