



# Ingeniería en Software

pruebas de software

# Objetivos de la Clase

- Qué son las pruebas del software.
- Pruebas unitarias, de caja blanca y caja negra.
- Coverage
- Mocks

# Pruebas

*“Las pruebas intentan demostrar que el programa hace lo que se espera que se haga, así como descubrir defectos en el programa antes de usarlo”*

## **Dos metas:**

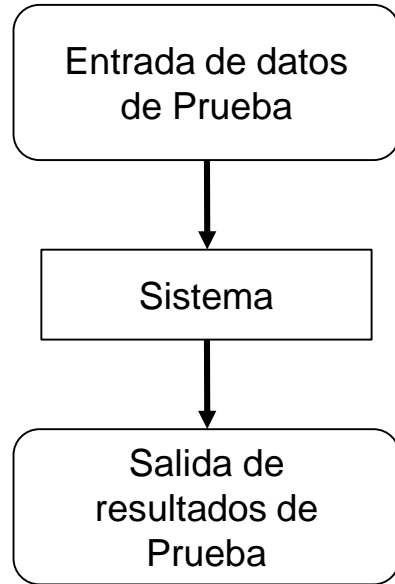
1. Cumple con la documentación de requerimientos.
1. Encontrar situaciones de funcionamiento incorrecto

# Pruebas

*“Las Pruebas pueden mostrar solo la presencia de errores, pero no su ausencia.”*

*Dijkstra et al., 1972*

# Pruebas



Las entradas buscan un  
comportamiento anómalo

Las salidas revelan  
la presencia de defectos

# Pruebas

Tienen 2 componentes:

- ❖ Validación: ¿Construimos el producto que quiere el cliente?
- ❖ Verificación: ¿Construimos bien el producto?

# Nivel de Confianza.

1. Propósito del software: Nivel de criticidad.
1. Expectativas del usuario: Tolerancia a Fallos.
1. Entorno de Mercado: Precio dispuestos a pagar.

# Inspecciones

- Son estáticas se realizan a través de la documentación.
- Se verifica que se cumpla con la documentación.
- Se enfocan en el código fuente.



# Inspecciones

## Ventajas:

- ❖ Se realizan en cualquier estadio del proyecto.
- ❖ Al ser estáticas, no se ven afectadas por cambios en el código.
- ❖ Se puede analizar cumplimientos de estándares, rendimiento de algoritmos, etc.

# Inspecciones

## Desventajas:

- ❖ No sustituyen las pruebas de software.
- ❖ Su ejecución depende de conformar un equipo que entienda todo el proyecto pero no está inmerso en el.
- ❖ Se debe esperar a cada revisión para detectar errores.

# Etapas de Prueba

Pruebas de Desarrollo: Durante el proceso se pone a prueba constantemente para descubrir errores y defectos.

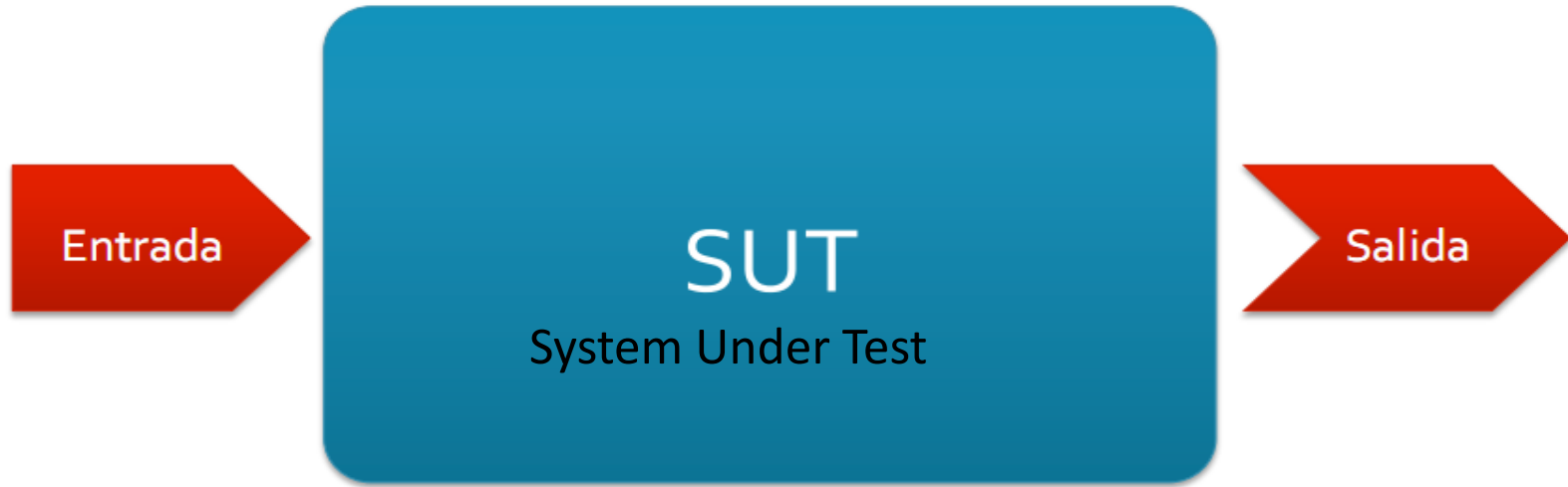
Versiones de Prueba: Se prueba todo un sistema en conjunto, se ponen a prueba requerimientos

Pruebas de usuario: se realizan del lado del cliente.

# Pruebas de Desarrollo.

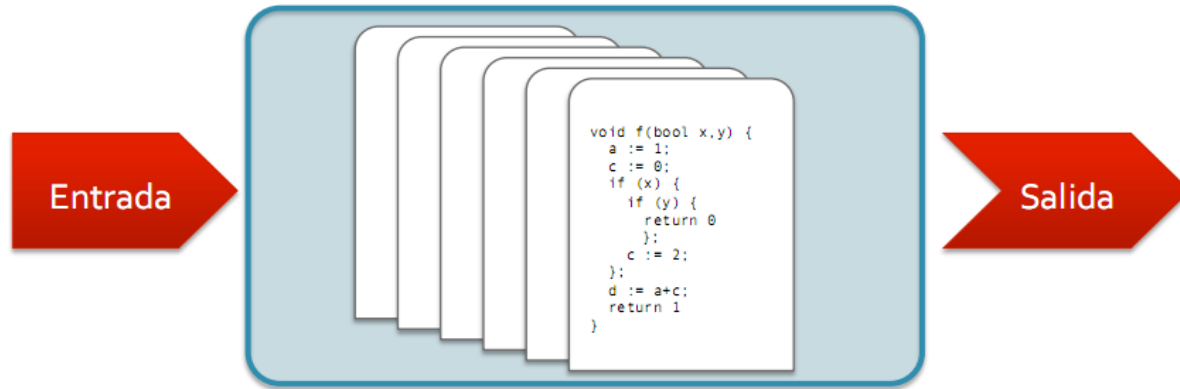
- **Pruebas unitarias:** Se prueban las unidades mínimas, objetos o métodos.
- **Pruebas de Componentes:** Se enfoca en probar las interfaces entre varios componentes.
- **Pruebas de Sistema (o de integración):** Gran parte o todos los componentes se integran y se prueba como un todo

# Técnicas de Prueba - Caja Negra



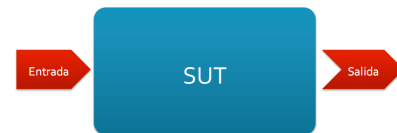
Testing de caja negra

# Técnicas de Prueba - Caixa Blanca



Testing de caja blanca

# Caja Negra



Testing de caja negra

- Están basadas en la definición de requerimientos o una descripción funcional del producto bajo prueba.
- Se centran en “qué hace” el producto y no en “cómo lo hace”.
- No observan el comportamiento interno, lo que dificulta la localización de defectos

# Pruebas unitarias.

- Es una metodología para testear unidades individuales de código, preferentemente de forma aislada de sus dependencias
- Las unidades pueden ser de distinta granularidad, porción de código, método, clase, librería, etc.
- Normalmente las pruebas son creadas por los mismos programadores durante el proceso de desarrollo.
- Normalmente se utiliza uno o mas frameworks para su sistematización y automatización.



# Pruebas Unitarias

Una librería (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.

# Pruebas Unitarias Python.

**Unittest** es un paquete estándar de Python que permite, como su nombre indica, hacer pruebas unitarias. Consiste en

- ❑ Hacer una clase que extienda `unittest.TestCase`
- ❑ Añadir métodos que empiecen por "test\_" con la implementación de las pruebas
- ❑ Hacer aserciones para comprobar que el resultado es el esperado

# Ejemplo

```
1. def cuadrado(num):  
2.     """Calcula el cuadrado de un numero."""  
3.  
4.     return num ** 2
```

Test:

```
1. import unittest  
2. import cuadrado  
3. class EjemploPruebas(unittest.TestCase):  
4.     def test(self):  
5.         l = [0, 1, 2, 3]  
6.         r = [cuadrado(n) for n in l]  
7.         self.assertEqual(r, [0, 1, 4, 9])
```

# Estructura básica

```
import unittest
```

```
class SimplisticTest(unittest.TestCase):
```

```
    def test(self):
```

```
        self.assertTrue(True)
```

```
if __name__ == '__main__':
```

```
    unittest.main()
```

# Asserts

Method	Checks that	New in
<code>assertEqual(a, b)</code>	<code>a == b</code>	
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	
<code>assertTrue(x)</code>	<code>bool(x)</code> is True	
<code>assertFalse(x)</code>	<code>bool(x)</code> is False	
<code>assertIs(a, b)</code>	<code>a is b</code>	2.7
<code>assertIsNot(a, b)</code>	<code>a is not b</code>	2.7
<code>assertIsNone(x)</code>	<code>x is None</code>	2.7
<code>assertIsNotNone(x)</code>	<code>x is not None</code>	2.7
<code>assertIn(a, b)</code>	<code>a in b</code>	2.7
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	2.7
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	2.7
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	2.7

# assertTrue o assertFalse

Nos permiten saber si el resultado es verdadero o falso:

Probemos **test\_truth.py** y vemos que nos dice.

# Nuestro primer test.

Ejecutemos primero:

```
$ python3 test_simple.py
```

Después:

```
$ python3 test_simple.py -v
```

**-v me permite ver test a Test.**

# Salidas

unittest tiene 3 posibles salidas:

- OK : Se ejecutó correctamente el Test y paso el **assert**
- FAIL: Se ejecutó correctamente el Test y **NO** paso el **assert**
- ERROR: Se ejecutó erróneamente el Test

Ejecutemos el archivo **test\_outcomes**



# Ahora vamos a hacer nuestros propios test

- Miremos el archivo de la Carpeta SUT.
- Vamos a hacer los test para este código.
- Primero para la función área.

```
import unittest
import sut

class TestSut(unittest.TestCase):

    def tests_area(self):
        area = sut.area(3,2)
        self.assertTrue(area==6)

if __name__ == '__main__':
    unittest.main()
```

# Manos a la obra

Hagamos nuestros propios test para las funciones:

**Saludar, sumar, sumatoria, productoria, valor\_absoluto**

# Caja Blanca



Testing de caja blanca

- ❖ A diferencia del testing de caja negra, en los criterios de test de caja blanca se analiza el código del SUT para decidir si una suite es adecuada o no.
- ❖ Es decir, los criterios de caja blanca se enfocan en la implementación.
- ❖ Muchos de los criterios **exploran la estructura del código a testear**, intentando dar casos que ejerciten el código de maneras diferentes.

# Cobertura de sentencias



Testing de caja blanca

- Una test suite satisface el criterio de **cobertura** de sentencias si todas las sentencias del programa son ejecutadas al menos una vez por algún test de la suite.
  - Es uno de los criterios de caja blanca más débiles!
- Errores en condiciones compuestas y ramificaciones de programas pueden ser pasados por alto.
- En muchos casos, con suites pequeñas se puede satisfacer este criterio.

# Cobertura de decisión



Testing de caja blanca

- Una decisión es un punto en el código en el que se produce una ramificación o bifurcación.

*Ej.: condiciones de ciclos, condiciones de **if-else***

- ❖ Una suite satisface el criterio de cobertura de decisión si todas las decisiones del programa son ejecutadas por true y por false al menos una vez.
  - Propiedad: cobertura de decisión es más fuerte que cobertura de sentencias.
  - **Si una suite satisface cobertura de decisión, también satisface cobertura de sentencias.**

# Cobertura de condición



Testing de caja blanca

Una decisión puede estar compuesta por una o más condiciones

*Ej.: If  $i < \text{size}$  and not found*

Una suite satisface el criterio de cobertura de condición si cada condición de cada decisión es ejecutada por true y por false al menos una vez.

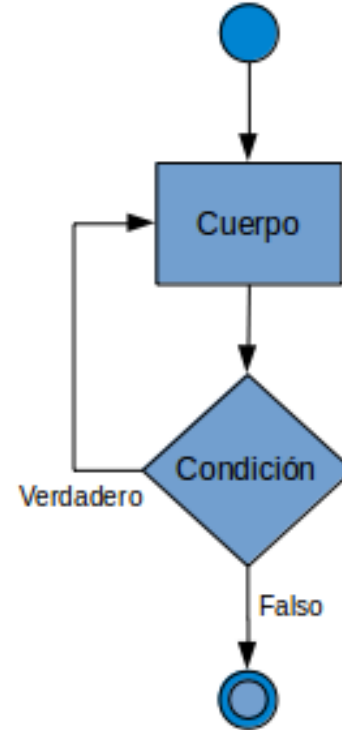
No es lo mismo que todas las combinaciones!

OJO: cobertura de condición NO es más fuerte que cobertura de decisión.

# Grafos de flujo de control

El grafo de flujo de control de un programa es una representación, mediante grafos dirigidos, del flujo de control del programa:

- Los nodos del grafo representan segmentos de sentencias que se ejecutan secuencialmente.
- Las decisiones representan bifurcaciones.
- Los arcos del grafo representan transferencias de control entre nodos.



## Testing de caja blanca

# Cobertura de caminos

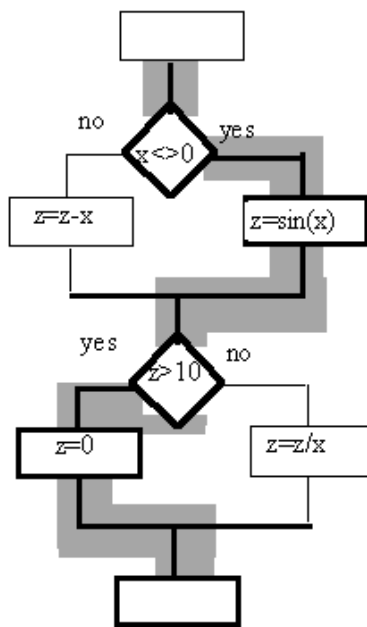
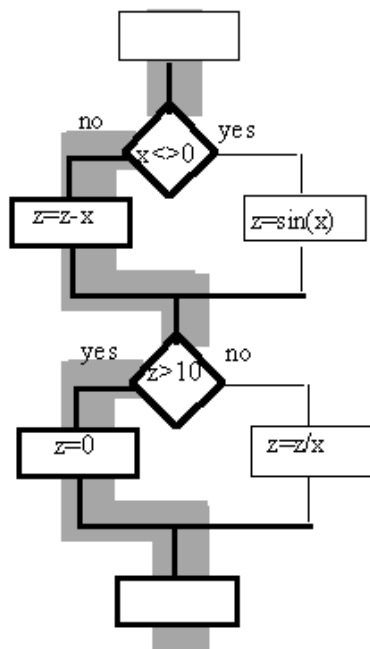
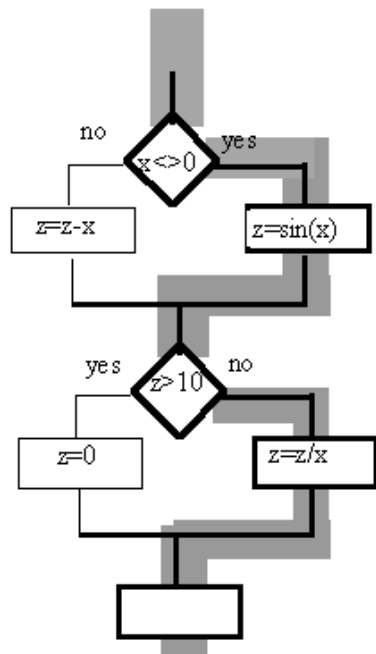


Testing de caja blanca

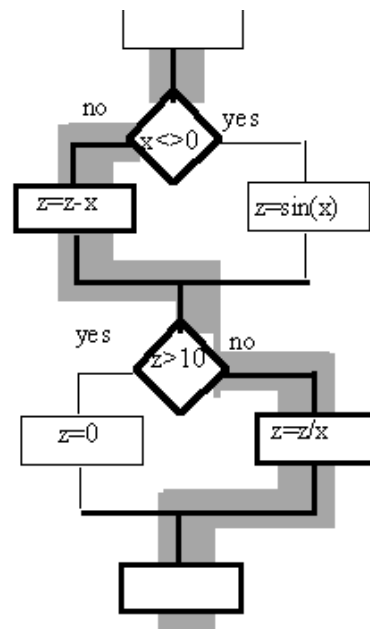
- Una suite satisface el criterio de cobertura de caminos si todos los caminos del grafo de flujo de control del programa SUT son recorridos al menos una vez.
  - Es un criterio muy fuerte: conseguirlo puede requerir suites muy grandes.
- Suelen imponerse restricciones al criterio para hacerlo practicable:
  - cobertura de caminos simples: requiere cubrir caminos sin repetición de arcos.
  - cobertura de caminos elementales: requiere cubrir caminos sin repetición de nodos.



# Cobertura de caminos



Testing de caja blanca



# Herramientas de Cobertura

*\$pip install coverage*

*\$coverage run test.py*

*\$ coverage report -m*

Name	Stmts	Miss	Cover	Missing
-----				
sut.py	26	11	58%	7, 10, 13-18, 21-23
test.py	26	4	85%	21-24
-----				
TOTAL	52	15	71%	

*\$ coverage html*



Testing de caja blanca

# Coverage



Testing de caja blanca

Coverage for **util** : 80%

5 statements

4 run

1 missing

0 excluded

```
1 | from math import ceil, pi
2 |
3 | def borkify(i):
4 |     return int(ceil(i/pi) + 1)
5 |
6 | def fnord(j):
7 |     return j/2 - 3
```

# Coverage



Testing de caja blanca

```
494 | @staticmethod
495 | def _make_xml_node(xml):
496 |     """Transform a variety of input formats to an XML DOM node."""
497 |     try:
498 |         ntype = xml.nodeType
499 |     except AttributeError:
500 |         if isinstance(xml, basestring):
501 |             try:
502 |                 xml = minidom.parseString(xml)
503 |             except Exception, e:
504 |                 raise XmlError(e)
505 |         elif hasattr(xml, "read"):
506 |             try:
507 |                 xml = minidom.parse(xml)
508 |             except Exception, e:
509 |                 raise XmlError(e)
510 |         else:
511 |             raise ValueError("Can't convert that to an XML DOM node")
512 |         node = xml.documentElement
513 |     else:
514 |         if ntype == xml.DOCUMENT_NODE:
515 |             node = xml.documentElement
516 |         else:
517 |             node = xml
518 |     return node
519 |
```

# Coverage

Vamos correr:

**\$sudo pip3 install coverage**

**\$coverage run testsut.py**

**\$coverage report**

**\$coverage html**

Y vamos a la carpeta htmlcov, abrimos index.html miremos. Ahora click sobre sut.py

# Test con coverage

Hagamos los test de **valor absoluto**, primero usamos un valor positivo miremos el coverage. Y ahora?

Armemos los tests para **comparar**, son varios.

# Estructuras

## **setup:**

permite inicializar nuestro entorno, es decir, establecer un entorno de pruebas para cada prueba. Se ejecutará siempre antes de cada test.

## **tearDown:**

que realiza la operación contraria, es decir, se ejecuta siempre después de cada test, generalmente para limpiar lo creado.

# Estructuras

```
import unittest
```

```
class WidgetTestCase(unittest.TestCase):
```

```
    def setUp(self):
```

```
        self.widget = Widget('The widget')
```

```
    def tearDown(self):
```

```
        self.widget.dispose()
```

```
        self.widget = None
```

```
    def test_default_size(self):
```

```
        self.assertEqual(self.widget.size(), (50,50),  
                           'incorrect default size')
```

```
    def test_resize(self):
```

```
        self.widget.resize(100,150)
```

```
        self.assertEqual(self.widget.size(), (100,150),  
                           'wrong size after resize')
```

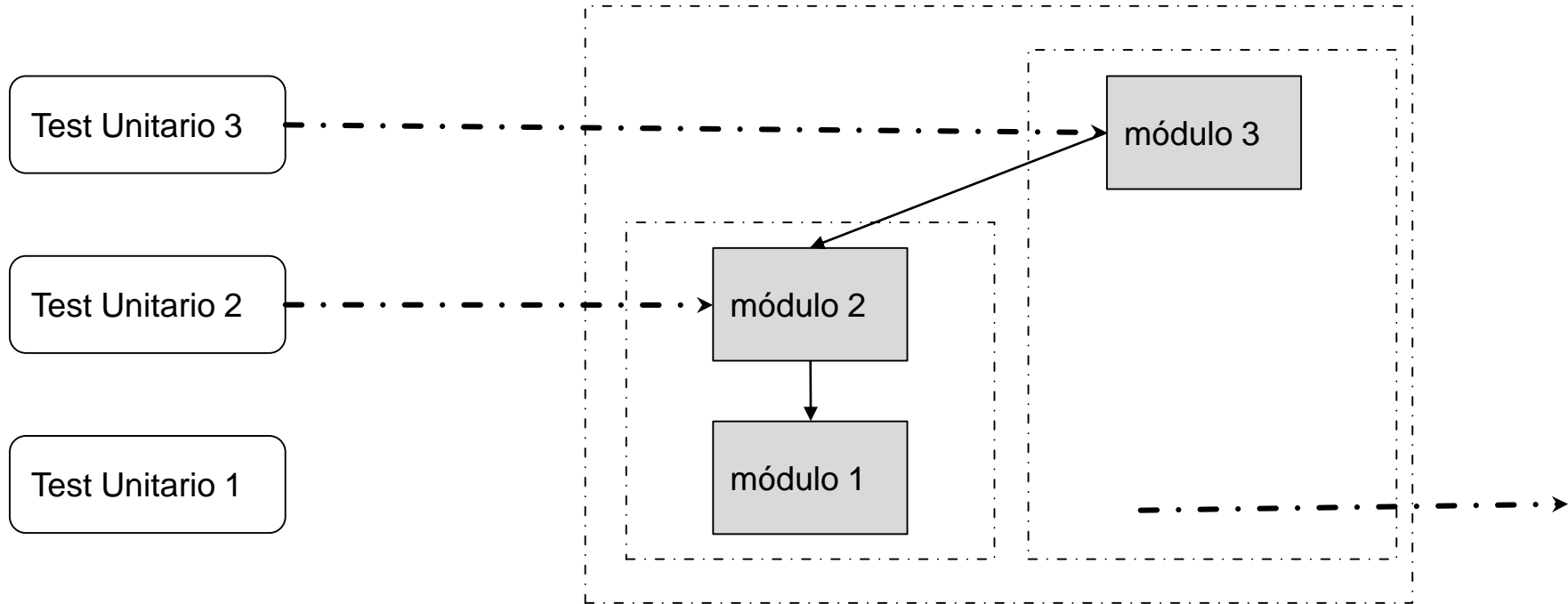


# Dobles de prueba

- 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.
- 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.

# Problemas de Dependencias.

Tradicionalmente un proyecto de software se integra de adentro hacia afuera

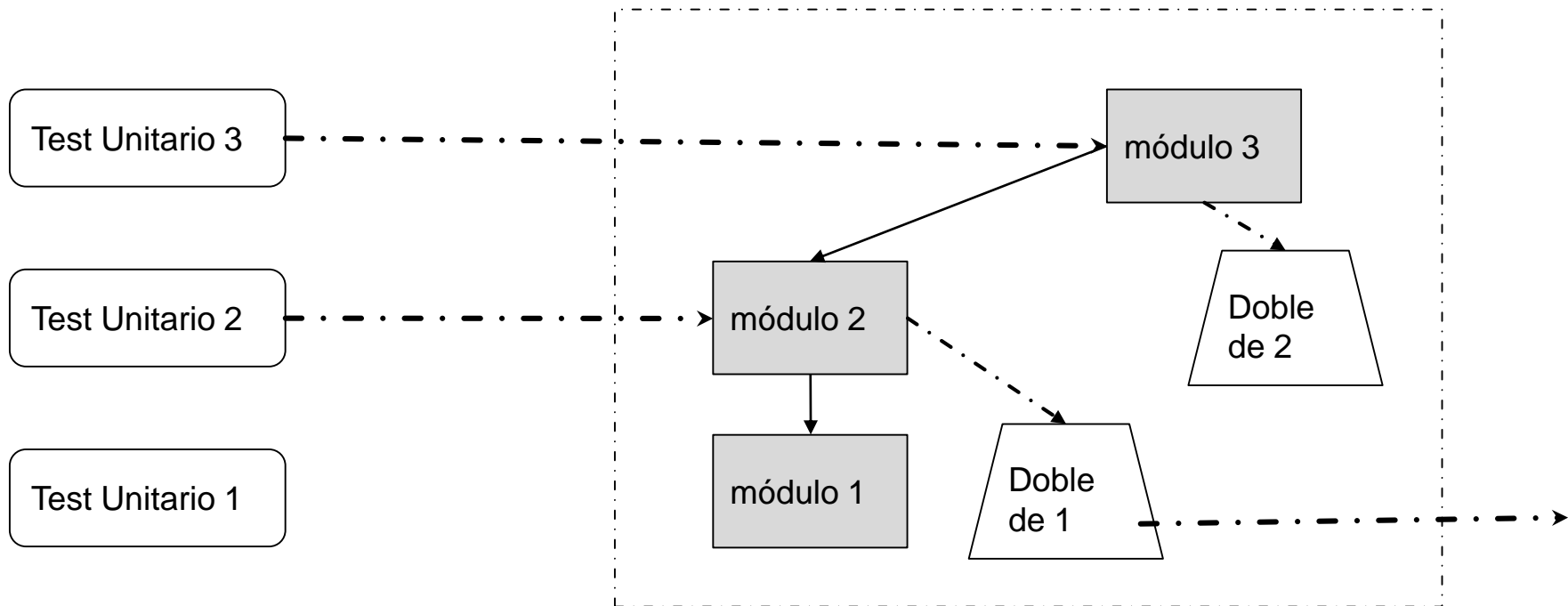


# Dobles

- En un test unitario solo debería estar probando el modulo, no sus dependencias.
- Los dobles son objetos que imitan el comportamiento de objetos reales de una forma controlada.
- La expresión doble se usa en el mismo sentido de los actores dobles en las películas de acción, ya que se hace pasar por un colaborador del SUT.

# Soluciones de Dependencias.

Podemos usar dobles!!



# Razones

1. Devuelven resultados no determinísticos (por ejemplo la hora o la temperatura)
2. Su estado es difícil de crear o reproducir (por ejemplo errores de conexión)
3. Es lento (por ejemplo el resultado de un cálculo intensivo o una búsqueda en una BBDD)
4. El objeto todavía no existe o su comportamiento puede cambiar.
5. Debería incluir atributos o métodos exclusivamente para el testeo.

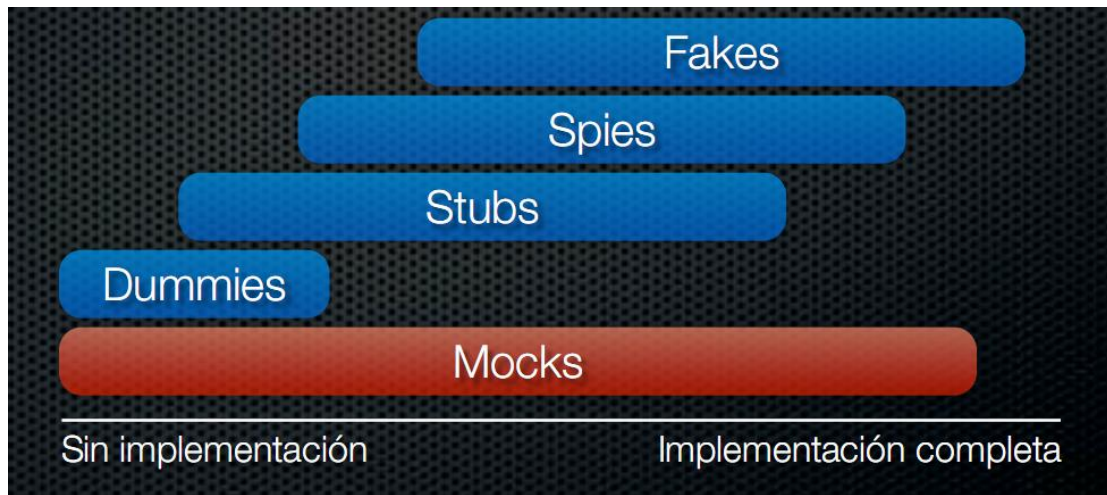
# Tipos de Dobles

- **Dummy**: se pasa como argumento pero nunca se usa realmente. Normalmente, los objetos dummy se usan sólo para rellenar listas de parámetros.
- **Fake**: tiene una implementación que realmente funciona pero, por lo general, toma algún atajo o cortocircuito que le hace inapropiado para producción (como una base de datos en memoria por ejemplo).
- **Stub**: proporciona respuestas predefinidas a llamadas hechas durante los tests, frecuentemente, sin responder en absoluto a cualquier otra cosa fuera de aquello para lo que ha sido programado. Los stubs pueden también grabar información sobre las llamadas; tal como una pasarela de email que recuerda cuántos mensajes envió.
- **Mock**: objeto preprogramado con expectativas que conforman la especificación de cómo se espera que se reciban las llamadas. Son más complejos que los stubs aunque sus diferencias son sutiles.

# Tipos de dobles

Aunque los tipos parecen diferentes en teoría, en la práctica las diferencias se vuelven borrosas.

Parece más apropiado, pensar los dobles como miembros de un continuo:



# Mocks

- Mock es una denominación general para dobles que controlan entrada y salida indirecta.
- En general los mocks se crean en tiempo de ejecución con la ayuda de una librería específica, que permite:
  - En una primera fase, se crea el objeto cuyos métodos serán invocados.
  - En una segunda fase se especifica el comportamiento esperado.
  - En una tercera fase se verifica el comportamiento ejercido respecto al especificado.
  - No mezclar las fases.
- Así, no es necesario escribir el código que implementa el mock.



# Mocks en Python

< python 3.3

```
$pip install mock
```

```
import mock
```

desde python 3.3 integrado

```
import unittest
```

# Magic Mock

Código:

```
def calcula(val):  
    lb = cuadrado(val)  
    a= dividir(lb)  
    return a
```

```
def cuadrado(n):  
    return n**2
```

```
def dividir(n):  
    return n/2
```

# Magic Mock

Test:

```
import calculos
import unittest
from unittest.mock import MagicMock

class TestBase(unittest.TestCase):

    def test_rightSequenceOfCalls(self):
        calculos.cuadrado = MagicMock(return_value=2)
        calculos.dividir = MagicMock(return_value=2)
        a = calculos.calcula(5)
```

# Mock

Ahora miremos la función **costototal**, usa la función suma, tenemos que hacer un mock sobre eso.

Para eso importamos:

```
from unittest.mock import MagicMock
```

y dentro del test vamos a hacer

```
sut.sumar=MagicMock(return_value=2)
```

Esto hace que no llamemos a suma y siempre devuelva valor 2

# Mock más complejos

Miremos la función **supercalc**, usa 2 funciones de la librería math, intentemos armar un mock, con Mockito.

## ¿Qué problemas encontramos?

# Mock más complejos

Debería verse:

```
def test_supercalc(self):  
    math.exp=MagicMock(return_value=2)  
    math.sqrt=MagicMock(return_value=2)  
    a = sut.supercalc(3)  
    self.assertTrue(a == 2)
```

Pero algo nos falta, importamos la libreria math en nuestro test.  
MagicMock, necesita conocer la librería a mockear.

# Patch

- Los decoradores de parche se utilizan para parchear los objetos sólo en el ámbito de la función que decorar.
- Ellos se encargan de forma automática el unpatching.
- No necesitamos conocer la librería, simplemente va armar el mock de la llamada.

# Patch

Funciona como un decorador de la siguiente manera.

```
@patch('modulo.funcion2') # fijense que va como string
@patch('modulo.funcion1')
def parchando(self, funcion1, funcion2): # se reciben como
    parámetros
        funcion1.return_value = 2
    funcion2.return_value = 2
```

**No hace falta, importar las librería** patch nos resuelve todo, armemos nuestro test ahora sacando la librería math y usando patch.



# Patch

Debería verse algo así:

```
@patch('math.exp')
@patch('math.sqrt')
def test_supercalc(self, sqrt, exp):
    sqrt.return_value = 2
    exp.return_value = 2
    a = sut.supercalc(3)
    self.assertTrue(a == 2)
```