POO · CEPY · UT3

Tests para Repositorios SQLite

Aislamiento, setUp/tearDown y excepciones de dominio

Persistencia con bases de datos

Índice

1. **Aislamiento en los tests de base de datos**
2. **El ciclo `setUp` y `tearDown`**
3. **Testar las operaciones del repositorio**
4. **Testar excepciones de dominio**

1. Aislamiento en los tests de base de datos

Por qué cada test necesita su propia base de datos limpia

El problema: tests que comparten BD

Un **test de repositorio** lee y escribe en un fichero `.db` real — no en memoria.

🚫 **Tests acoplados por datos compartidos**

Si el test A guarda un producto, el test B lo verá.

El resultado depende del orden de ejecución → tests frágiles e impredecibles.

✅ **La solución**

Cada test recibe una **base de datos nueva y vacía**, creada antes de empezar y eliminada al terminar.

BD_TEST — la ruta del fichero de test

`pathlib.Path` gestiona la ruta y proporciona `.exists()` y `.unlink()`.

```
from pathlib import Path
import unittest

class TestRepositorioSQLite(unittest.TestCase):

    BD_TEST = Path("test_expendedora.db")
```

💡 **`BD_TEST` es un atributo de clase** — todas las instancias comparten la misma ruta, pero cada test crea y elimina el fichero de nuevo.

Usa un nombre distinto al de la BD de producción (`expendedora.db`) para que los tests nunca modifiquen datos reales.

2. El ciclo `setUp` y `tearDown`

Crear y destruir la base de datos antes y después de cada test

`setUp` — preparar antes de cada test

`setUp` se ejecuta automáticamente **antes de cada método de test**.

Responsabilidad	Acción
Limpiar estado previo	`BD_TEST.unlink()` si existe
Crear el esquema	`CREATE TABLE ...` vía `sqlite3`
Instanciar el repositorio	`self.repo = RepositorioProductosSQLite( ... )`

💡 El `unlink()` inicial garantiza un estado limpio aunque el proceso anterior se interrumpiera sin llegar a `tearDown`.

`setUp` — estructura del código

```
def setUp(self):
    if self.BD_TEST.exists():
        self.BD_TEST.unlink()    # estado limpio de partida

    conn = sqlite3.connect(self.BD_TEST)
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE ...")   # crear el esquema necesario
    conn.commit()
    conn.close()

    self.repo = MiRepositorio(self.BD_TEST)
```

`sqlite3.connect()` acepta objetos `Path` directamente — no hace falta convertir a `str`.

`tearDown` — limpiar después del test

`tearDown` se ejecuta **después de cada test**, tanto si pasa como si falla.

```
def tearDown(self):
    if self.BD_TEST.exists():
        self.BD_TEST.unlink()
```

💡 **Python 3.8+** permite omitir la comprobación previa:

```
self.BD_TEST.unlink(missing_ok=True)
```

`setUp` crea — `tearDown` destruye. Cada test recibe una BD vacía y la deja igual al terminar.

Antipatrón: no limpiar tras el test

🚫 Incorrecto

```
def setUp(self):
    conn = sqlite3.connect(self.BD_TEST)
    ...
# (sin tearDown)
```

Los ficheros se acumulan y los tests se contaminan entre ejecuciones.

✅ Correcto

```
def setUp(self):
    if self.BD_TEST.exists():
        self.BD_TEST.unlink()
    ...

def tearDown(self):
    if self.BD_TEST.exists():
        self.BD_TEST.unlink()
```

3. Testar las operaciones del repositorio

Verificar guardar, obtener y el tipo del objeto devuelto

`test_guardar_persiste_el_producto`

El test verifica el **resultado observable**: tras `guardar()`, los datos se recuperan con `obtener()`.

```
def test_guardar_persiste_el_producto(self):
    self.repo.guardar(Item("A1", "Agua", 1.00, 10))
    item = self.repo.obtener("A1")
    self.assertEqual(item.nombre, "Agua")
    self.assertEqual(item.precio, 1.00)
    self.assertEqual(item.cantidad, 10)
```

⚠️ Verificamos **atributos individuales**, no el objeto completo, porque `Item` no tiene `__eq__` definido. Comparar objetos enteros produciría siempre `False`.

`assertIsInstance` — el tipo correcto

El repositorio debe devolver el tipo exacto según si el producto tiene descuento o no.

```
def test_obtener_devuelve_item_con_descuento(self):
    self.repo.guardar(ItemConDescuento("D1", "Refresco", 2.50, 5, 20))
    item = self.repo.obtener("D1")
    self.assertIsInstance(item, ItemConDescuento)
    self.assertAlmostEqual(item.porcentaje_descuento, 20.0)

def test_obtener_devuelve_item_normal(self):
    self.repo.guardar(Item("A1", "Agua", 1.00, 10))
    item = self.repo.obtener("A1")
    self.assertIsInstance(item, Item)
    self.assertNotIsInstance(item, ItemConDescuento)
```

`assertIsInstance` comprueba el tipo del objeto, no solo sus atributos — es el test adecuado cuando la lógica depende de la clase concreta.

Aplicar en el lab UT3-A3

El fichero `tests/test_repositorio_sqlite.py` se construye paso a paso en el **Lab UT3-A3**, aplicando todos los patrones de esta sección.

Desde la carpeta padre de `expendedora/`:

```
python -m unittest expendedora.tests.test_repositorio_sqlite
```

✅ Cada test recibe una BD vacía y la deja igual al terminar — el fichero `test_expendedora.db` no persiste entre ejecuciones.

4. Testar excepciones de dominio

`assertRaises` con excepciones propias y verificación del mensaje

`assertRaises` con excepciones de dominio

Los tests verifican que el repositorio lanza **excepciones de dominio**, no las internas de `sqlite3`.

```
def test_obtener_codigo_inexistente_lanza_excepcion(self):
    with self.assertRaises(ProductoNoEncontradoError):
        self.repo.obtener("X9")

def test_guardar_duplicado_lanza_excepcion(self):
    self.repo.guardar(Item("A1", "Agua", 1.00, 10))
    with self.assertRaises(ProductoYaExisteError):
        self.repo.guardar(Item("A1", "Duplicado", 0.50, 3))
```

Testar `ProductoYaExisteError` en lugar de `sqlite3.IntegrityError` garantiza que la capa de presentación **nunca ve excepciones de SQLite**.

Verificar el mensaje de la excepción

Si el mensaje de error aporta información útil, lo comprobamos con `assertIn` sobre el contexto del `assertRaises`.

```
def test_excepcion_incluye_el_codigo(self):
    with self.assertRaises(ProductoNoEncontradoError) as ctx:
        self.repo.obtener("X9")
    self.assertIn("X9", str(ctx.exception))
```

💡 **`ctx.exception`** contiene el objeto excepción capturado.
`str()` lo convierte a texto para buscar dentro del mensaje con `assertIn`.

Antipatrón: testar la excepción interna de SQLite

🚫 Incorrecto

```
with self.assertRaises(
    sqlite3.IntegrityError
):
    self.repo.guardar(
        Item("A1", "Duplicado", 0.50, 3)
    )
```

El test se acopla a la infraestructura — si cambias SQLite, el test se rompe.

✅ Correcto

```
with self.assertRaises(
    ProductoYaExisteError
):
    self.repo.guardar(
        Item("A1", "Duplicado", 0.50, 3)
    )
```

El test solo conoce excepciones de dominio.

¿Preguntas?

setUp / tearDown

BD_TEST con Path

assertIsInstance

assertRaises

Excepciones de dominio

POO · CEPY · UT3 — Tests para Repositorios SQLite