CAPÍTULO 2

Capítulo 1: Unit Testing

2.1 Objetivos

Este capítulo presenta los conceptos fundamentales de las pruebas unitarias mediante ejemplos simples. Aprenderás a utilizar los métodos `setUp` y `tearDown`, entender su propósito dentro del ciclo de vida de una prueba, y reconocer los principales desafíos al diseñar pruebas eficaces.

Las pruebas unitarias son una de las herramientas más efectivas para detectar errores de forma temprana y asegurar la calidad del software.

Definición

Desde un punto de vista técnico, una prueba unitaria es un pequeño programa que permite verificar que un componente del programa está funcionando correctamente. Estos componentes, también llamados unidades del programa, suelen ser métodos, funciones o clases.

2.2 Mi Primera Prueba de Unidad

Para comprender cómo escribir pruebas unitarias, trabajaremos con un ejemplo simple: una clase que representa una billetera. Esta clase permite depositar y retirar dinero, y consultar el saldo disponible.

```
# utilizaremos la libreria unittest de python
import unittest
```

Para aprender a cómo escribir pruebas unitarias, utilizaremos un ejemplo sencillo. A continuación, tenemos una clase que modela el funcionamiento de una billetera. La misma permite depositar y retirar dinero, y también tiene un método que permite obtener el monto actual de la billetera.

```
class Wallet:
    def __init__(self, owner):
```

(continúe en la próxima página)

(proviene de la página anterior)

```
        self.balance = 0
        self.owner_name = owner

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
        else:
            print("Not enough minerals")

    def total_balance(self):
        return self.balance
```

A continuación se presenta un ejemplo de prueba unitaria para verificar el comportamiento esperado de la clase `Wallet`.

```
class WalletTest(unittest.TestCase):

  def test_deposit(self):
    # Arrange
    wallet = Wallet("juampi")
    # Act
    wallet.deposit(10)
    # Assert
    self.assertEqual(10, wallet.total_balance())
```

Existen varias formas de ejecutar las pruebas de unidad. Inicialmente crearemos una funcion que nos permita ejecutar todos los test dentro de una clase:

```
def runTests(testClass):
  # Add tests from the test class
  loader = unittest.TestLoader()
  suite = loader.loadTestsFromTestCase(testClass)
  # Run the test suite
  runner = unittest.TextTestRunner()
  runner.run(suite)

runTests(WalletTest)
```

```
.
```

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

OK
```

Hasta este punto usted vio como crear una prueba de unidad para una clase, en el ejemplo, la clase `Wallet`.

2.3 Estructura de una Prueba Unitaria

Las pruebas unitarias normalmente están conformadas por tres partes:

- **Arrange**: Primero, las pruebas tienen un conjunto de instrucciones que crean objetos o datos que serán utilizados para realizar la prueba. Esto incluye la configuración del entorno de prueba, inicialización de variables y la creación de objetos necesarios.
- **Act**: Posteriormente, existen instrucciones que ejecutan la unidad a probar, por ejemplo, la función que queremos evaluar. Esta parte implica llamar al método o función que estamos probando con los datos preparados en la etapa anterior.
- **Assert**: Finalmente, se tienen un conjunto de instrucciones que verifican que la unidad a probar funcionó correctamente. Esto incluye la comprobación de los resultados obtenidos contra los resultados esperados, verificando si el valor retornado es el correcto o si el estado final de los objetos que participaron en la prueba es el deseado.

2.4 Métodos setUp y tearDown

Cuando se crean varios tests para una clase en particular, es común encontrar que varios tests puedan necesitar una configuración (**Arrange**) similar. Los métodos setUp y tearDown de unittest permiten establecer y limpiar este entorno común.

Considere el siguiente ejemplo que contiene dos tests para la clase `Wallet`:

```
class WalletTest(unittest.TestCase):
    def test_deposit(self):
        self.wallet = Wallet("juampi")
        self.wallet.deposit(100)
        self.wallet.deposit(50)
        balance = self.wallet.total_balance()
        self.assertEqual(150, balance)

    def test_withdraw(self):
        self.wallet = Wallet("juampi")
        self.wallet.deposit(100)
        self.wallet.withdraw(30)
        balance = self.wallet.total_balance()
        self.assertEqual(70, balance)
```

Es posible refactorizar el código anterior haciendo uso del método `setUp`. Como se puede ver a continuación

```
class WalletTest(unittest.TestCase):

    def setUp(self):
        # Inicializa una billetera que se puede utilizar en los test
        # Note que este método se ejecuta por cada test
        self.wallet = Wallet("juampi")
        self.wallet.deposit(100)

    def tearDown(self):
        # Este método es opcional y se usa para limpiar recursos si es necesario.
        pass

    def test_deposit(self):
        self.wallet.deposit(50)
```

(continúe en la próxima página)

(proviene de la página anterior)

```
        balance = self.wallet.total_balance()
        self.assertEqual(150, balance)

    def test_withdraw(self):
        self.wallet.withdraw(30)
        balance = self.wallet.total_balance()
        self.assertEqual(70, balance)
```

El método teardown, normalmente sirve para limpiar el entorno, variables entre otras cosas antes que se ejecute el siguiente tests. Recuerde que ahora si ejecutamos los test de la clase WalletTest ahora se ejecutarán dos test por lo que en consola se mostrarán dos puntos.

```
runTests(WalletTest)
```

```
.
```

```
.
```

```

```

```
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

2.5 Caracteristicas deseables de una prueba de unidad

A continuación, se describen un conjunto de carateristicas que las pruebas de unidad deben cumplir:

- **Simplicidad**: Una prueba unitaria debe ser simple y enfocarse en una sola funcionalidad o comportamiento. Esto facilita la identificación de fallos y asegura que cada test tenga un propósito claro y definido.
- **Aislamiento**: Cada prueba debe ser independiente de las demás. Esto significa que las pruebas no deben compartir estado o depender unas de otras. El uso de métodos como `setUp` y `tearDown` ayuda a mantener este aislamiento al configurar y limpiar el entorno de prueba para cada test individualmente.
- **Repetibilidad**: Las pruebas unitarias deben ser repetibles, es decir, deben producir los mismos resultados cada vez que se ejecutan, independientemente del entorno en el que se corran. Esto asegura la fiabilidad de los tests y la confianza en los resultados obtenidos.
- **Automatización**: Las pruebas unitarias deben ser fácilmente automatizables para permitir su ejecución frecuente como parte del proceso de desarrollo continuo. Esto ayuda a detectar y corregir errores de manera temprana en el ciclo de desarrollo.
- **Claridad**: Las pruebas deben ser claras y fáciles de entender. Los nombres de los métodos de prueba, así como los mensajes de aserción, deben describir claramente lo que se está probando y qué resultado se espera.

Al aplicar estos primeros principios al escribir pruebas unitarias, se logra un conjunto de tests más robusto, mantenible y eficaz en la detección de errores y problemas en el código.

2.6 Pruebas de unidad y el encapsulamiento

El encapsulamiento es la capacidad que tiene una clase de ocultar información, proporcionando solo métodos públicos para interactuar con ella. Las clases normalmente tienen atributos privados para que las demás clases no dependan de sus datos sino de sus operaciones públicas, reduciendo así el acoplamiento entre clases.

Si bien el encapsulamiento es una propiedad deseable, a veces dificulta la creación de pruebas de unidad. Al crear pruebas de unidad, puede ser necesario verificar que los atributos de un objeto estén en el estado correcto después de ejecutar la unidad bajo prueba. Por ejemplo, analicemos nuestro ejemplo anterior.

```
class WalletTest(unittest.TestCase):

  def test_deposit(self):
    # Arrange
    wallet = Wallet("juampi")
    # Act
    wallet.deposit(10)
    # Assert: esta linea accede al atributo de un objeto para verifiar su valor
    self.assertEqual(10, wallet.total_balance())
```

Se dice que una clase tiene baja observabilidad si es dificil acceder a sus datos al momento de crear pruebas de unidad. Para probar un objeto a veces necesitas verificar si alguno de sus atributos esta en un estado particular. Para lo cual es necesario crear métodos públicos que permitan acceder a los datos del objeto.

Sin embargo, la desventaja de tener accesso público a los datos del objeto, es que si no se tiene cuidado podría crear más dependencias de otras clases a estos datos. Tener muchas depedencias entre clases (acoplamiento) podría traer problemas a futuro, ya que, por ejemplo, un defecto podría propagarse a lo largo de sus dependencias y sería difícil de entender, entre otro tipo de problemas.

Por lo que existen una disyuntiva entre observabilidad y encapsulamiento.

2.7 ¿Cuándo un método es difícil de testear?

Un ejemplo de un método dificil de testear es cuando por ejemplo se imprime un mensaje en consola, la pregunta seria: como verifico que se imprimio algo en consola?

```
class WalletV2:
  def __init__(self, owner):
      self.balance = 0
      self.owner_name = owner

  def deposit(self, amount):
      self.balance += amount

  def withdraw(self, amount):
    if amount < self.balance:
      self.balance -= amount
    else:
      print("not enough minerals")

  def total_balance(self):
      return self.balance
```

Alternativa 1: Una alternativa es agregar una bandera en el código fuente la cual nos permita verificar que el cuerpo del else fue fue ejecutado. En este ejemplo, agregamos una bandera y devolvemos su valor. Esto permite poder verificar el estado de la bandera desde la prueba de unidad.

```
class WalletV3:
  def __init__(self, owner):
      self.balance = 0
      self.owner_name = owner

  def deposit(self, amount):
      self.balance += amount

  def withdraw(self, amount):
    if amount < self.balance:
      self.balance -= amount
      return True
    else:
      print("not enough minerals")
      return False

  def total_balance(self):
      return self.balance

class WalletTestV3(unittest.TestCase):
  def test_withdraw(self):
    wallet = WalletV3("juampi")
    wallet.deposit(100)
    success= wallet.withdraw(110)
    self.assertFalse(success)

runTests(WalletTestV3)
```

```
.
```

```

```

```
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

```
not enough minerals
```

Alternativa 2: Otra alternativa es utilizar excepciones. En este caso desde la prueba unidad podemos verificar si la prueba de unidad lanzo una excepción o no. La ventaja de las excepciones, es que un mismo método pueden lanzar diferentes tipos de excepciones y desde la prueba de unidad podemos verificar que excepción lanzo. Otra ventaja es que no necesitamos utilizar return, esto es importante porque el método a testear podría ya devolver un valor util para la ejecución del programa.

```
class WalletV4:
  def __init__(self, owner):
      self.balance = 0
      self.owner_name = owner

  def deposit(self, amount):
      self.balance += amount

  def withdraw(self, amount):
    if amount < self.balance:
      self.balance -= amount
```

(continúe en la próxima página)

(proviene de la página anterior)

```
    else:
      raise Exception("not enough minerals")

  def total_balance(self):
      return self.balance

class WalletTestV4(unittest.TestCase):
  def test_withdraw(self):
    wallet = WalletV4("juampi")
    wallet.deposit(100)
    try:
      wallet.withdraw(110)
      self.assertFalse(True)
    except:
      self.assertFalse(False)

runTests(WalletTestV4)
```

```
.

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Alterantiva 2.1 El ejemplo anterior utilizar un try catch para ver si el código lanzo una excepción. La libreria unittest al igual que otras librerias permite agregar anotaciones en la cabecera del test. Por ejemplo, en este caso agregaremos una anotación que indique que al ejecutar la prueba de unidad esperamos que lanze una excepción. Existen varias configuraciones de anotaciones, por ejemplo, es posible especificar incluso que tipo de excepcion se espera entre otras cosas.

```
class WalletTestV5(unittest.TestCase):
  @unittest.expectedFailure
  def test_withdraw(self):
    wallet = WalletV4("juampi")
    wallet.deposit(100)
    wallet.withdraw(110)

runTests(WalletTestV5)
```

```
x

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK (expected failures=1)
```

2.8 Testeando Efectos Colaterales

Un efecto colateral en programación, o side effect, ocurre cuando una función o un bloque de código modifica algún estado fuera de su alcance local, más allá de devolver un valor. Estos cambios pueden incluir la alteración de variables globales, la modificación de los argumentos de entrada, la manipulación de estructuras de datos externas, o la interacción con el sistema (como escribir en un archivo o imprimir en la consola). Los efectos colaterales pueden dificultar la comprensión y el mantenimiento del código, ya que introducen dependencias y comportamientos no evidentes a simple vista, lo que puede llevar a errores inesperados y complicar la depuración. En muchos paradigmas de programación, especialmente en la programación funcional, se busca minimizar o eliminar los efectos colaterales para mejorar la predictibilidad y la claridad del código.

Crear pruebas de unidad para efectos colaterales es complicado, por ejemplo, analizemos como podriamos crear pruebas de unidad para el efecto colateral mas conocido, imprimir un valor en consola. Para esto agregaremos un método a la clase Wallet que imprime la información de la billetera en consola.

```
class WalletV5:
  def __init__(self, owner):
      self.balance = 0
      self.owner_name = owner

  def deposit(self, amount):
      self.balance += amount

  def withdraw(self, amount):
    if amount < self.balance:
      self.balance -= amount
    else:
      raise Exception("not enough minerals")

  def total_balance(self):
      return self.balance

  def printInfo(self):
    print(self.ownerName)
    print(str(self.balance))
```

El método anterior es un poco dificil de testear porque tendriamos que verificar si se imprimio lo esperado en consola. Podriamos utilizar algún metodo avanzado de python para recuperar lo que se escribio en consola pero en este ejemplo veremos otras alternativas que pueden ser utiles en otros casos.

Alternativa 1 Una alternativa es separar la lógica del string a imprimir de la impresion en si. Por ejemplo como se ve a continuación.

```
class WalletV6:
  def __init__(self, owner):
      self.balance = 0
      self.owner_name = owner

  def deposit(self, amount):
      self.balance += amount

  def withdraw(self, amount):
    if amount < self.balance:
      self.balance -= amount
    else:
      raise Exception("not enough minerals")
```

(continúe en la próxima página)

(proviene de la página anterior)

```
  def total_balance(self):
      return self.balance

  def printInfo(self):
    print(self.owner_name)

  def printString(self):
    return self.owner_name+" "+ str(self.balance)


class WalletTestV6(unittest.TestCase):
  def test_print(self):
    wallet = WalletV6("juampi")
    self.assertEqual(wallet.printString(),"juampi 0")


runTests(WalletTestV6)
```

```
.
```

```
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Al estar separado podemos testear si el string que imprime es el correcto. Ausmiendo que el método print funciona bien.

Alternativa 2 Podemos agregar un objeto que se encarge de imprimir en consola y que por defecto utilize la función print para imprimir. Este objeto basicamente seria un proxy que encapsula la función print.

```
class Printer:
  def print(self, string):
    print(string)

class WalletV7:
  def __init__(self, owner):
      self.balance = 0
      self.owner_name = owner

  def deposit(self, amount):
      self.balance += amount

  def withdraw(self, amount):
    if amount < self.balance:
      self.balance -= amount
    else:
      raise Exception("not enough minerals")

  def total_balance(self):
      return self.balance

  def printInfo(self, printer):
    printer.print(self.owner_name)
    printer.print(" ")
```

(continúe en la próxima página)

(proviene de la página anterior)

```
    printer.print(str(self.balance))
```

Si ejecutamos el código anterior deberiamos tener la misma funcionalidad. Sin embargo, ahora tenemos la funcionalidad imprimir desacoplada de la clase Wallet. Por lo que es posible aun verificar si se esta imprimimiendo «juampi 0», pero al estar desacoplada podemos enviar una impresora «falsa» que en lugar de imprimir en consola, guarde los strings a imprimir en una variable, variable que podemos verificar que tiene el valor correcto. Por ejemplo, de la siguiente forma:

```
class PrinterStub(Printer):
  def __init__(self):
    self.result = ""
  def print(self, string):
    self.result += string
  def printedString(self):
    return self.result

class WalletTestV7(unittest.TestCase):
  def test_print(self):
    stub = PrinterStub()
    wallet = WalletV7("juampi")
    wallet.printInfo(stub)
    self.assertEqual(stub.printedString(),"juampi 0")

runTests(WalletTestV7)
```

```
.
```

```
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

2.9 Test Smells

Los test, como cualquier código fuente, pueden estar sujetos a malos olores (smells). Los test smells son características no deseadas en las pruebas, ya que a la larga serán más difíciles de mantener o interpretar si un test falla.

Por ejemplo, un test no debe probar varias cosas al mismo tiempo, ya que cuando falle, será más difícil para el desarrollador entender la razón de la falla. Tampoco es recomendable tener dependencias externas, como archivos, ya que los tests se pueden ejecutar en diferentes computadoras por cada miembro del equipo y algunos pueden no tener el archivo, o tener una versión diferente del mismo.

Los **test smells** son señales de que una prueba puede ser difícil de mantener, comprender o poco confiable. A continuación, se presentan algunos test smells.

Assertion Roulette.

- **Definición:** El test contiene múltiples aserciones sin mensajes, lo que dificulta saber la razón de un falló.
- **Definición:** Al fallar, no se entiende qué condición específica no se cumplió.

```
def test_order_total():
    order = Order()
```

(continúe en la próxima página)

(proviene de la página anterior)

```
    order.add(Product("Book", 10))
    order.add(Product("Pen", 2))
    assert order.total() == 12
    assert order.item_count() == 2
    assert order.has_discount() is False
```

- **Definición:** Agregar mensajes explicativos o separar en múltiples tests.

Eager Test

- **Definición:** El test verifica múltiples funcionalidades distintas a la vez.
- **Definición:** Es difícil de mantener y depurar.

```
def test_order_process():
    order = Order()
    order.add(Product("Laptop", 1000))
    order.checkout()
    assert order.status == "PAID"
    assert order.shipping_status == "READY"
```

- **Definición:** Dividir en tests enfocados, como `test_order_payment` y `test_shipping_status`.

Lazy Test

- **Definición:** El test agrupa múltiples métodos pero verifica poco comportamiento.
- **Definición:** No captura errores específicos por método.

```
def test_user_behavior(self):
        user = User("alice", "admin")
        self.assertEqual(user.username, "alice")
        self.assertTrue(user.is_admin())
        self.assertTrue(user.is_active())
        user.deactivate()
        self.assertFalse(user.is_active())
```

- **Definición:** Escribir un test dedicado para cada comportamiento importante.

Mystery Guest

- **Definición:** El test depende de datos externos o fixtures ocultos.
- **Definición:** El test no es autocontenible ni claro.

```
def test_login():
    user = load_user_from_db("test_user")  # ¿de dónde viene este usuario?
    assert login(user.username, "secret") == True
```

- **Definición:** Usar mocks o fixtures definidos explícitamente en el test.

General Fixture

- **Definición:** Se inicializan objetos innecesarios en el setup.
- **Definición:** Se introduce complejidad y posibles dependencias ocultas.

```
def setUp(self):
    self.user = User("Alice")
    self.cart = Cart()
    self.order = Order(self.user, self.cart)
```

(continúe en la próxima página)

(proviene de la página anterior)

```
def test_cart_empty(self):
    assert self.cart.is_empty()
```

- **Definición:** Crear solo los objetos necesarios para cada test.

Test Run War

- **Definición:** Los tests fallan si se ejecutan en un orden distinto.
- **Definición:** Los resultados no son confiables.

```
# test_a cambia el estado que test_b necesita intacto

def test_a():
    with open("data.txt", "w") as f:
        f.write("changed")

def test_b():
    with open("data.txt") as f:
        assert f.read() == "original"
```

- **Solución:** Aislar los efectos y evitar estados compartidos entre tests.

Indirect Testing

- **Definición:** El test no prueba directamente la funcionalidad del componente bajo prueba, sino que depende de otros objetos para verificar el comportamiento.
- **Problema:** Si el test falla, no es claro cuál componente es el responsable.

```
def test_invoice_generation():
    order = Order()
    order.add(Product("Book", 10))
    invoice = Invoice(order)
    assert invoice.total() == 10  # indirectamente se está probando Order
```

- **Solución:** Asegurarse de probar directamente el componente relevante en tests separados.

Otros. Existe una lista extensa de test smells. Los más básicos y antiguos se pueden encontrar en la publicación de Ari Van Deursen. En la actualidad, hay más de 100 test smells identificados. Pueden encontrar una lista más o menos actualizada en el siguiente artículo:

https://www.sciencedirect.com/science/article/pii/S0164121217303060

2.10 Consejos para las Pruebas de Software

2.10.1 ¿Por qué es importante probar?

Muchos desarrolladores piensan que las pruebas son una pérdida de tiempo. Sin embargo, tener una suite de pruebas no solo ayuda a identificar errores actuales, sino también a mantener la calidad cuando el sistema evoluciona.

Las pruebas sirven para:

- Documentar el comportamiento esperado del sistema de forma activa.
- Verificar que cambios recientes no rompen otras funcionalidades.

- Ayudar a diseñar pensando en la funcionalidad deseada desde el punto de vista del usuario.

> **Consejo**
>
> Aunque no es posible probar todos los aspectos de una aplicación realista, una buena suite de pruebas ayuda a reducir significativamente los riesgos.

¿Qué hace que una prueba sea buena?

Escribir buenas pruebas es una habilidad que se aprende con la práctica. Algunas características clave incluyen:

- **Repetibilidad**: una prueba debe dar siempre el mismo resultado.
- **Automatización**: debe poder ejecutarse sin intervención humana.
- **Claridad**: cada prueba debe contar una historia clara de una funcionalidad específica.
- **Estabilidad**: una prueba debería cambiar menos frecuentemente que la funcionalidad que cubre.

Una buena estrategia es centrarse en probar las interfaces públicas de las clases y reducir las dependencias con detalles internos.

Consejos prácticos para diseñar pruebas

- **Pruebas autocontenidas**: deben ser independientes y no requerir configuración especial.
- **Evita la sobrecarga de pruebas**: múltiples pruebas del mismo aspecto pueden dificultar la comprensión y mantenimiento.
- **Separa las pruebas lentas**: pruebas que interactúan con bases de datos, red o sistema de archivos pueden ser valiosas, pero deberían distinguirse de las pruebas unitarias rápidas.

Regla de Black sobre pruebas

Cada prueba debe incrementar tu confianza en una propiedad concreta del sistema. No deberían existir pruebas redundantes que verifiquen lo mismo, ya que:

- Dificultan la lectura y comprensión del comportamiento esperado.
- Multiplican los errores reportados por un mismo fallo.

Reglas de Pharo sobre pruebas

La comunidad de desarrollo de Pharo promueve estas tres reglas fundamentales:

- **Una prueba que no está automatizada, no es una prueba.**
- **Todo lo que no está probado, no existe.**
- **Todo lo que no está probado, se romperá.**

Estas reglas enfatizan la necesidad de cubrir el sistema con pruebas automatizadas para garantizar su funcionamiento a largo plazo.