# Unit Testing

## IIC2143

*Prof. Juan Pablo Sandoval*

# *Contenido*

- ***Aspectos Básicos***
- *Estructura de un Unit Test*
- *Principios FIRST*
- *Unit Test y Encapsulamiento*
- *Testeando Efectos Colaterales*
- *Test Smells*

```python
class Wallet:
    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
```

***Class Under Test***

- ***Object State:***
- ***Object Invariant:***
- ***Getter:***
- ***Encapsulamiento:***

```python
class Wallet:
    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
```

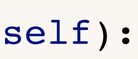
***Class Under Test***

- ***Object State:*** *Valores actuales de los atributos de un objeto.*
- ***Object Invariant:*** *Condición que verifica que un objeto esta en un estado correcto.*
- ***Getter:*** *Método que devuelve el valor de un atributo.*
- ***Encapsulamiento:*** *Grado de ocultamiento de un método/atributo (privado, protegido, publico)*

```python
class Wallet:
    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
```

***Class Under Test***

```python
import unittest

class WalletTest(unittest.TestCase):

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



***Unit Test Library***

***Test method***

***Test Class***

# *Contenido*

- *Aspectos Básicos*
- ***Estructura de un Unit Test***
- *Principios FIRST*
- *Unit Test y Encapsulamiento*
- *Testeando Efectos Colaterales*
- *Test Smells*

- ***Arrange/Setup:*** *Inicializa los objetos, valores, inputs, que se necesitan para realizar la prueba.*
- ***Act/Stimulus:*** *Ejecuta la función/ método/unidad a testear.*
- ***Assert/Verificación:*** *Verifica que el resultado sea el correcto. A veces también verifica que los objetos involucrados terminen en un estado esperado/correcto.*

```python
import unittest

class WalletTest(unittest.TestCase):

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

***Unit Test Library***

***A Unit test***

***Test Class***

```
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)
```

***Test Class***

- ***Suite:*** *Objeto que dentro tiene una colección de instancias de la test class (WalletTest). Si la clase tiene 3* ***test methods*** *tiene 3 instancias de la clase.*
- ***Runner:*** *Ejecuta cada test method utilizando una instancia diferente para cada uno, garantizando cierta independencia, que un test no afecte la ejecución de otro.*

```python
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)
```

- *Cuando los test tienen que realizar acciones similares al inicio o al final. Por ejemplo, crear una base de datos de prueba, y limpiarla al final de cada prueba. Es posible utilizar los métodos* ***setup*** *y* ***teardown****.*

```python
class WalletTest(unittest.TestCase):

    def setUp(self):
        self.wallet = Wallet("juampi")
        self.wallet.deposit(100)

    def tearDown(self):
       pass

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

- *Este ejemplo sobre-escribe los métodos de la clase Test case (**setUp** y **tearDown**)*
- *Se refactoring el ejemplo anterior subiendo el **set up** común que tenián ambos test al método setUp.*
- *El método **tearDown** esta vacío porque no tenemos que ejecutar instrucciones similares al final de cada test. Es posible incluso borrarlo ya que la clase padre actualmente ya define uno vacío.*

# *Contenido*

- *Aspectos Básicos*
- *Estructura de un Unit Test*
- ***Principios FIRST***
- *Unit Test y Encapsulamiento*
- *Testeando Efectos Colaterales*
- *Test Smells*

# *FIRST*

***Fast*** *— Quick execution for rapid feedback.*

***Independent*** *— No test should depend on another.*

***Repeatable*** *— The same results every time, no flakiness.*

***Self-Validating*** *— Clear pass/fail results.*

***Timely*** *— Written before production code.*

# *Contenido*

- *Aspectos Básicos*
- *Estructura de un Unit Test*
- *Principios FIRST*
- ***Unit Test y Encapsulamiento***
- *Testeando Efectos Colaterales*
- *Test Smells*

```python
class WalletTest(unittest.TestCase):

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

**Field Access**

- *Para verificar si el estado final de un objeto es el esperado normalmente necesitamos acceder a la información de sus atributos, por lo que necesitamos que sea publico o que tenga un getter.*
- *Sin embargo, para reducir el acoplamiento necesitamos ocultar la toda información posible. Por eso se invento el encapsulamiento en primer lugar.*
- *Que dilema*

# *Contenido*

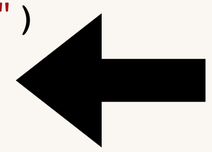
- *Aspectos Básicos*
- *Estructura de un Unit Test*
- *Principios FIRST*
- *Unit Test y Encapsulamiento*
- ***Testeando Efectos Colaterales***
- *Test Smells*

```python
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
```

- ***Como creo un unit test para el scenario que no alcance el dinero?***

# Alternativa 1: Flag

```
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
    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)
```

- *Aprovecho el valor de retorno para dar mas información sobre la operación.*

# *Alternativa 2: Exceptions*

```
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
    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)
```

- *Utilizo Excepciones como el buen programador que soy.*
- *La ventaja es que puedo lanzar diferentes tipos de excepciones y ademas retornar un valor en una misma función.*

# *Alternativa 2.1: Exceptions*

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

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

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

- *Utilizo anotaciones asi mi código de test es mas elegante y legible.*

```python
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))
```

- ***Como pruebo el método print?***

# *Alternativa 1: Separo la Lógica*

```
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")
  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")
```

# *Alternativa 2.1: Usando un Stub*

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

class WalletV7:
  …

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

```
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")
```

# *Alternativa 2.1: Usando un Mocks*

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

class WalletV7:
  …
  def total_balance(self):
      return self.balance

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

```python
class WalletTestV7(unittest.TestCase):
    def test_print_with_mock(self):
        # Creamos un mock que simula un Printer
        mock_printer = MagicMock()
        # Creamos el wallet
        wallet = WalletV7("juampi")
        # Llamamos al método que usa el printer
        wallet.printInfo(mock_printer)
        # Verificamos que se llamó a print 3 veces
        self.assertEqual(mock_printer.print.call_count, 3)
        # Verificamos que se llamó con los valores
        # correctos en orden
        mock_printer.print.assert_has_calls([
            unittest.mock.call("juampi"),
            unittest.mock.call(" "),
            unittest.mock.call("0")
        ])
```

# *Lectura Obligatoria: Stubs vs Mocks*

https://www.turing.com/kb/stub-vs-mock

# *Contenido*

- *Aspectos Básicos*
- *Estructura de un Unit Test*
- *Principios FIRST*
- *Unit Test y Encapsulamiento*
- *Testeando Efectos Colaterales*
- ***Test Smells***

# *Lectura Complementaria y Obligatoria*

## ***Apuntes de Clase***

*Unit Testing*
*Mocking Objects*