CAPÍTULO 3

Bonus: Mocking Objects

Imagina que tu quieres testear un protocolo de red, y no quieres depender de una red real para crear tus pruebas de unidad. Lo que se puede hacer es crear un objeto que se simule el comportamiento de una red real, solo para los métodos y parametros que necesitas en tus pruebas. Un mock object es un objeto que permite realizar pruebas de unidad en insolación, es decir, permite testecar la unidad sin depender de otros componentes o elementos externos.

Existen varios frameworks para definir mock objects en todos los lenguajes de programación en este capitulo bonus veremos la libreria de mocks de python.

3.1 Objetivos

Entender que es un mock object y crear una prueba de unidad utilizando un mock object en python.

3.2 Definición

Un mock object, es un «test double» que permite se utilizado como un stub, a fake, or a mock. Este objeto proporciona un protocolo simple (métodos), donde el usuario puede enseñar a este objeto que responder a ciertos mensajes, simulando asi parcialmente el comportamiento del objeto real solo para realizar pruebas.

3.3 Metodos Auxiliares

```
def runTests(testClass):
  # Create a test suite
  suite = unittest.TestSuite()
  # Add tests from the test class
  suite.addTest(unittest.makeSuite(testClass))
  # Run the test suite
```

(continúe en la próxima página)

(proviene de la página anterior)

```
  runner = unittest.TextTestRunner()
  runner.run(suite)
```

3.4 Mi Primer Mock

Primero veamos un ejemplo donde una clase depende de un API externo para realizar una operación. El problema de este escenario es que no podria probar esta clase de forma aislada, y el test dependeria que el servidor donde este el API este encendido y funcional. Considere el siguiente ejemplo:

```
class Warehouse:
    def quantityOnStock(self,productName):
        #very complex method
        print("Llamando a un API externo")
        # devuelvo 5 para que compile el API externo podria devolver cualquier número
        return 5

class Order:
  def __init__(self,name, quantity):
    self.name = name
    self.requestedQuantity = quantity

  def checkAvailability(self, warehouse):
    return warehouse.quantityOnStock(self.name)  >= self.requestedQuantity

  def setEmailService(self,service):
    self.service = service

  def sendByEmail(self):
    self.service.sendOrder(self.name + ":" +  str(self.requestedQuantity))
```

Crear un unit test para este clase se realizaria de la siguiente forma:

```
import unittest

class TestOrder(unittest.TestCase):

  def test_order_is_not_available_when_stock_is_insufficient(self):
    # Arrange
    order = Order("iPhone 14", 10)
    warehouse = Warehouse()
    # Act
    available = order.checkAvailability(warehouse)
    # Assert
    self.assertFalse(available, "Expected product to be unavailable due to␣
 ↪insufficient stock.")


runTests(TestOrder)
```

```
/var/folders/gm/n29_08rj557_fclvr60s6gpw0000gn/T/ipykernel_62314/831825700.py:5:␣
 ↪DeprecationWarning: unittest.makeSuite() is deprecated and will be removed in␣
 ↪Python 3.13. Please use unittest.TestLoader.loadTestsFromTestCase() instead.
  suite.addTest(unittest.makeSuite(testClass))
.
```

```
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

```
Llamando a un API externo
```

Como puede ver el código anterior pasa, pero internamente esta llamando a un API externo. Para testear que la clase Order funciona bien sin llamar al API externo, crearemos un mock de la clase Warehouse. A este mock solo le enseñaremos a que cuando alguien quiera usar el método quantityOnStock, este siempre devuelva 6.

```
from unittest.mock import MagicMock
from unittest.mock import patch
#from warehouse import Warehouse
import unittest

class TestOrderV2(unittest.TestCase):

  @patch('__main__.Warehouse')  # Cambia '__main__' por 'warehouse' si está en otro␣
↪módulo
  def test_order_not_available_due_to_low_stock(self, MockWarehouse):
    # Arrange
    mock_warehouse = MockWarehouse()
    mock_warehouse.quantityOnStock = MagicMock(return_value=5)
    order = Order("iPhone 14", 10)

    # Act
    available = order.checkAvailability(mock_warehouse)

    # Assert
    self.assertFalse(available, "Expected the order to be unavailable due to␣
↪insufficient stock.")
    mock_warehouse.quantityOnStock.assert_called_once_with("iPhone 14")
```

Ejecutamos las pruebas de esa clase:

```
runTests(TestOrderV2)
```

```
/var/folders/gm/n29_08rj557_fclvr60s6gpw0000gn/T/ipykernel_62314/831825700.py:5:␣
↪DeprecationWarning: unittest.makeSuite() is deprecated and will be removed in␣
↪Python 3.13. Please use unittest.TestLoader.loadTestsFromTestCase() instead.
  suite.addTest(unittest.makeSuite(testClass))
.
```

```
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

3.5 Mas Ejemplos de Uso de la libreria Mock

```
import unittest
from unittest.mock import patch, MagicMock, call

class TestOrderV3(unittest.TestCase):

    # 1. Uso de @patch para reemplazar una clase entera
    @patch('__main__.Warehouse')
    def test_patch_class_and_mock_method(self, MockWarehouse):
        mock_warehouse = MockWarehouse()
        mock_warehouse.quantityOnStock.return_value = 5

        order = Order("iPhone", 10)
        result = order.checkAvailability(mock_warehouse)

        self.assertFalse(result)
        mock_warehouse.quantityOnStock.assert_called_once_with("iPhone")

    # 2. Uso de patch como context manager
    def test_patch_method_temporarily(self):
        with patch('__main__.Warehouse.quantityOnStock', return_value=20) as mock_
↪method:
            order = Order("iPhone", 10)
            warehouse = Warehouse()  # Método está parcheado
            result = order.checkAvailability(warehouse)

            self.assertTrue(result)
            mock_method.assert_called_once_with("iPhone")

    # 3. Uso directo de MagicMock
    def test_using_manual_magicmock(self):
        mock_warehouse = MagicMock()
        mock_warehouse.quantityOnStock.return_value = 8

        order = Order("iPhone", 10)
        result = order.checkAvailability(mock_warehouse)

        self.assertFalse(result)

    # 4. Simulación de múltiples respuestas (side_effect)
    def test_mock_with_multiple_returns(self):
        mock_warehouse = MagicMock()
        mock_warehouse.quantityOnStock.side_effect = [5, 15]

        order = Order("iPhone", 10)
        self.assertFalse(order.checkAvailability(mock_warehouse))
        self.assertTrue(order.checkAvailability(mock_warehouse))
        self.assertEqual(mock_warehouse.quantityOnStock.call_count, 2)

    # 5. Simulación de excepción como side_effect
    def test_mock_side_effect_exception(self):
        mock_warehouse = MagicMock()
        mock_warehouse.quantityOnStock.side_effect = RuntimeError("Connection failed")

        order = Order("iPhone", 10)
        with self.assertRaises(RuntimeError):
```

(continúe en la próxima página)

(proviene de la página anterior)

```
            order.checkAvailability(mock_warehouse)

    # 6. Verificación de múltiples llamadas con diferentes argumentos
    def test_mock_call_arguments(self):
        mock_warehouse = MagicMock()
        mock_warehouse.quantityOnStock.side_effect = lambda name: 10 if name ==
↪"iPhone" else 5

        order1 = Order("iPhone", 9)
        order2 = Order("Tablet", 6)

        self.assertTrue(order1.checkAvailability(mock_warehouse))
        self.assertFalse(order2.checkAvailability(mock_warehouse))

        self.assertEqual(mock_warehouse.quantityOnStock.call_args_list, [call("iPhone
↪"), call("Tablet")])
```

```
runTests(TestOrderV3)
```

```
/var/folders/gm/n29_08rj557_fclvr60s6gpw0000gn/T/ipykernel_62314/831825700.py:5:␣
 ↪DeprecationWarning: unittest.makeSuite() is deprecated and will be removed in␣
 ↪Python 3.13. Please use unittest.TestLoader.loadTestsFromTestCase() instead.
  suite.addTest(unittest.makeSuite(testClass))
.
```

```
.
```

```
.
```

```
.
```

```
.
```

```

```

```
----------------------------------------------------------------------
Ran 6 tests in 0.003s

OK
```

Tenemos el mismo resultado, pero sin llamar al API externo. Claro ahora tenemos un mock que solo simula el Warehouse con 6 de stock, pero cada test puede crear un mock diferente con diferente cantidad de stock. La idea es que ahora podemos testear la clase Order, sin llamar al API externo o depender de la funcionalidad de la otra clase.