



UNIVERSIDAD
NACIONAL
DE COLOMBIA

Universidad Nacional de Colombia - sede Bogotá

Facultad de Ingeniería

Departamento de Sistemas e Industrial

Curso: Ingeniería de Software 1 (2016701)

Estudiantes: John Alejandro Pastor Sandoval, Daniel Esteban

Lopez Guaca, Jefferson Jair Figueroa Escobar, Andrés Hernando

Borda Muñoz

Testeo

- 1) El primer test fue de obtener lecturas filtradas por mes, se hace principalmente que se comprueben si los meses se están recogiendo bien de la base de datos para el filtrado de información:

```
@patch("models.classes.usuarios.Lectura")
def test_obtener_lecturas_filtradas_por_mes(self, mock_lectura):
    """Test para obtener lecturas filtradas por mes."""

    lectura_instance = mock_lectura.return_value
    lectura_instance.obtener_por_apartamento.side_effect = lambda aid: [
        {"lec_mes": "ENERO", "valor": 10, "apar_id": aid}
    ]

    lecturas = self.usuario.obtener_lecturas(mes="ENERO")
    self.assertEqual(len(lecturas), 2)
    for lec in lecturas:
        self.assertEqual(lec["lec_mes"], "ENERO")
```

- 2) En este test filtramos los recibos en base a un apartamento y mes (en nuestro sistema no existe el apartamento 2, pero es para probar) así sabemos si estamos recogiendo bien la información de los métodos generados.

```
@patch("models.classes.usuarios.Recibo")
def test_obtener_recibos_admin_filtrado_por_mes_y_apartamento(self, mock_recibo):
    """Test para obtener recibos filtrados por mes y apartamento cuando es admin."""
    self.usuario.es_admin = MagicMock(return_value=True)

    recibo_instance = mock_recibo.return_value
    recibo_instance.obtener_por_apartamento.return_value = [
        {"reci_mes": "ABRIL", "apar_id": 2}
    ]


    recibos = self.usuario.obtener_recibos(mes="ABRIL", apar_id=2)
    self.assertEqual(len(recibos), 1)
    self.assertEqual(recibos[0]["reci_mes"], "ABRIL")
    self.assertEqual(recibos[0]["apar_id"], 2)
```


- 3) Este test sirve para comprobar que no se recojan lecturas si no se pasa un apartamento, como lo hacemos cuando es un reporte general al cual un inquilino no tiene permiso.

```
@patch("models.classes.usuarios.Lectura")
def test_obtener_lecturas_sin_apartamentos(self, mock_lectura):
    """Test para obtener lecturas cuando el usuario no tiene apartamentos."""
    self.usuario.obtener_apartamentos = MagicMock(return_value=[])

    lectura_instance = mock_lectura.return_value
    lectura_instance.obtener_por_apartamento.return_value = []

    lecturas = self.usuario.obtener_lecturas()
    self.assertEqual(lecturas, [])
```


- 4)  **test_generar_reporte_mes:** Esta prueba válida la funcionalidad central de “GeneradorReportes” para crear reportes mensuales de arrendos y servicios públicos. Se simula una base de datos con dos arrendos de valores \$500,000 y \$600,000, verificando que el sistema calcule correctamente el total de \$1,100,000 y mantenga la estructura esperada del reporte con campos como mes, total_arrendos y total_general.

```
class TestGeneradorReportes:
    """Pruebas para GeneradorReportes 
    def test_generar_reporte_mes(self):
        """Reporte mensual"""
        mock_connector = Mock()
        mock_connector.get_filtered.return_value = [
            {"arre_valor": 500000},
            {"arre_valor": 600000}
        ]

        generador = GeneradorReportes(mock_connector)

        # Act
        reporte = generador.generar_reporte_mes("JULIO")

        # Assert
        assert reporte["mes"] == "JULIO"
        assert reporte["total_arrendos"] == 1100000
        assert "total_general" in reporte
```

- 5)  **test_obtener_apartamentos:** Esta prueba verifica que el “GestorApartamentos” “traiga” correctamente la información completa de todos los apartamentos registrados en el sistema. Se mockea (simula) una respuesta con dos apartamentos (101 y 102)

que tienen 2 y 3 personas respectivamente, verificando que el método `obtener_apartamentos()` devuelva la lista completa sin pérdida de información y mantenga la integridad de los campos `apar_id` y `apar_cantidadPersonas`.

```
class TestGestorApartamentos:
    """Pruebas para GestorApartamentos"""


    def test_obtener_apartamentos(self):
        """Prueba obtener lista de apartamentos"""

        mock_connector = Mock()
        mock_connector.get_all.return_value = [
            {"apar_id": 101, "apar_cantidadPersonas": 2},
            {"apar_id": 102, "apar_cantidadPersonas": 3}
        ]

        gestor = GestorApartamentos(mock_connector)

        # Act
        apartamentos = gestor.obtener_apartamentos()

        # Assert
        assert len(apartamentos) == 2
        assert apartamentos[0]["apar_id"] == 101
        assert apartamentos[1]["apar_cantidadPersonas"] == 3
```

- 6)  **test_obtener_pagos_pendientes:** Esta prueba comprueba que el “GestorPagos” filtre y retorne únicamente los pagos que se encuentran en estado "PENDIENTE" dentro del sistema de gestión financiera. Se mockea una base de datos con dos pagos pendientes por valores de \$75,000 y \$45,000, verificando que el método aplique correctamente el filtro por estado y mantenga la precisión de los montos

```
class TestGestorPagos:
    """Pruebas para GestorPagos"""

    def test_obtener_pagos_pendientes(self):
        """Prueba obtener pagos pendientes"""

        mock_connector = Mock()
        mock_connector.get_filtered.return_value = [
            {"pago_id": 1, "pago_estado": "PENDIENTE", "pago_valorTotal": 75000},
            {"pago_id": 2, "pago_estado": "PENDIENTE", "pago_valorTotal": 45000}
        ]

        gestor = GestorPagos(mock_connector)

        # Act
        pagos = gestor.obtener_pagos_pendientes()

        # Assert
        assert len(pagos) == 2
        assert all(p["pago_estado"] == "PENDIENTE" for p in pagos)
        assert pagos[0]["pago_valorTotal"] == 75000
```

- 7) **test_apartamentos_validos:** Este test permite verificar si el “gestor de apartamentos” logra filtrar y obtener los apartamentos validos ya ingresados en la base de datos. Se mockea una base de datos con ID de los apartamentos validos, verificando que el método aplique correctamente el filtro por estado y mantenga la precisión de los apartamentos ingresados.

```
class TestManagers:
    """Pruebas para managers"""

    def test_apartamentos_validos(self):
        """Prueba obtener lista de apartamentos"""

        mock_connector = Mock()
        mock_connector.get_all.return_value = [
            {"apar_id": 1},
            {"apar_id": 101},
            {"apar_id": 201},
            {"apar_id": 202},
            {"apar_id": 301},
            {"apar_id": 302},
            {"apar_id": 401}
        ]

        gestor = GestorApartamentos(mock_connector)

        # Act
        apartamentos = gestor.obtener_apartamentos()

        # Assert
        assert len(apartamentos) != 0
        assert apartamentos[0]["apar_id"] != 0
```

- 8) **test_generar_reporte_mes_enero:** Este test permite verificar si el “gestor de reportes” logra filtrar y obtener los reportes válidos en el mes de ENERO ya ingresados en la base de datos. Se mockea una base de datos con arriendos que ya han sido validados en el mes de ENERO (válidos), verificando que el método aplique correctamente el filtro por estado y mantenga la precisión de los arriendos filtrados.

```
def test_generar_reporte_mes_enero(self):
    """Reporte mensual enero"""
    mock_connector = Mock()
    mock_connector.get_filtered.return_value = [
        {"arre_valor": 500000},
        {"arre_valor": 600000}
    ]

    generador = GeneradorReportes(mock_connector)

    # Act
    reporte = generador.generar_reporte_mes("ENERO")

    # Assert
    assert reporte["mes"] == "ENERO"
    assert reporte["total_arrendos"] != 0
    assert "total_general" in reporte
```

- 9) **test_obtener_pagos_cancelados:** Este test permite verificar si el “gestor de pagos” logra filtrar y obtener los pagos CANCELADOS ya ingresados en la base de datos. Se mockea una base de datos con arriendos que ya han sido CANCELADOS (válidos), verificando que el método aplique correctamente el filtro por estado y mantenga la precisión de los arriendos filtrados.

```
def test_obtener_pagos_cancelados(self):
    """Prueba obtener pagos cancelados"""

    mock_connector = Mock()
    mock_connector.get_filtered.return_value = [
        {"pago_id": 1, "pago_estado": "CANCELADO", "pago_valorTotal": 75000},
        {"pago_id": 2, "pago_estado": "CANCELADO", "pago_valorTotal": 45000}
    ]

    gestor = GestorPagos(mock_connector)

    # Act
    pagos = gestor.obtener_pagos_pendientes()

    # Assert
    assert len(pagos) == 2
    assert all(p["pago_estado"] == "CANCELADO" for p in pagos)
    assert pagos[0]["pago_valorTotal"] == 45000
```

LINTER:

Pylint es una herramienta de análisis estático de código fuente Python, que permite detectar errores de sintaxis, problemas de estilo y convenciones de codificación, así como advertencias sobre código mal estructurado

Usamos la versión de Pylint: 2.17.4 y como archivo de configuración se usó el archivo por defecto, con algunos ajustes locales vía comentarios como:

```
# pylint: disable=broad-exception-caught

# pylint: disable=line-too-long
```

Le pasamos el linter a la carpeta con el proyecto y la IA nos resumió estos principales problemas o advertencias en el proyecto y los archivos correspondientes:

Tras analizar el archivo `usuario.py`, se obtuvieron los siguientes errores y advertencias relevantes:

Tipo	Código	Descripción	Línea	
Error	E0401	Unable to import 'services.servicio_energia'	4	
Convención	C0411	Wrong import order: third party before first party	5	
Advertencia	W0611	Unused import 'datetime'	2	
Convención	C0301	Line too long (exceeds 100 characters)	21	

Muchos de estos fueron resueltos formateando el código, añadiendo los docstrings a las clases y métodos y en algunos usando comentarios para excepciones de pylint, sin embargo, otros de los errores siguen presentes porque no consideramos conveniente arreglarlos aun. Aunque la IA no lo muestra, en nuestras pruebas sin interfaz gráfica, se creó un menú que contiene muchos if anidados los cuales tampoco se buscan arreglar debido a que pretendemos montar la interfaz gráfica como prototipo funcional.

```
***** Module main
src/main.py:10:0: C0413: Import "from connector.connector import Connector" should be placed at the top of the module (wrong-import-order)
src/main.py:11:0: C0413: Import "from models.classes.usuarios import Usuario" should be placed at the top of the module (wrong-import-order)
src/main.py:26:4: R1702: Too many nested blocks (6/5) (too-many-nested-blocks)
src/main.py:23:0: R0912: Too many branches (24/12) (too-many-branches)
src/main.py:23:0: R0915: Too many statements (59/50) (too-many-statements)
src/main.py:99:0: R0914: Too many local variables (20/15) (too-many-locals)
src/main.py:101:4: R1702: Too many nested blocks (6/5) (too-many-nested-blocks)
src/main.py:101:4: R1702: Too many nested blocks (6/5) (too-many-nested-blocks)
src/main.py:99:0: R0912: Too many branches (27/12) (too-many-branches)
src/main.py:99:0: R0915: Too many statements (92/50) (too-many-statements)
src/main.py:267:0: R0914: Too many local variables (19/15) (too-many-locals)
src/main.py:273:14: E1120: No value for argument 'connector' in unbound method call (no-value-for-parameter)
src/main.py:278:4: R1702: Too many nested blocks (6/5) (too-many-nested-blocks)
src/main.py:267:0: R0912: Too many branches (32/12) (too-many-branches)
src/main.py:267:0: R0915: Too many statements (110/50) (too-many-statements)
```

```
def actualizar_rol_texto(self):
    if self.role_switch.isChecked():
        self.role_label.setText("Soy arrendador")
    else:
        self.role_label.setText("Soy arrendatario") (duplicate-code)

-----
Your code has been rated at 7.30/10 (previous run: 7.30/10, +0.00)
```

El linter nos entregó un resultado de 7.3/10, lo que significa que el código es legible entendible y sigue buenos patrones de diseño, pero podría ser mejor arreglando los demás errores que se muestra en el reporte entregado el cual está en el repositorio del proyecto.