

Grupo: Rodriguez del Busto, Sofía
Naessens, Maia Guadalupe
Ason, Belén

Pruebas del software: uso de un framework de testing

Trabajo Práctico 1 - 29 de Agosto del 2025

Ingeniería de Software 1 (PU) - Ingeniería de Software 2 (LI / II)

Este trabajo práctico sirve como guía básica sobre las pruebas unitarias y de integración hasta escenarios de prueba más complejos, aumentando gradualmente la complejidad. Puedes usar un lenguaje de programación y framework a elección.

Problema: “Sistema de Gestión de Productos en una Tienda”

Vas a desarrollar y probar un sistema básico de gestión de productos en una tienda, utilizando dos clases principales: `Producto` y `Tienda` dadas a continuación. La clase `Tienda` gestionará un inventario de productos, permitiendo agregar, eliminar, buscar productos, aplicar descuentos, y calcular el precio total de un carrito de compras.

Organización del Proyecto (ejemplo python):

- **Producto** (`producto.py`): Clase que representa un producto en la tienda.
- **Tienda** (`tienda.py`): Clase que maneja el inventario y las operaciones de la tienda.
- **Tests** (`test_tienda.py` `test_producto.py`): Archivos donde se implementarán las pruebas utilizando el framework.

Código base en python:

```
class Producto:
    def __init__(self, nombre, precio, categoria):
        self.nombre = nombre
        self.precio = precio
        self.categoria = categoria
```

```
from producto import Producto

class Tienda:
    def __init__(self):
        self.inventario = []

    def agregar_producto(self, producto):
        self.inventario.append(producto)

    def buscar_producto(self, nombre):
        for producto in self.inventario:
            if producto.nombre == nombre:
                return producto
        return None

    def eliminar_producto(self, nombre):
        for producto in self.inventario:
            if producto.nombre == nombre:
                self.inventario.remove(producto)
                return True
        return False
```

Grupo: Rodriguez del Busto, Sofía
Naessens, Maia Guadalupe
Ason, Belén

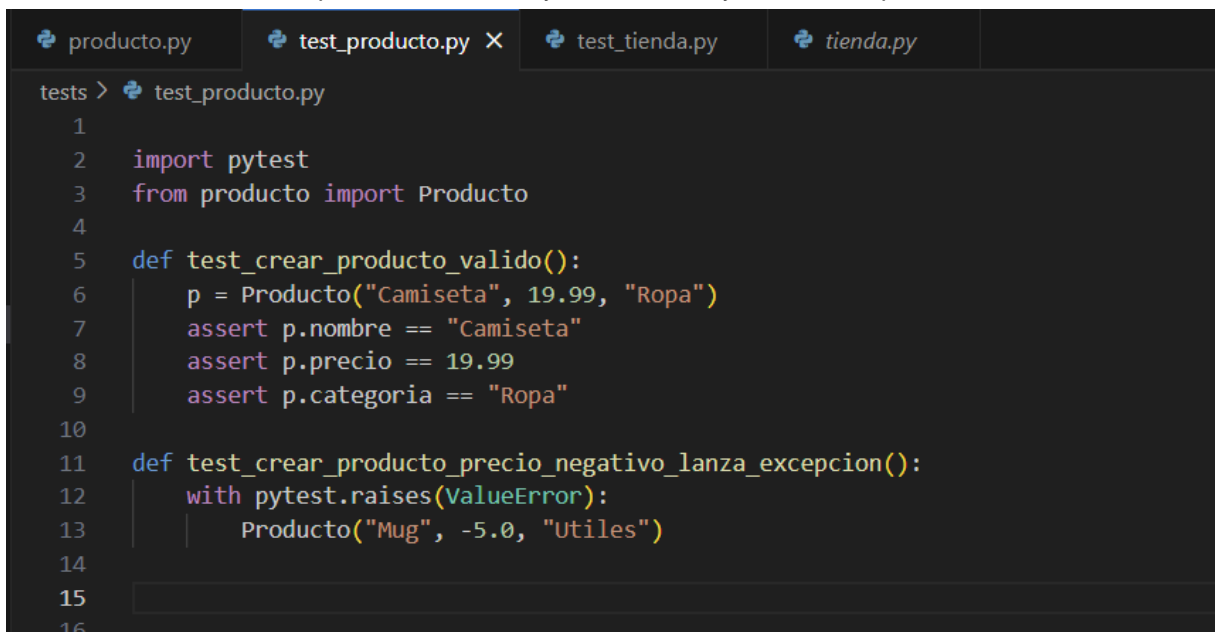
1) Configuración Inicial y Pruebas Básicas

Objetivo: Aprender a configurar un entorno de pruebas con el framework elegido y a realizar pruebas básicas.

1. Usa la implementación base de la clase `Producto` con atributos como `nombre`, `precio`, y `categoria`.
2. Usa la implementación base la clase `Tienda` con métodos para:
 - a. Agregar un producto al inventario.
 - b. Buscar un producto por su nombre.
 - c. Eliminar un producto del inventario.

Actividad práctica:

- Escribe pruebas básicas para estas clases y métodos utilizando el framework elegido.
- Asegúrate de cubrir casos simples, como agregar un producto y verificar que esté en la tienda, buscar un producto existente y no existente, y eliminar un producto.



```
producto.py  test_producto.py X  test_tienda.py  tienda.py

tests > test_producto.py
1
2  import pytest
3  from producto import Producto
4
5  def test_crear_producto_valido():
6      p = Producto("Camiseta", 19.99, "Ropa")
7      assert p.nombre == "Camiseta"
8      assert p.precio == 19.99
9      assert p.categoria == "Ropa"
10
11  def test_crear_producto_precio_negativo_lanza_excepcion():
12      with pytest.raises(ValueError):
13          Producto("Mug", -5.0, "Utiles")
14
15
16
```

Grupo: Rodriguez del Busto, Sofía
Naessens, Maia Guadalupe
Ason, Belén

```
product.py  test_producto.py  test_tienda.py  tienda.py
tests > test_tienda.py
2  import pytest
3  from tienda import Tienda
4  from producto import Producto
5
6  def test_agregar_producto_y_verificar_en_tienda():
7      tienda = Tienda()
8      p = Producto("Laptop", 999.99, "Tecnología")
9      tienda.agregar_producto(p)
10     assert any(prod.nombre == "Laptop" for prod in tienda.inventario)
11
12 def test_buscar_producto_existente():
13     tienda = Tienda()
14     p = Producto("Teclado", 29.99, "Tecnología")
15     tienda.agregar_producto(p)
16     encontrado = tienda.buscar_producto("Teclado")
17     assert encontrado is not None
18     assert isinstance(encontrado, Producto)
19     assert encontrado.nombre == "Teclado"
20
21 def test_buscar_producto_no_existente():
22     tienda = Tienda()
23     encontrado = tienda.buscar_producto("Ratón Inalámbrico")
24     assert encontrado is None
25
26 def test_eliminar_producto_existente():
27     tienda = Tienda()
28     p = Producto("Monitor", 199.99, "Tecnología")
29     tienda.agregar_producto(p)
30     eliminado = tienda.eliminar_producto("Monitor")
31     assert eliminado is True
32     assert tienda.buscar_producto("Monitor") is None
33
34 def test_eliminar_producto_no_existente():
35     tienda = Tienda()
36     eliminado = tienda.eliminar_producto("Teclado Mecánico")
37     assert eliminado is False
38
```

```
(.venv) D:\SofiaRB\Documents\Facultad\4to Año\IngenieriaDeSoftwareII\TP1\tiendapy>set PYTHONPATH=.
(.venv) D:\SofiaRB\Documents\Facultad\4to Año\IngenieriaDeSoftwareII\TP1\tiendapy>pytest -v
===== test session starts =====
platform win32 -- Python 3.13.7, pytest-8.4.1, pluggy-1.6.0 -- D:\SofiaRB\Documents\Facultad\4to Año\IngenieriaDeSoftwareII\TP1\tiendapy\venv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: D:\SofiaRB\Documents\Facultad\4to Año\IngenieriaDeSoftwareII\TP1\tiendapy
collected 7 items

tests/test_producto.py::test_crear_producto_valido PASSED [ 14%]
tests/test_producto.py::test_crear_producto_precio_negativo_lanza_excepcion PASSED [ 28%]
tests/test_tienda.py::test_agregar_producto_y_verificar_en_tienda PASSED [ 42%]
tests/test_tienda.py::test_buscar_producto_existente PASSED [ 57%]
tests/test_tienda.py::test_buscar_producto_no_existente PASSED [ 71%]
tests/test_tienda.py::test_eliminar_producto_existente PASSED [ 85%]
tests/test_tienda.py::test_eliminar_producto_no_existente PASSED [100%]

===== 7 passed in 0.04s =====
```

Preguntas conceptuales:

- ¿Puedes identificar pruebas de unidad y de integración en la práctica que se realizó?

En el archivo test_producto.py se implementaron únicamente pruebas de unidad. En ambos casos, se prueba el constructor de la clase producto el cual no depende de otros módulos del proyecto. En el primer caso, se verifica que el constructor funcione correctamente y en el segundo caso, que lance la excepción correspondiente cuando el precio es negativo.

En el archivo test_tienda.py, test_agregar_producto_y_verificar_en_tienda, test_buscar_producto_existente y test_eliminar_producto_existente hacen uso de los constructores de la clase producto por lo que implica una prueba de integración. En los métodos restantes, se prueban los métodos de la clase tienda sin dependencia de otros módulos.

2) Pruebas con Excepciones

Objetivo: Ampliar las pruebas para manejar y verificar el uso adecuado de excepciones.

Grupo: Rodriguez del Busto, Sofía
Naessens, Maia Guadalupe
Ason, Belén

1. Modifica el método `buscar_producto` para que lance una excepción si no se encuentra un producto.
2. Modifica el método `eliminar_producto` para que lance una excepción si se intenta eliminar un producto que no existe.
3. Implementa un método `actualizar_precio` en la clase `Producto` que permita cambiar el precio de un producto y lance una excepción si el nuevo precio es negativo.

```
product.py X test_producto.py test_tienda.py tienda.py
product.py
1 class Producto:
2     def __init__(self, nombre, precio, categoria):
3         if precio < 0:
4             raise ValueError("El precio no puede ser negativo")
5         self.nombre = nombre
6         self.precio = precio
7         self.categoria = categoria
8
9     def actualizar_precio(self, nuevo_precio: float) -> None:
10        if nuevo_precio < 0:
11            raise ValueError("El nuevo precio no puede ser negativo")
12        self.precio = nuevo_precio
13
```

```
product.py X test_producto.py test_tienda.py tienda.py X
tienda.py
1 from producto import Producto
2 class ProductoNoEncontradoError(Exception):
3     pass
4
5 class Tienda:
6     def __init__(self):
7         self.inventario = []
8     def agregar_producto(self, producto):
9         self.inventario.append(producto)
10    def buscar_producto(self, nombre):
11        for producto in self.inventario:
12            if producto.nombre == nombre:
13                return producto
14        raise ProductoNoEncontradoError(f"Producto '{nombre}' no encontrado.")
15        #return None
16    def eliminar_producto(self, nombre):
17        for producto in self.inventario:
18            if producto.nombre == nombre:
19                self.inventario.remove(producto)
20                return True
21        raise ProductoNoEncontradoError(f"No se puede eliminar '{nombre}': producto no existe.")
22        #return False
```

Actividad práctica:

- Escribe pruebas que verifiquen que se lanzan las excepciones correctas al intentar eliminar un producto inexistente o actualizar un precio a un valor negativo.
- Utiliza el framework para comprobar que las excepciones se lanzan correctamente.

Grupo: Rodriguez del Busto, Sofía
Naessens, Maia Guadalupe
Ason, Belén

```
producto.py test_producto.py X test_tienda.py tienda.py
tests > test_producto.py
1
2 import pytest
3 from producto import Producto
4
5 def test_crear_producto_valido():
6     p = Producto("Camiseta", 19.99, "Ropa")
7     assert p.nombre == "Camiseta"
8     assert p.precio == 19.99
9     assert p.categoria == "Ropa"
10
11 def test_crear_producto_precio_negativo_lanza_excepcion():
12     with pytest.raises(ValueError):
13         Producto("Mug", -5.0, "Utiles")
14
15 def test_actualizar_precio_valido():
16     p = Producto("Zapatillas", 50.0, "Calzado")
17     p.actualizar_precio(45.0)
18     assert p.precio == 45.0
19
20 def test_actualizar_precio_negativo_lanza_excepcion():
21     p = Producto("Gorra", 10.0, "Accesorios")
22     with pytest.raises(ValueError):
23         p.actualizar_precio(-1.0)
24
```

```
producto.py test_producto.py test_tienda.py X tienda.py
tests > test_tienda.py
1
2 import pytest
3 from tienda import Tienda, ProductoNoEncontradoError
4 from producto import Producto
5
6 def test_agregar_producto_y_verificar_en_tienda():
7     tienda = Tienda()
8     p = Producto("Laptop", 999.99, "Tecnología")
9     tienda.agregar_producto(p)
10    assert any(prod.nombre == "Laptop" for prod in tienda.inventario)
11
12 def test_buscar_producto_existente():
13     tienda = Tienda()
14     p = Producto("Teclado", 29.99, "Tecnología")
15     tienda.agregar_producto(p)
16     encontrado = tienda.buscar_producto("Teclado")
17     assert encontrado is not None
18     assert isinstance(encontrado, Producto)
19     assert encontrado.nombre == "Teclado"
20
21 def test_buscar_producto_no_existente():
22     tienda = Tienda()
23     with pytest.raises(ProductoNoEncontradoError):
24         tienda.buscar_producto("Ratón Inalámbrico")
25     #assert encontrado is None
26
27 def test_eliminar_producto_existente():
28     tienda = Tienda()
29     p = Producto("Monitor", 199.99, "Tecnología")
30     tienda.agregar_producto(p)
31     eliminado = tienda.eliminar_producto("Monitor")
32     assert eliminado is True
33     with pytest.raises(ProductoNoEncontradoError):
34         tienda.buscar_producto("Monitor")
35     #assert tienda.buscar_producto("Monitor") is None
36
37 def test_eliminar_producto_no_existente():
38     tienda = Tienda()
39     with pytest.raises(ProductoNoEncontradoError):
40         tienda.eliminar_producto("Teclado Mecánico")
41     #assert eliminado is False
42
```

Grupo: Rodriguez del Busto, Sofía
Naessens, Maia Guadalupe
Ason, Belén

```
(.venv) D:\SofiaRB\Documents\Facultad\4to Año\IngenieriaDeSoftwareII\TP1\tiendapy>set PYTHONPATH=.

(.venv) D:\SofiaRB\Documents\Facultad\4to Año\IngenieriaDeSoftwareII\TP1\tiendapy>pytest -v
===== test session starts =====
platform win32 -- Python 3.13.7, pytest-8.4.1, pluggy-1.6.0 -- D:\SofiaRB\Documents\Facultad\4to Año\IngenieriaDeSoftwareII\TP1\tiendapy\.venv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: D:\SofiaRB\Documents\Facultad\4to Año\IngenieriaDeSoftwareII\TP1\tiendapy
collected 9 items

tests/test_producto.py::test_crear_producto_valido PASSED [ 11%]
tests/test_producto.py::test_crear_producto_precio_negativo_lanza_excepcion PASSED [ 22%]
tests/test_producto.py::test_actualizar_precio_valido PASSED [ 33%]
tests/test_producto.py::test_actualizar_precio_negativo_lanza_excepcion PASSED [ 44%]
tests/test_tienda.py::test_agregar_producto_y_verificar_en_tienda PASSED [ 55%]
tests/test_tienda.py::test_buscar_producto_existente PASSED [ 66%]
tests/test_tienda.py::test_buscar_producto_no_existente PASSED [ 77%]
tests/test_tienda.py::test_eliminar_producto_existente PASSED [ 88%]
tests/test_tienda.py::test_eliminar_producto_no_existente PASSED [100%]

===== 9 passed in 0.05s =====
```

Preguntas conceptuales:

- Podría haber escrito las pruebas primero antes de modificar el código de la aplicación?
¿Cómo sería el proceso de escribir primero los tests? Describe el proceso con tus palabras.

Si, podrían escribirse los tests antes del código lo que se conoce como Test Driven Development (TDD). En ese caso, debería analizar el comportamiento o requisitos esperados del sistema. Por ejemplo, si los productos deben contener un nombre y precio, y éste último no debe ser negativo, entonces debería primero crear un test que construya un objeto producto con un precio negativo y verificar que el código lance una excepción y otro en el que se cree un producto exitosamente. Una vez creado el test, se debe escribir el código de manera que pase el test. Siguiendo el ejemplo, debería implementar el constructor que realice el control del precio y que en caso de que sea negativo, lance una excepción del tipo especificado en el test. Caso contrario, se creará el objeto. Por último, si al ejecutar el test éste no falla el código cumple con el comportamiento especificado. En caso de que no lo pase satisfactoriamente, se deberá reformular el código.

3) Uso de Mocks para aislar unidades bajo prueba

Objetivo: Introducir el uso de “test doubles” para aislar unidades de prueba y evitar dependencias innecesarias durante la prueba. En las pruebas de software, los dobles de prueba son objetos o componentes que reemplazan componentes reales para aislar la unidad bajo prueba.

1. Implementa un método `aplicar_descuento` en la clase `Tienda` que acepte un nombre de producto y un porcentaje de descuento y que modifique el precio del producto.
2. Utiliza un “test doble” para simular el comportamiento del producto.

Actividad práctica:

- Escriba pruebas utilizando mocks para verificar que el método `aplicar_descuento` calcula correctamente el nuevo precio sin usar un objeto real.

Grupo: Rodriguez del Busto, Sofía
Naessens, Maia Guadalupe
Ason, Belén

- Asegúrate de verificar que la función llama correctamente al método `actualizar_precio` del producto.
- [Opcional] Utilizar dobles de prueba para los tests de agregar, buscar y eliminar productos.

```
def aplicar_descuento(self, nombre: str, porcentaje: float):
    if not (0 <= porcentaje <= 100):
        raise ValueError("El porcentaje de descuento debe estar entre 0 y 100.")
    producto = self.buscar_producto(nombre)
    descuento = producto.precio * (porcentaje / 100)
    producto.actualizar_precio(producto.precio - descuento)
```

```
def test_aplicar_descuento_valido():
    tienda = Tienda()

    #Crea un mock de producto
    producto_mock = Mock()
    producto_mock.nombre = "Laptop"
    producto_mock.precio = 1000.0
    producto_mock.categoria = "Tecnología"

    #Crea un mock del método para no probar la implementación real del método sino que registra llamadas
    producto_mock.actualizar_precio = Mock()

    tienda.inventario.append(producto_mock)

    tienda.aplicar_descuento("Laptop", 10)

    #Verifico que el método llame correctamente a actualizar_precio de producto con el argumento correspondiente.
    producto_mock.actualizar_precio.assert_called_once_with(900.0)
```

Preguntas conceptuales:

- En lo que va del trabajo práctico, ¿puedes identificar 'Controladores' y 'Resguardos'?
Los mocks para simular un producto y sus métodos funcionan como resguardos para poder llevar a cabo la prueba de unidad de tienda ya que nos permite hacer una prueba de unidad de la clase tienda sin tener que implementar la dependencia del módulo Producto.
Los métodos como `test_buscar_producto_existente` o `test_eliminar_producto_existente` funcionan como controladores para los métodos `buscar_producto` y `eliminar_producto` de la clase tienda respectivamente ya que genera las condiciones y objetos necesarios para poder utilizar dichas funcionalidades de la clase tienda y luego las invoca para poder testearlas.
- ¿Qué es un "test double"? ¿Hay otros nombres para los objetos/funciones simulados?
Un test double es cualquier objeto u componente utilizado para reemplazar otro objeto o componente real de nuestro sistema para poder llevar a cabo una prueba, permitiendo aislar una unidad y centrarse en su lógica. Dependiendo el propósito con el que se lo utiliza, un test double puede tomar una de las siguientes formas:
 - Dummy object: es un contenedor de un objeto que se le pasa al sistema que se está testeando (SUT, System Under Testing) como argumento pero que nunca se utiliza realmente. solo existen para completar el espacio en los argumentos de un método o constructor.
 - Test stub: es un objeto que reemplaza a un componente real del cual depende el sistema bajo prueba de manera que el test pueda controlar las entradas indirectas del mismo. Proporciona entradas indirectas predeterminadas que permiten controlar la ruta de ejecución del SUT para que el código pueda seguir un flujo de ejecución que de otra manera sería difícil de alcanzar.
 - Test spy: es una versión mas avanzada de un test stub, que no solo proporciona entradas indirectas al SUT sino que registra llamadas que se le hacen, incluyendo los argumentos que se pasaron. Esto permite controlar al final de la ejecución que las salidas indirectas del SUT son las correctas.
 - Mock Objects: un mock funciona como un punto de observación para las salidas indirectas del sistema como un spy. Puede necesitar devolver información de las llamadas a sus métodos como un stub y también, brinda información sobre como fueron invocados como un spy. Lo que diferencia a un mock object es que compara las llamadas reales recibidas con las expectativas predefinidas usando aserciones y hace que la prueba falle en nombre del método de prueba.
 - Fake Objects: un fake es un objeto que reemplaza una dependencia real con una implementación simplificada y funcional de la misma. Suelen utilizarse cuando la dependencia real es demasiado lenta o no está disponible, como una base de datos en

Grupo: Rodriguez del Busto, Sofía
Naessens, Maia Guadalupe
Ason, Belén

memoria que se usa en lugar de una base de datos real.

FUENTE: Meszaros, G. (2007). xUnit Test Patterns: Refactoring Test Code. Addison-Wesley Professional.

4) Uso de Fixtures para Reutilización de Datos

Objetivo: Aprender a usar “fixtures” para preparar y limpiar el entorno de pruebas.

1. Crea un fixture que inicialice una tienda con algunos productos de ejemplo antes de cada prueba.
2. Modifica las pruebas existentes para usar este fixture en lugar de crear los productos manualmente en cada prueba.

Actividad práctica:

- Implementa un fixture y utilízalo en las pruebas de `agregar_producto` y `buscar_producto`.
- Verifica que las pruebas se ejecutan correctamente utilizando los productos predefinidos.

```
@pytest.fixture
def tienda_con_productos_fixture():
    """Un fixture que configura una instancia de Tienda con productos para la prueba."""
    tienda = Tienda()
    tienda.agregar_producto(Producto("Placa de video", 999.99, "Tecnología"))
    tienda.agregar_producto(Producto("Teclado", 29.99, "Tecnología"))
    return tienda

def test_agregar_producto_y_verificar_en_tienda_fix(tienda_con_productos_fixture):
    p = Producto("Laptop", 999.99, "Tecnología")
    tienda_con_productos_fixture.agregar_producto(p)
    assert any(prod.nombre == "Laptop" for prod in tienda_con_productos_fixture.inventario)

def test_buscar_producto_existente_fix(tienda_con_productos_fixture):
    encontrado = tienda_con_productos_fixture.buscar_producto("Teclado")
    assert encontrado is not None
    assert isinstance(encontrado, Producto)
    assert encontrado.nombre == "Teclado"

def test_buscar_producto_no_existente_fix(tienda_con_productos_fixture):
    with pytest.raises(ProductoNoEncontradoError):
        tienda_con_productos_fixture.buscar_producto("Ratón Inalámbrico")
```

Preguntas conceptuales:

- ¿Qué ventajas ve en el uso de fixtures? ¿Qué enfoque estaríamos aplicando (caja negra/blanca)?

El uso de fixtures nos permite preparar de una manera más optimizada nuestro entorno de prueba. El fixture nos permite reutilizar código ya que por ejemplo, en este caso, nos permite inicializar una tienda con productos para poder probar sus métodos evitando que en cada uno de los tests debamos crear los objetos necesarios para su realización y repetir código. A su vez, esto nos permite que nuestras pruebas sean más legibles y mantener la consistencia entre ellas.

No considero que el fixture en general tenga un único enfoque. Puede ser utilizado tanto en un enfoque de caja blanca inicializando objetos con determinadas características para probar flujos de control específico. En nuestro caso en particular estamos llevando a cabo este tipo de enfoque. Sin embargo, podría existir otro caso en el que el fixture que establezca un estado externo como un script que carga información que provendría de una base de datos. Esto representaría un enfoque de

Grupo: Rodriguez del Busto, Sofía
Naessens, Maia Guadalupe
Ason, Belén
caja negra.

- Explique los conceptos de Setup y Teardown en testing.

El concepto de Setup en testing implica la creación e inicialización del entorno de prueba de manera que se establezcan todas las condiciones previas necesarias para llevar a cabo las pruebas del código. Esto puede implicar desde crear e inicializar objetos, abrir conexiones de bases de datos, entre otros.

El concepto de Teardown se refiere a una fase posterior a la prueba donde se busca deshacer los efectos del setup como cerrar las conexiones a bases de datos, borrar archivos temporales, etc. El objetivo es que se limpie el entorno de prueba para que los efectos de una prueba no tengan impacto en la siguiente.

5) Pruebas de Integración y Cobertura Completa

Objetivo: Realizar pruebas más complejas que verifiquen la integración de varias funciones y un flujo completo del sistema.

1. Implementa un método `calcular_total_carrito` en la clase `Tienda` que reciba una lista de nombres de productos (el carrito) y devuelva el precio total.
2. Integra todas las funciones anteriores para permitir agregar productos a un carrito y calcular el total.

Actividad práctica:

- Escribe pruebas de integración que verifiquen el flujo completo para calcular el total del carrito después de aplicar los descuentos.
- Utiliza un fixture para inicializar una tienda con varios productos.
- Verifica que el precio total calculado sea correcto.

```
def calcular_total_carrito(self, carrito: list[str]) -> float:
    total = 0.0
    for nombre_producto in carrito:
        try:
            producto = self.buscar_producto(nombre_producto)
            total += producto.precio
        except ProductoNoEncontradoError:
            print(f"Advertencia: El producto '{nombre_producto}' no se encontró en el inventario y no se incluyó en el total.")
            continue
    return total
```

```
def test_flujo_completo_calcular_total_con_descuento(tienda_con_productos_fixture):
    """
    Prueba de integración: verifica el flujo completo de aplicar un descuento
    y luego calcular correctamente el total del carrito.
    """
    tienda = tienda_con_productos_fixture

    nombre_producto = "Monitor"
    descuento_porcentaje = 10
    tienda.aplicar_descuento(nombre_producto, descuento_porcentaje)

    carrito = ["Placa de video", "Teclado", "Monitor"]

    total_final = tienda.calcular_total_carrito(carrito)

    precio_placa_video = 999.99
    precio_teclado = 29.99
    precio_monitor_con_descuento = 199.99 - 199.99 * (descuento_porcentaje / 100)

    total_esperado = precio_placa_video + precio_teclado + precio_monitor_con_descuento
    assert total_final == pytest.approx(total_esperado)

    print(f"Total calculado: {total_final}")
    print(f"Total esperado: {total_esperado}")
```

Grupo: Rodriguez del Busto, Sofía
Naessens, Maia Guadalupe
Ason, Belén

```
PS D:\SofiaRB\Documents\Facultad\4to Año\IngenieriaDeSoftwareII\TP1\tiendapy> .venv\Scripts\python.exe -m pytest -v
===== test session starts =====
platform win32 -- Python 3.13.7, pytest-8.4.1, pluggy-1.6.0 -- D:\SofiaRB\Documents\Facultad\4to Año\IngenieriaDeSoftwareII\TP1\tiendapy\venv\Scripts\python.exe
cachedir: .pytest_cache
rootdir: D:\SofiaRB\Documents\Facultad\4to Año\IngenieriaDeSoftwareII\TP1\tiendapy
collected 15 items

tests/test_producto.py::test_crear_producto_valido PASSED [ 6%]
tests/test_producto.py::test_crear_producto_precio_negativo_lanza_excepcion PASSED [ 13%]
tests/test_producto.py::test_actualizar_precio_valido PASSED [ 20%]
tests/test_producto.py::test_actualizar_precio_negativo_lanza_excepcion PASSED [ 26%]
tests/test_producto.py::test_agregar_producto_y_verificar_en_tienda PASSED [ 33%]
tests/test_producto.py::test_eliminar_producto_existente PASSED [ 40%]
tests/test_producto.py::test_eliminar_producto_no_existente PASSED [ 46%]
tests/test_producto.py::test_aplicar_descuento_valido PASSED [ 53%]
tests/test_producto.py::test_aplicar_descuento_invalido PASSED [ 60%]
tests/test_producto.py::test_agregar_producto_y_verificar_en_tienda_fix PASSED [ 66%]
tests/test_producto.py::test_eliminar_producto_existente_fix PASSED [ 73%]
tests/test_producto.py::test_eliminar_producto_no_existente_fix PASSED [ 80%]
tests/test_producto.py::test_agregar_producto_y_verificar_en_tienda_fix PASSED [ 86%]
tests/test_producto.py::test_eliminar_producto_existente_fix PASSED [ 93%]
tests/test_producto.py::test_eliminar_producto_no_existente_fix PASSED [100%]

===== 15 passed in 0.11s =====
PS D:\SofiaRB\Documents\Facultad\4to Año\IngenieriaDeSoftwareII\TP1\tiendapy> |
```

Preguntas conceptuales:

- ¿Puede describir una situación de desarrollo para este caso en donde se plantee pruebas de integración ascendente? Describa la situación.

En nuestro sistema de tienda se pueden observar pruebas de integración ascendente probando primero la clase producto con todos sus métodos ya que representa una clase que no depende de otra y de la cual si depende nuestra clase tienda.

Una vez probada la clase producto, se prueban en la clase tienda todos los métodos que operan con productos como agregar_producto, buscar_producto y eliminar_producto.

Finalmente, se prueban los métodos que invocan a los métodos base de tienda mencionados anteriormente como el método calcular_carrito.