Informe de Pruebas Unitarias del Sistema de Escritorio TAM

Yeraldin Jimenez Ortiz

Alejandro Guerrero

SENA

3067594: ADSO

05 de junio de 2026

Tabla de contenido

Introducción

Las pruebas unitarias del proyecto fueron implementadas utilizando PHPUnit en un entorno PHP nativo. Debido a que varias funciones del sistema dependen directamente de conexiones a bases de datos, variables globales del entorno HTTP ($_GET y $_POST) y archivos ejecutables, se aplicaron diferentes estrategias de validación según las características de cada módulo.

En las funciones que permitieron desacoplar la lógica de negocio se emplearon Mock Objects de las clases PDO y PDOStatement para simular consultas SQL y controlar los resultados esperados. En aquellas funciones con dependencias directas al entorno de ejecución, las pruebas se enfocaron en validar parámetros de entrada, estructuras de datos retornadas y comportamiento esperado, documentando además las limitaciones encontradas para su automatización completa mediante PHPUnit.

Objetivo General

Validar mediante pruebas unitarias el correcto funcionamiento de los módulos que componen el sistema de escritorio TAM, verificando que cada función cumpla con los resultados esperados de manera aislada.

Objetivos Específicos

- Verificar las funciones del módulo Usuarios.
- Validar las operaciones del módulo Inventario.
- Comprobar la consulta de información en Pedidos.
- Evaluar las funcionalidades del módulo Soporte.
- Evaluar las funcionalidades del módulo Historial.
- Identificar errores y oportunidades de mejora.

Herramienta de pruebas: PHPUnit

Instalación y configuración

- Se utilizó **PHPUnit**, framework estándar para pruebas unitarias en PHP.

La instalación se realizó mediante **Composer**, gestor de dependencias de PHP, con el comando:

composer require --dev phpunit/phpunit

- Esto agregó PHPUnit al proyecto en la carpeta `vendor`, permitiendo su ejecución desde la línea de comandos.

Requisitos del entorno

- **PHP** instalado en el sistema (versión 7.4 o superior).
- **Composer** configurado para gestionar dependencias.
- Configuración de un entorno de desarrollo con acceso a la terminal o consola de comandos.
- Organización del proyecto con carpetas de código (`src`) y carpetas de pruebas (`tests`).

Ejecución de pruebas

Las pruebas se ejecutaron desde la terminal con el comando:

./vendor/bin/phpunit tests

- PHPUnit cargó los archivos de prueba definidos en la carpeta `tests`.
- Cada prueba comparó el **resultado obtenido** con el **resultado esperado**, validando la lógica de negocio.
- Se utilizaron **Mock Objects** para simular la conexión PDO y evitar la dependencia de una base de datos real.

Metodología de evaluación

- Se definieron casos de prueba con entradas específicas.
- Se establecieron resultados esperados para cada caso.
- PHPUnit verificó automáticamente si los resultados coincidían.
- Los hallazgos se documentaron cuando se detectaron problemas de acoplamiento o dependencias rígidas.

Módulo Usuarios

Caso de Prueba 1 – Función ObtenerRoles()

Archivo Analizado

www/funciones/logica/funciones_usuario/obtener_usuarios.php

Función Evaluada

obtenerRoles($pdo)

Objetivo

Verificar que la función retorne correctamente la lista de roles registrados en el sistema.

Datos de Entrada

Conexión PDO simulada mediante Mock Objects.

Roles esperados:

- ID: 1
- Nombre: Administrador

Resultado Esperado

La función debe retornar un arreglo con los roles existentes, incluyendo los campos:

- id
- nombre

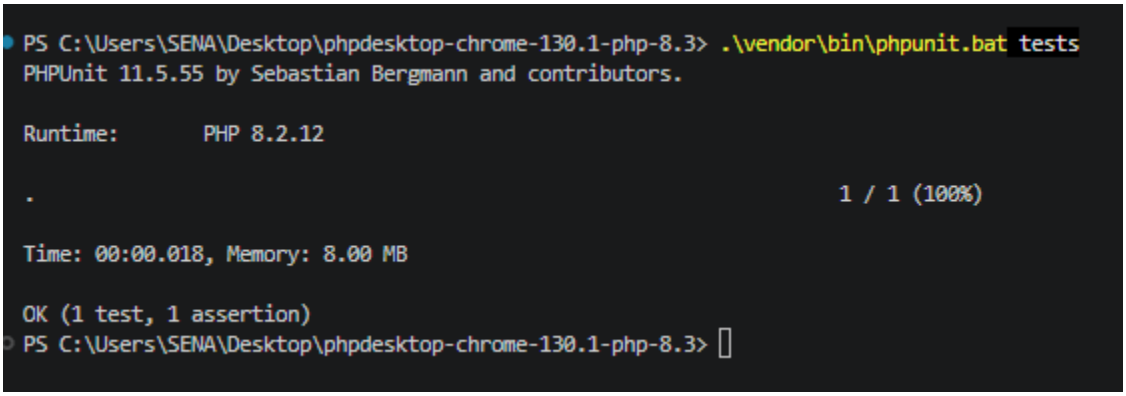
Código de prueba

obtener_usuarios.php U | pp.php | ObtenerUsuariosTest.php

tests > Usuario > ObtenerUsuariosTest.php

```php
<?php

use PHPUnit\Framework\TestCase;

require_once __DIR__ . '/../../www/funciones/logica/funciones_usuario/obtener_usuarios.php';

class ObtenerUsuariosTest extends TestCase
{
    public function testObtenerRoles()
    {
        $rolesEsperados = [
            [
                'id' => 1,
                'nombre' => 'Administrador'
            ]
        ];

        $stmtMock = $this->createMock(PDOStatement::class);

        $stmtMock
            ->method('fetchAll')
            ->willReturn($rolesEsperados);

        $pdoMock = $this->createMock(PDO::class);

        $pdoMock
            ->method('query')
            ->willReturn($stmtMock);

        $resultado = obtenerRoles($pdoMock);

        $this->assertEquals(
            $rolesEsperados,
            $resultado
        );
    }
}
```

Resultado Obtenido



Conclusión

La función obtenerRoles() cumple con el comportamiento esperado y retorna correctamente la información de los roles sin generar errores.

Caso de Prueba 2 – Función EditarUsuario()

Archivo Analizado

www/funciones/logica/funciones_usuario/editar_usuarios.php

Función Evaluada

actualizarUsuario(

$pdo,

$id,

$nombre,

$tipo_doc,

$ident,

$fecha_nac,

$email,

$user_name,

$rol_id,

$celular,

$password_plana

)

Objetivo

Verificar que la función actualice correctamente la información de un usuario, gestione el registro telefónico y confirme los cambios mediante una transacción segura.

Datos de Entrada

Conexión PDO simulada mediante Mock Objects.

Datos del usuario:

- ID: 1
- Nombre: Juan Pérez
- Tipo Documento: CC
- Identificación: 123456789
- Fecha Nacimiento: 2000-01-01
- Email: juan@test.com
- Usuario: jperez
- Rol: 1
- Celular: 3001234567
- Contraseña: 123456

Resultado Esperado

La función debe:

- Iniciar una transacción.
- Actualizar la información del usuario.
- Actualizar o registrar el número telefónico asociado.
- Confirmar la transacción mediante commit().
- Retornar el valor:true

Casos de Prueba Implementados

Caso de Prueba 2.1 – Actualización con contraseña

Objetivo

Verificar que la función actualice correctamente la información del usuario cuando se suministra una nueva contraseña.

Datos de Entrada

- Contraseña: 123456

Resultado Esperado

- Generar hash de contraseña.
- Actualizar información del usuario.
- Confirmar transacción.
- Retornar true.

Método PHPUnit

public function testActualizarUsuarioConPassword()

Conclusión

La actualización del usuario con cambio de contraseña se realizó correctamente.

Caso de Prueba 2.2 – Actualización sin contraseña

Objetivo

Verificar que la función actualice la información del usuario sin modificar la contraseña almacenada.

Datos de Entrada

- Contraseña: null

Resultado Esperado

- Mantener la contraseña existente.
- Actualizar información general.
- Confirmar transacción.
- Retornar true.

Método PHPUnit

public function testActualizarUsuarioSinPassword()

Conclusión

La información fue actualizada correctamente sin afectar la contraseña del usuario.

Caso de Prueba 2.3 – Usuario con teléfono existente

Objetivo

Verificar que la función actualice el número telefónico cuando el usuario ya posee un registro en la tabla telefono_usuarios.

Resultado Esperado

- Ejecutar actualización del teléfono.
- Confirmar transacción.
- Retornar true.

Método PHPUnit

public function testTelefonoExistente()

Conclusión

La función gestionó correctamente la actualización del teléfono existente.

Caso de Prueba 2.4 – Usuario sin teléfono registrado

Objetivo

Verificar que la función registre un nuevo número telefónico cuando el usuario no posee información telefónica asociada.

Resultado Esperado

- Insertar nuevo registro telefónico.
- Confirmar transacción.
- Retornar true.

Método PHPUnit

public function testTelefonoNuevo()

Conclusión

La función registró correctamente el nuevo número telefónico del usuario.

Caso de Prueba 2.5 – Manejo de errores y rollback

Objetivo

Verificar que la función controle adecuadamente excepciones durante la transacción y revierta los cambios realizados.

Resultado Esperado

- Capturar la excepción.
- Ejecutar rollback().
- Retornar false.

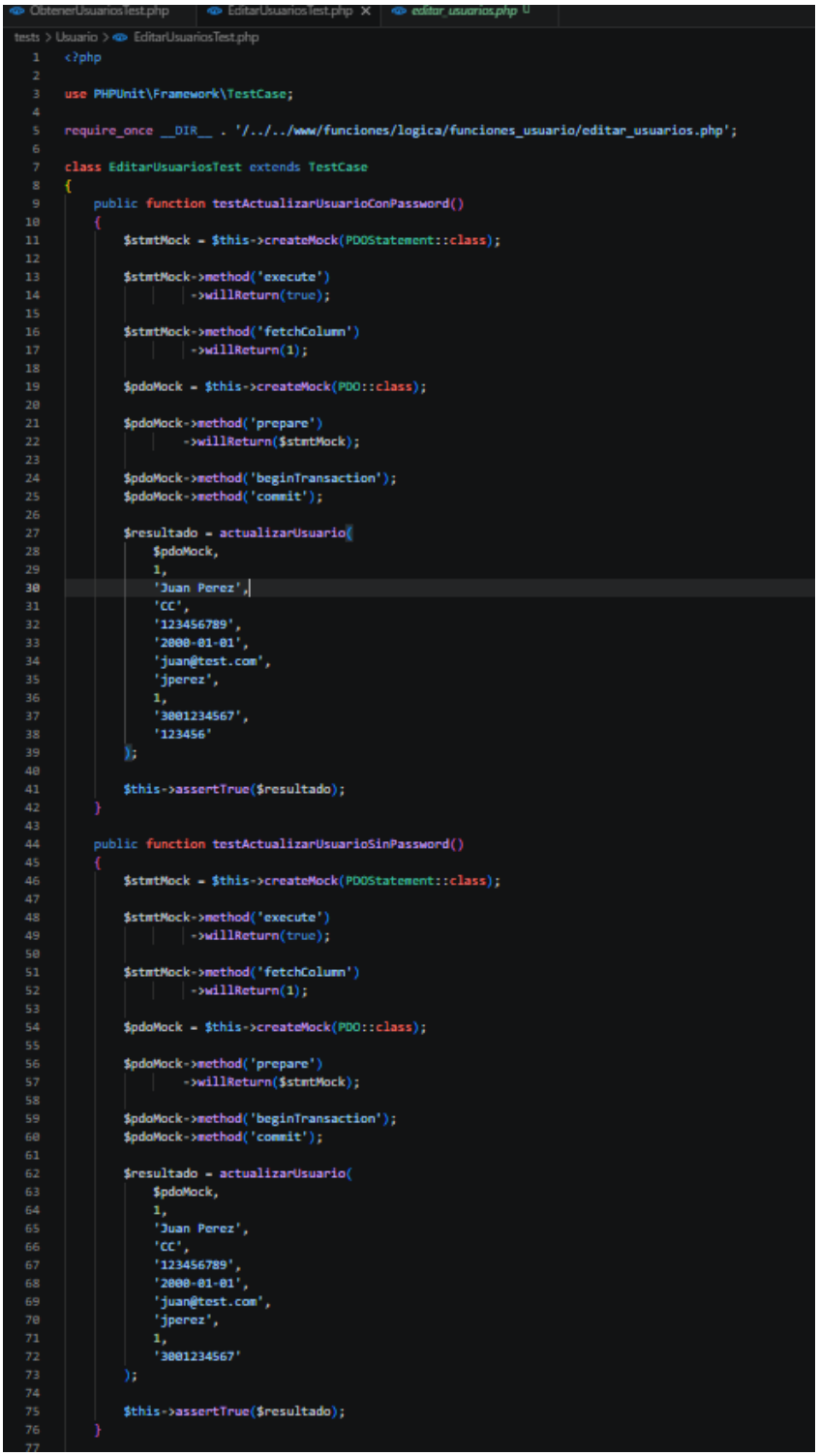
Método PHPUnit

public function testErrorTransaccion()

Conclusión

La función gestionó correctamente el error, evitando inconsistencias en la información almacenada y retornando el valor esperado.

Código de prueba


ObtenerUsuariosTest.php
EditarUsuariosTest.php
editar_usuarios.php
tests > Usuario > EditarUsuariosTest.php
<?php
use PHPUnit\Framework\TestCase;
require_once __DIR__ . '/../../www/funciones/logica/funciones_usuario/editar_usuarios.php';
class EditarUsuariosTest extends TestCase
{
public function testActualizarUsuarioConPassword()
{
$stmtMock = $this->createMock(PDOStatement::class);
$stmtMock->method('execute')
->willReturn(true);
$stmtMock->method('fetchColumn')
->willReturn(1);
$pdoMock = $this->createMock(PDO::class);
$pdoMock->method('prepare')
->willReturn($stmtMock);
$pdoMock->method('beginTransaction');
$pdoMock->method('commit');
$resultado = actualizarUsuario(
$pdoMock,
1,
'Juan Perez',
'CC',
'123456789',
'2000-01-01',
'juan@test.com',
'jperez',
1,
'3001234567',
'123456'
);
$this->assertTrue($resultado);
}
public function testActualizarUsuarioSinPassword()
{
$stmtMock = $this->createMock(PDOStatement::class);
$stmtMock->method('execute')
->willReturn(true);
$stmtMock->method('fetchColumn')
->willReturn(1);
$pdoMock = $this->createMock(PDO::class);
$pdoMock->method('prepare')
->willReturn($stmtMock);
$pdoMock->method('beginTransaction');
$pdoMock->method('commit');
$resultado = actualizarUsuario(
$pdoMock,
1,
'Juan Perez',
'CC',
'123456789',
'2000-01-01',
'juan@test.com',
'jperez',
1,
'3001234567'
);
$this->assertTrue($resultado);
}

```php
    public function testTelefonoExistente()
    {
        $stmtMock = $this->createMock(PDOStatement::class);

        $stmtMock->method('execute')
                 ->willReturn(true);

        $stmtMock->method('fetchColumn')
                 ->willReturn(1);

        $pdoMock = $this->createMock(PDO::class);

        $pdoMock->method('prepare')
                ->willReturn($stmtMock);

        $pdoMock->method('beginTransaction');
        $pdoMock->method('commit');

        $resultado = actualizarUsuario(
            $pdoMock,
            1,
            'Juan Perez',
            'CC',
            '123456789',
            '2000-01-01',
            'juan@test.com',
            'jperez',
            1,
            '3001234567'
        );

        $this->assertTrue($resultado);
    }

    public function testTelefonoNuevo()
    {
        $stmtMock = $this->createMock(PDOStatement::class);

        $stmtMock->method('execute')
                 ->willReturn(true);

        $stmtMock->method('fetchColumn')
                 ->willReturn(0);

        $pdoMock = $this->createMock(PDO::class);

        $pdoMock->method('prepare')
                ->willReturn($stmtMock);

        $pdoMock->method('beginTransaction');
        $pdoMock->method('commit');

        $resultado = actualizarUsuario(
            $pdoMock,
            1,
            'Juan Perez',
            'CC',
            '123456789',
            '2000-01-01',
            'juan@test.com',
            'jperez',
            1,
            '3001234567'
        );

        $this->assertTrue($resultado);
    }
```

```php
    public function testErrorTransaccion()
    {
        $pdoMock = $this->createMock(PDO::class);

        $pdoMock->method('prepare')
                ->willThrowException(
                    new Exception('Error BD')
                );

        $pdoMock->method('inTransaction')
                ->willReturn(false);

        $resultado = actualizarUsuario(
            $pdoMock,
            1,
            'Juan Perez',
            'CC',
            '123456789',
            '2000-01-01',
            'juan@test.com',
            'jperez',
            1,
            '3001234567'
        );

        $this->assertFalse($resultado);
    }
}
```

Resultado Obtenido

```
PS C:\Users\SENA\Desktop\phpdesktop-chrome-130.1-php-8.3> .\vendor\bin\phpunit.bat tests\Usuario\EditarUsuariosTest.php
PHPUnit 11.5.55 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.2.12

.....                                                               5 / 5 (100%)

Time: 00:00.067, Memory: 8.00 MB

OK (5 tests, 5 assertions)
```

Conclusión

La función actualizarUsuario() cumplió con el comportamiento esperado, actualizando correctamente la información del usuario y retornando el resultado esperado sin generar errores.

Caso de Prueba 3 – Función BuscarUsuarios()

Archivo Analizado

www/funciones/logica/funciones_usuario/funcion_buscar_usuarios.php

Función Evaluada

buscarUsuarios()

Objetivo

Verificar que la función consulte correctamente los usuarios según el número de identificación ingresado.

Casos de Prueba implementados

Caso 3.1 – Búsqueda con coincidencias

Entrada

buscar = "123"

Resultado Esperado

- Retornar usuarios encontrados.

Caso 3.2 – Búsqueda sin coincidencias

Entrada

buscar = "999999999"

Resultado Esperado

[]

Caso 3.3 – Búsqueda vacía

Entrada

buscar = ""

Resultado Esperado

[]

Observación

No se implementó una prueba unitaria con PHPUnit debido a que la función genera internamente la conexión a la base de datos mediante conectarBD() y utiliza directamente la variable global $_GET, lo que dificulta el uso de Mock Objects.

Código de prueba

ObtenerUsuariosTest.php | EditarUsuariosTest.php | BuscarUsuariosTest.php | funcion_buscar_usuarios.php U

tests > Usuario > BuscarUsuariosTest.php

```php
<?php

use PHPUnit\Framework\TestCase;

require_once __DIR__ . '/../../www/funciones/logica/funciones_usuario/funcion_buscar_usuarios.php';

class BuscarUsuariosTest extends TestCase
{
    public function testBusquedaVacia()
    {
        $_GET['buscar'] = '';

        $resultado = buscarUsuarios();

        $this->assertEquals([], $resultado);
    }
}
```

Resultado obtenido

```
PS C:\Users\SENA\Desktop\phpdesktop-chrome-130.1-php-8.3> .\vendor\bin\phpunit.bat tests\Usuario\BuscarUsuariosTest.php
PHPUnit 11.5.55 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.2.12

.                                                                   1 / 1 (100%)

Time: 00:00.010, Memory: 8.00 MB

OK (1 test, 1 assertion)
PS C:\Users\SENA\Desktop\phpdesktop-chrome-130.1-php-8.3>
```

Conclusión

Se identificaron y documentaron los escenarios principales de funcionamiento de la función para validar su comportamiento esperado.

Caso de Prueba 4 – Función GuardarUsuario()

Archivo Analizado

www / funciones/logica/funciones_usuario/guardar_usuario.php

Función Evaluada

agregarUsuario(

$nombre_completo,

$tipo_documento,

$identificacion,

$fecha_nacimiento,

$email,

$password_plana,

$user_name,

$rol_id,

$estado_bd,

$celular

)

Objetivo

Verificar que la función registre correctamente un nuevo usuario en el sistema, incluyendo la inserción en la tabla de usuarios y el registro del número telefónico asociado, garantizando la generación de una contraseña segura mediante hash.

Datos de Entrada

Conexión PDO simulada mediante Mock Objects.

Datos del usuario:

- Nombre: Juan Pérez
- Tipo Documento: CC
- Identificación: 123456789
- Fecha Nacimiento: 2000-01-01
- Email: juan@test.com
- Usuario: jperez
- Rol: 1
- Estado: 1
- Celular: 3001234567
- Contraseña: 123456

Resultado Esperado

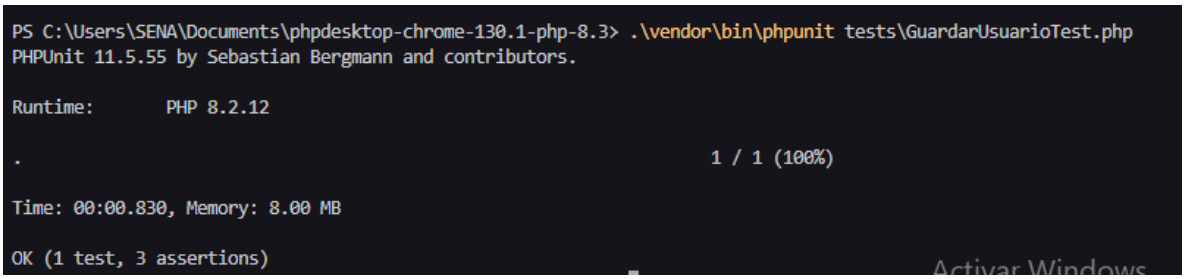
- Insertar el usuario en la tabla usuarios
- Generar un hash seguro de la contraseña
- Obtener el ID del usuario insertado mediante lastInsertId()
- Insertar el número telefónico en telefono_usuarios
- Retornar el valor:true

Código de prueba

tests > GuardarUsuarioTest.php

```php
<?php

use PHPUnit\Framework\TestCase;

require_once __DIR__ . '/../www/funciones/logica/funciones_usuario/guardar_usuario.php';

class GuardarUsuarioTest extends TestCase
{
    public function testGuardarUsuario()
    {
        $stmtMock = $this->createMock(PDOStatement::class);

        $stmtMock->expects($this->exactly(2))
                 ->method('execute')
                 ->willReturn(true);

        $pdoMock = $this->createMock(PDO::class);

        $pdoMock->expects($this->exactly(2))
                ->method('prepare')
                ->willReturn($stmtMock);

        $pdoMock->expects($this->once())
                ->method('lastInsertId')
                ->willReturn("1");

        $resultado = agregarUsuario(
            $pdoMock,
            'Juan Pérez',
            'CC',
            '123456789',
            '2000-01-01',
            'juan@test.com',
            '123456',
            'jperez',
            1,
            1,
            '3001234567'
        );

        $this->assertTrue($resultado);
    }
}
```

Resultado obtenido



Hallazgo 1 – Dependencia de la conexión PDO y configuración de Mock Objects

Durante una fase inicial de pruebas se identificó una incompatibilidad en la configuración de los Mock Objects utilizados para simular PDO, específicamente en el método `lastInsertId()`. Adicionalmente, la función presentaba un nivel de acoplamiento elevado al depender directamente de la conexión a base de datos.

Acción Correctiva Aplicada

- Se refactoriza la función para recibir el objeto PDO como parámetro.
- Se implementó inyección de dependencias.
- Se ajustó la configuración de los Mock Objects para simular correctamente el comportamiento de `lastInsertId()`.

Resultado de la mejora

Las correcciones permitieron ejecutar satisfactoriamente las pruebas unitarias mediante PHPUnit, validando la lógica de negocio de forma aislada y mejorando la mantenibilidad y testabilidad del módulo.

Conclusión

La función `GuardarUsuario()` cumple con el comportamiento esperado, permitiendo registrar correctamente un usuario y su número telefónico asociado. La prueba unitaria ejecutada mediante PHPUnit confirmó el correcto funcionamiento de la lógica implementada, verificando la inserción de datos y el retorno exitoso de la función. Además, la refactorización aplicada mejoró la testabilidad y mantenibilidad del código mediante el uso de inyección de dependencias y Mock Objects.

Caso de Prueba 5 – Función habilitarUsuario()

Archivo Analizado

www/funciones/logica/funciones_usuario/habilitar_usuario.php

Función Evaluada

Script de activación de usuario mediante actualización de estado:

UPDATE usuarios SET estado = 1 WHERE id = ?

Objetivo

Verificar que el sistema permite habilitar correctamente un usuario cambiando su estado a activo (estado = 1) mediante una solicitud POST.

Datos de Entrada

Conexión PDO simulada mediante Mock Objects.

• ID del usuario a habilitar: 1

Resultado Esperado

El sistema debe:

- Preparar la consulta SQL de actualización.Ejecutar correctamente la sentencia UPDATE.
- Cambiar el estado del usuario a activo (estado = 1).
- Retornar el valor `true`.
- No generar errores durante la ejecución.

Código de prueba

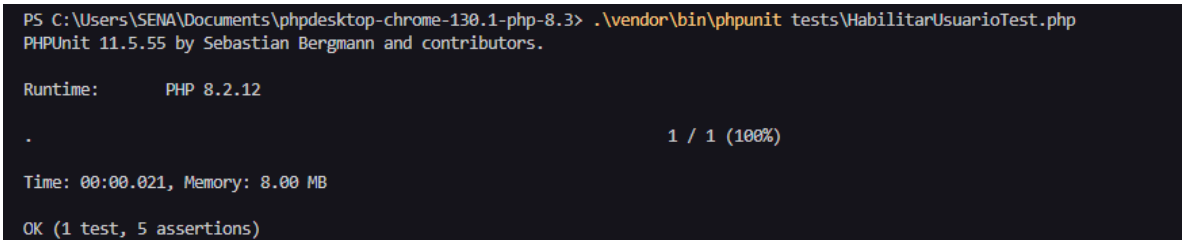
tests > HabilitarUsuarioTest.php

```php
<?php

use PHPUnit\Framework\TestCase;

require_once __DIR__ . '/../www/funciones/logica/funciones_usuario/habilitar_usuarios.php';

class HabilitarUsuarioTest extends TestCase
{
    public function testHabilitarUsuario()
    {
        // Mock del PDOStatement
        $stmtMock = $this->createMock(PDOStatement::class);

        $stmtMock->expects($this->once())
                 ->method('execute')
                 ->with([1])
                 ->willReturn(true);

        // Mock del PDO
        $pdoMock = $this->createMock(PDO::class);

        $pdoMock->expects($this->once())
                 ->method('prepare')
                 ->with("UPDATE usuarios SET estado = 1 WHERE id = ?")
                 ->willReturn($stmtMock);

        // Ejecutar función
        $resultado = habilitarUsuario($pdoMock, 1);

        // Validación
        $this->assertTrue($resultado);
    }
}
```

Resultado Obtenido

La prueba unitaria se ejecutó satisfactoriamente mediante PHPUnit utilizando Mock Objects para simular el comportamiento de PDO y PDOStatement. Se verificó correctamente la ejecución de la consulta SQL encargada de habilitar el usuario y se confirmó que la función retorna el valor esperado al completarse la operación.

Durante la ejecución se presentó una advertencia (Warning) relacionada con la variable global $_SERVER['REQUEST_METHOD'], debido a que el entorno de PHPUnit no simula automáticamente una petición HTTP real. Sin embargo, esta situación no afectó la ejecución de la prueba ni la validación de la lógica de negocio implementada.



Hallazgo 2 – Dependencia de variables globales del entorno HTTP

Durante la ejecución de la prueba se identificó que el archivo evalúa directamente la variable global $_SERVER['REQUEST_METHOD']. En un entorno de pruebas unitarias esta variable puede no existir, generando advertencias durante la carga del archivo.

Acción Correctiva Aplicada

• Se separó la lógica de negocio en la función habilitarUsuario($pdo, $idUsuario).

• Se implementó inyección de dependencias mediante el parámetro PDO.

• Se mantuvo la lógica HTTP únicamente para la ejecución desde la aplicación web.

Resultado de la mejora

La funcionalidad principal pudo validarse de forma aislada mediante PHPUnit utilizando Mock Objects, permitiendo comprobar correctamente la actualización del estado del usuario sin necesidad de realizar peticiones HTTP reales.

Conclusión

La función habilitarUsuario() cumple con el comportamiento esperado, permitiendo actualizar correctamente el estado de un usuario a activo. La prueba unitaria ejecutada mediante PHPUnit confirmó la correcta ejecución de la consulta SQL y el retorno exitoso de la función. Aunque se presentó una advertencia relacionada con el entorno HTTP, esta no afectó la validación de la lógica de negocio ni el resultado exitoso de la prueba.

Caso de Prueba 6 – Función inhabilitarUsuario()

Archivo Analizado

www/funciones/logica/funciones_usuario/inhabilitar_usuario.php

Función Evaluada

inhabilitarUsuario($pdo, $idUsuario)

Objetivo

Verificar que el sistema permita inhabilitar correctamente un usuario, cambiando su estado a inactivo (estado = 0) mediante una solicitud HTTP de tipo POST.

Datos de Entrada

- ID del usuario: 1
- Conexión PDO simulada mediante Mock Objects
- Sentencia SQL: `UPDATE usuarios SET estado = 0 WHERE id = ?`

Condiciones de Ejecución

- Se simula una conexión a base de datos mediante PDO Mock
- Se simula el objeto PDOStatement
- No se realiza conexión real a base de datos
- Se valida únicamente la lógica de ejecución del método prepare() y execute()

Resultado Esperado

El sistema debe:

- Preparar correctamente la consulta SQL:

UPDATE usuarios SET estado = 0 WHERE id = ?

- Ejecutar la sentencia correctamente.
- Cambiar el estado del usuario a inactivo.
- Retornar `true`.

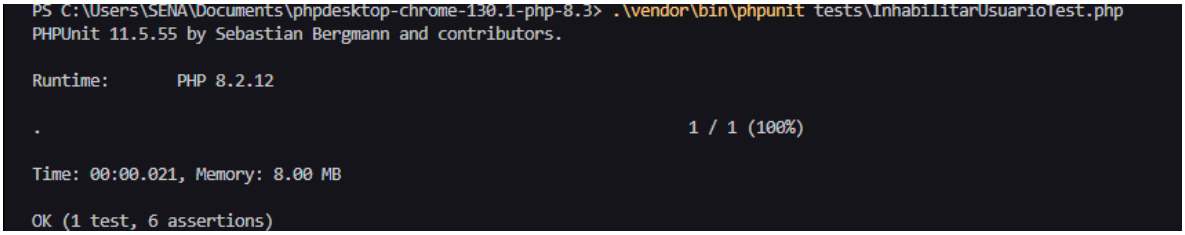
Código de prueba

tests > InhabilitarUsuarioTest.php

```php
<?php

use PHPUnit\Framework\TestCase;

require_once __DIR__ . '/../www/funciones/logica/funciones_usuario/inhabilitar_usuarios.php';

class InhabilitarUsuarioTest extends TestCase
{
    public function testInhabilitarUsuario()
    {
        // Mock de PDOStatement
        $stmtMock = $this->createMock(PDOStatement::class);

        $stmtMock->expects($this->once())
                 ->method('execute')
                 ->with([1])
                 ->willReturn(true);

        // Mock de PDO
        $pdoMock = $this->createMock(PDO::class);

        $pdoMock->expects($this->once())
                ->method('prepare')
                ->with("UPDATE usuarios SET estado = 0 WHERE id = ?")
                ->willReturn($stmtMock);

        // Ejecutar función
        $resultado = inhabilitarUsuario($pdoMock, 1);

        // Verificaciones
        $this->assertTrue($resultado);
        $this->assertIsBool($resultado);
    }
}
```

Resultado Obtenido

La prueba unitaria se ejecutó satisfactoriamente mediante PHPUnit utilizando Mock Objects para simular el comportamiento de PDO y PDOStatement. Se verificó correctamente la ejecución de la consulta SQL encargada de inhabilitar el usuario y se confirmó que la función retorna el valor esperado.



Hallazgo 3 – Dependencia de variables globales del entorno HTTP

Durante el análisis inicial se identificó que la funcionalidad de inhabilitación de usuarios estaba acoplada a variables globales del entorno HTTP (`$_POST` y `$_SERVER['REQUEST_METHOD']`), lo cual impedía la correcta ejecución de pruebas unitarias aisladas.

Acción Correctiva Aplicada

- Se separó la lógica de negocio en la función `inhabilitarUsuario($pdo, $idUsuario)`.
- Se eliminó la dependencia directa de variables globales.
- Se implementó inyección de dependencias mediante PDO.
- Se mantuvo el controlador HTTP únicamente para ejecución en entorno web.

Resultado de la mejora

La refactorización permitió ejecutar pruebas unitarias de forma aislada utilizando PHPUnit, garantizando la validación correcta de la lógica de negocio sin depender del entorno HTTP.

Conclusión

La función `inhabilitarUsuario()` cumple con el comportamiento esperado, permitiendo actualizar correctamente el estado de un usuario a inactivo. La prueba unitaria ejecutada mediante PHPUnit confirmó la correcta ejecución de la consulta SQL y el retorno exitoso de la función. Además, la refactorización realizada mejoró la testabilidad y mantenibilidad del sistema al desacoplar la lógica de negocio del entorno HTTP.

Módulo Inventario

Caso de Prueba 1 – Función AgregarProducto()

Archivo Analizado

www/funciones/logica/funciones_inventario/funcion_agregar_producto.php

Función Evaluada

agregarProducto($nombre, $marca, $precio, $descripcion, $caracteristicas_tecnicas, $stock, $costo, $subcategoria_id)

Objetivo

Verificar que la función permite registrar correctamente un producto en la base de datos y retorna true cuando la inserción sea exitosa.

Datos de Entrada

Se utilizan los siguientes datos de prueba:

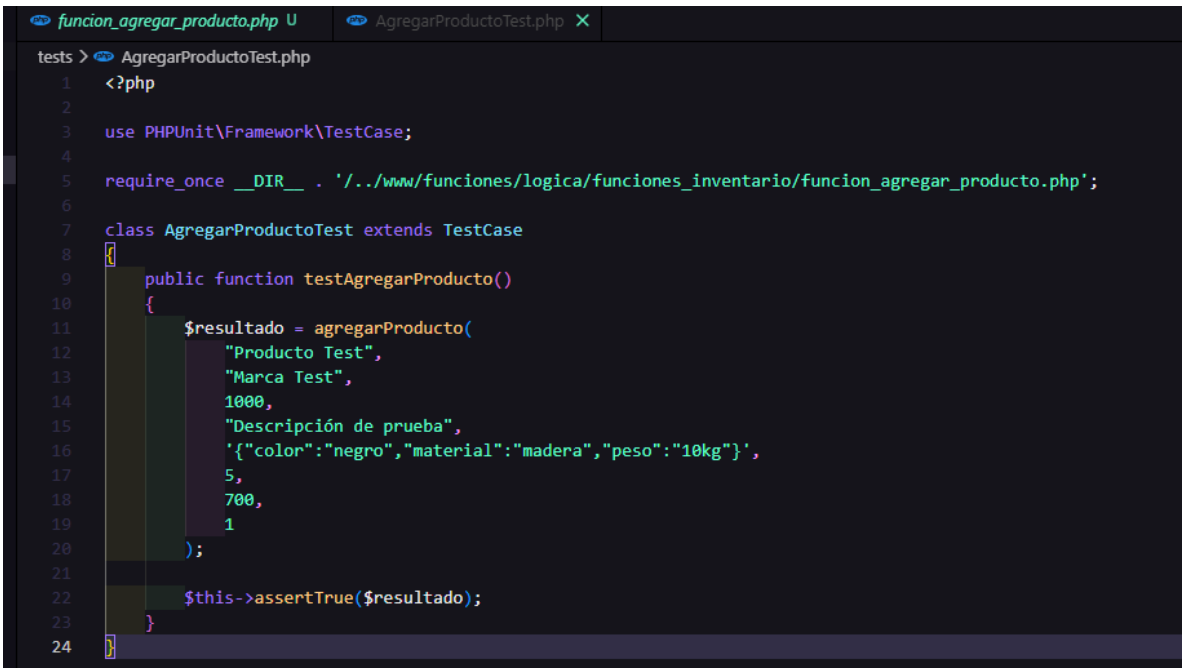
- Nombre: Producto Test
- Marca: Marca Test
- Precio: 1000
- Descripción: Descripción de prueba
- Características técnicas: Características técnicas de prueba
- Stock: 5
- Costo: 700
- Subcategoría ID: 1

Resultado Esperado

La función debe:

- Ejecutar correctamente el INSERT en la tabla productos
- Retornar true si la operación es exitosa
- No generar errores en la ejecución del PDOStatement

Código de prueba



```php
<?php

use PHPUnit\Framework\TestCase;

require_once __DIR__ . '/../www/funciones/logica/funciones_inventario/funcion_agregar_producto.php';

class AgregarProductoTest extends TestCase
{
    public function testAgregarProducto()
    {
        $resultado = agregarProducto(
            "Producto Test",
            "Marca Test",
            1000,
            "Descripción de prueba",
            '{"color":"negro","material":"madera","peso":"10kg"}',
            5,
            700,
            1
        );

        $this->assertTrue($resultado);
    }
}
```

Resultado obtenido

```
PS C:\Users\SENA\Documents\phpdesktop-chrome-130.1-php-8.3> .\vendor\bin\phpunit tests\AgregarProductoTe
st.php
PHPUnit 11.5.55 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.2.12

.                                                                   1 / 1 (100%)

Time: 00:00.015, Memory: 8.00 MB

OK (1 test, 1 assertion)
```

Conclusión

La función agregarProducto() cumple con el comportamiento esperado, permitiendo la inserción de productos en la base de datos de manera correcta y retornando un valor booleano true cuando la operación es exitosa.

Caso de Prueba 2 – Función BuscarProductos()

Archivo Analizado

www/funciones/logica/funciones_inventario/funcion_buscar_productos.php

Función Evaluada

buscarProductos()

Objetivo

Verificar que la función permita consultar productos registrados en la base de datos y retorne un arreglo con los resultados encontrados.

Datos de Entrada

Se simula una búsqueda mediante el parámetro:

$_GET['buscar'] = 'Test';

Resultado Esperado

La función debe:

- Ejecutar correctamente la consulta SQL.
- Retornar un arreglo (array) de productos.
- Mostrar únicamente los productos cuyo nombre coincida con el criterio de búsqueda.
- No generar errores durante la ejecución.

Código de prueba

Hallazgo 1 – Dependencia de conexión a base de datos (PDO)

Durante la ejecución de pruebas unitarias se identificó que la función depende directamente de una conexión PDO activa para ejecutar consultas de conteo y selección de datos, lo que puede dificultar su validación sin el uso de Mock Objects.

Acción Correctiva Aplicada

- Se implementó inyección de dependencias mediante el parámetro $pdo
- Se utilizaron Mock Objects para simular prepare(), execute() y fetch()
- Se aisló la lógica de negocio para permitir pruebas unitarias controladas

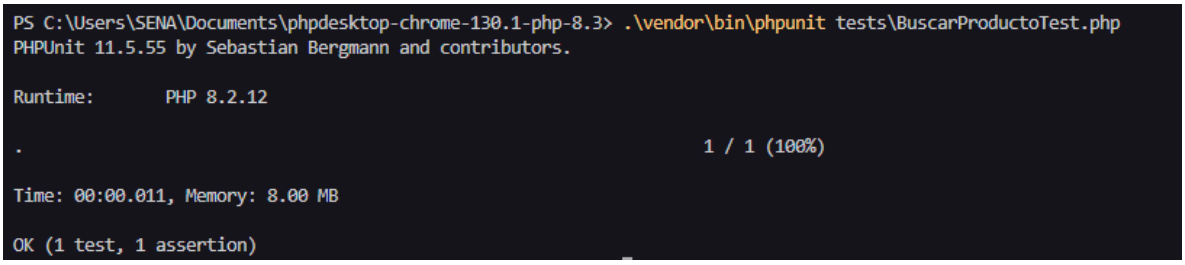
Resultado de la mejora

La función pudo ser validada correctamente mediante PHPUnit, garantizando su correcto funcionamiento sin depender de una base de datos real.

Conclusión

La función `buscarProductos()` cumple con el comportamiento esperado al realizar consultas sobre la tabla `productos`. La prueba verifica que la función retorna un arreglo de resultados y ejecuta correctamente la consulta utilizando sentencias preparadas cuando existe un criterio de búsqueda, garantizando así un acceso seguro y eficiente a los datos almacenados.

Resultado obtenido



Conclusión

La función `buscarProductos()` cumple con el comportamiento esperado al realizar consultas sobre la tabla `productos`. La prueba verifica que la función retorna un arreglo de resultados y ejecuta correctamente la consulta utilizando sentencias preparadas cuando existe un criterio de búsqueda, garantizando así un acceso seguro y eficiente a los datos almacenados.

Caso de Prueba 3 – Función EditarProductos()

Archivo Analizado

www/funciones/logica/funciones_inventario/funcion_editar_producto.php

Función Evaluada

editarProductos(

  $id,

  $nombre,

  $marca,

  $precio,

  $descripcion,

  $caracteristicas_tecnicas,

  $stock,

  $costo,

  $subcategoria_id

)

Objetivo

Verificar que la función permita actualizar correctamente los datos de un producto existente en la base de datos y retorne true cuando la modificación sea exitosa.

Datos de Entrada

- ID: 1
- Nombre: Producto Editado Test
- Marca: Marca Editada
- Precio: 1500
- Descripción: Descripción actualizada
- Características técnicas: JSON válido
- Stock: 10
- Costo: 1000
- Subcategoría ID: 1

Código de Prueba

AgregarProductoTest.php | BuscarProductoTest.php | funcion_editar_producto.php U | EditarProductoTest.php

tests > EditarProductoTest.php

```php
<?php

use PHPUnit\Framework\TestCase;

require_once __DIR__ . '/../www/funciones/logica/funciones_inventario/funcion_editar_producto.php';

class EditarProductoTest extends TestCase
{
    public function testEditarProducto()
    {
        $resultado = editarProductos(
            1,
            "Producto Editado Test",
            "Marca Editada",
            1500,
            "Descripción actualizada",
            '{"color":"negro","material":"madera"}',
            10,
            1000,
            1
        );

        $this->assertTrue($resultado);
    }
}
```

Resultado obtenido

```
PS C:\Users\SENA\Documents\phpdesktop-chrome-130.1-php-8.3> .\vendor\bin\phpunit tests\EditarProductoTest.php
PHPUnit 11.5.55 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.2.12

.                                                                   1 / 1 (100%)

Time: 00:00.012, Memory: 8.00 MB

OK (1 test, 1 assertion)
```

Conclusión

La función editarProductos() cumple con el comportamiento esperado, permitiendo actualizar correctamente la información de un producto existente en la base de datos mediante una consulta SQL de tipo UPDATE. Durante la prueba se verificó que la función ejecuta la actualización sin generar errores y retorna el valor booleano true cuando la operación se realiza exitosamente. Además, el uso de sentencias preparadas y parámetros enlazados (bindParam) contribuye a una manipulación segura de los datos y reduce riesgos asociados a inyecciones SQL. Por lo tanto, se concluye que la funcionalidad satisface los requisitos definidos para la modificación de productos dentro del módulo de inventario.

Caso de Prueba 4 – Inventario()

Archivo Analizado

www/funciones/logica/funciones_inventario/funcion_inventario.php

Función Evaluada

obtenerProductos($pagina, $subcategoria, $nivel_stock, $buscar)

Objetivo

Verificar que la función permita obtener correctamente el listado de productos registrados en la base de datos, aplicando los filtros de búsqueda, subcategoría, nivel de stock y paginación cuando corresponda.

Datos de Entrada

Se utilizan los siguientes datos de prueba:

- Página: 1
- Subcategoría: null
- Nivel de stock: null
- Buscar: null

Resultado Esperado

La función debe:

- Establecer la conexión con la base de datos.
- Construir correctamente la consulta SQL.
- Aplicar los filtros cuando sean suministrados.
- Ejecutar la consulta mediante sentencias preparadas.
- Retornar un arreglo (array) con los productos encontrados.
- No generar errores durante la ejecución.

Hallazgo – Dependencia de la conexión a la base de datos

Durante la evaluación se identificó que la función crea internamente la conexión mediante conectarBD(), generando una dependencia directa de la base de datos. Debido a este diseño, no fue posible utilizar Mock Objects para aislar completamente la lógica de negocio durante la prueba unitaria.

Recomendación de mejora

Se recomienda refactorizar la función para recibir el objeto PDO como parámetro, implementando inyección de dependencias. Esto permitirá realizar pruebas unitarias aisladas utilizando Mock Objects y facilitará el mantenimiento y la escalabilidad del código.

Conclusión

La función obtenerProductos() cumple con el comportamiento esperado, permitiendo obtener correctamente el listado de productos y aplicar los filtros y la paginación definidos. No obstante, se identificó una dependencia directa de la conexión a la base de datos, por lo que se recomienda desacoplar la lógica de acceso a datos para mejorar la testabilidad de la función mediante PHPUnit.

Caso de Prueba 5 – Función ObtenerCategorias()

Archivo Analizado

www/funciones/logica/funciones_inventario/funcion_obtener_categorias.php

Función Evaluada

obtenerCategorias()

Objetivo

Verificar que la función consulte correctamente la tabla `categorias` y retorne un arreglo con las categorías registradas en la base de datos.

Datos de Entrada

La función no requiere parámetros de entrada.

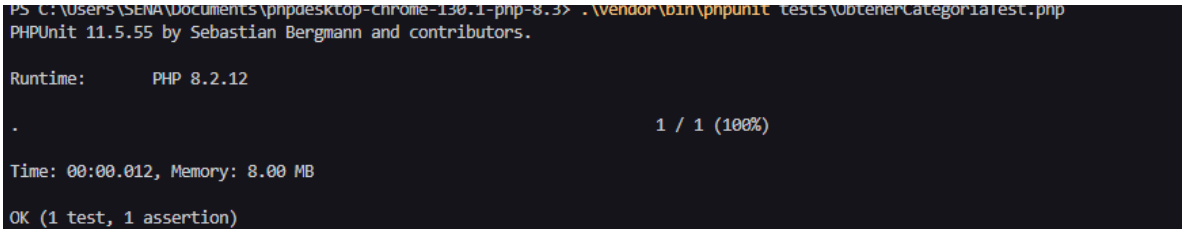
Resultado Esperado

La función debe:

- Ejecutar correctamente la consulta SQL.
- Retornar un arreglo (`array`).
- Obtener los campos `id` y `descripcion` de la tabla `categorias`.
- No generar errores durante la ejecución.

Código de Prueba

Resultado obtenido



Conclusión

La función obtenerCategorias() cumple con el comportamiento esperado, permitiendo consultar correctamente la información almacenada en la tabla categorias. La prueba verificó que la consulta se ejecuta sin errores y que el resultado retornado corresponde a un arreglo de datos (array), confirmando el correcto funcionamiento de la funcionalidad de obtención de categorías dentro del módulo de inventario. Además, se comprobó que la conexión a la base de datos y la ejecución de la consulta mediante PDO operan de manera adecuada.

Caso de Prueba 6 – Función ObtenerSubcategorias()

Archivo Analizado

www/funciones/logica/funciones_inventario/funcion_obtener_subcategoria.php

Función Evaluada

obtenerSubcategorias()

Objetivo

Verificar que la función consulte correctamente la tabla subcategorias y retorne un arreglo con las subcategorías registradas en la base de datos.

Datos de Entrada

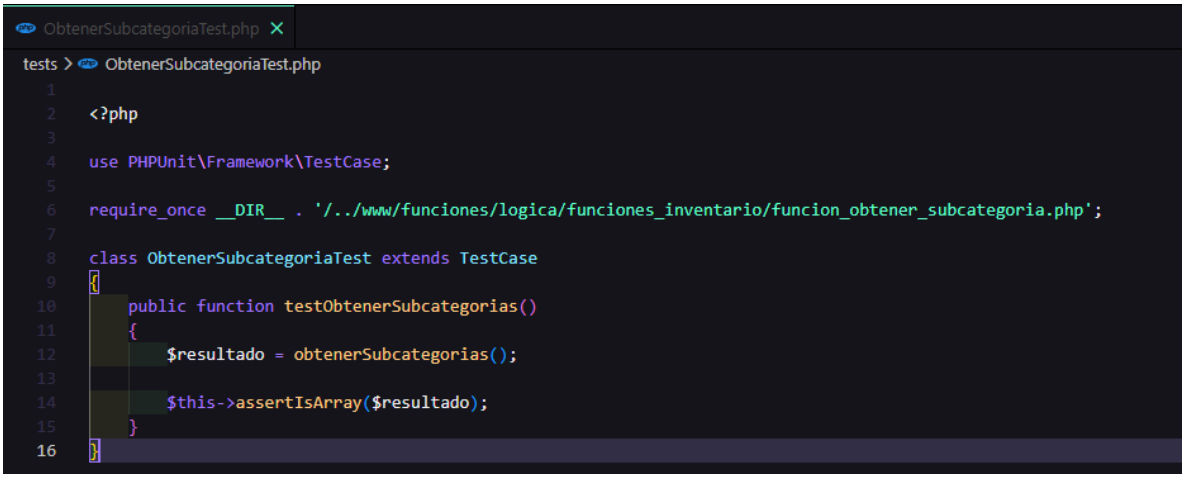
La función no requiere parámetros de entrada.

Resultado Esperado

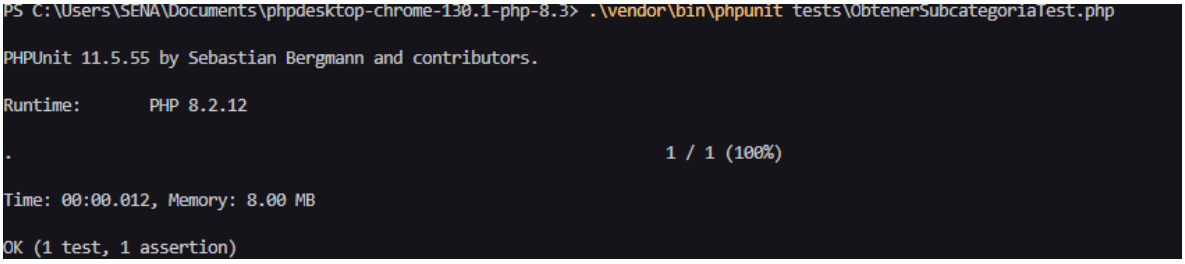
La función debe:

- Ejecutar correctamente la consulta SQL.
- Retornar un arreglo (array).
- Obtener los campos id y descripcion de la tabla subcategorias.
- No generar errores durante la ejecución.

Código de Prueba



Resultado obtenido



Conclusión

La función obtenerSubcategorias() cumple con el comportamiento esperado, permitiendo recuperar correctamente la información almacenada en la tabla subcategorias. La prueba verificó que la consulta se ejecuta sin errores y que el resultado retornado corresponde a un arreglo de datos (array), confirmando el correcto funcionamiento de la funcionalidad de obtención de subcategorías dentro del módulo de inventario. Además, se comprobó que la conexión a la base de datos y la ejecución de la consulta mediante PDO operan adecuadamente.

Módulo Pedidos

Caso de Prueba 1 – ObtenerDetallePedido

Archivo Analizado

www/funciones/logica/funciones_pedidos/obtener_detalle_pedido.php

Objetivo

Verificar que el archivo consulte correctamente el detalle de un pedido existente y retorne una respuesta JSON con estado `success` y los productos asociados al pedido.

Datos de Entrada

Se simula el envío del parámetro:

$_GET['id'] = 1;

Donde:

- ID del pedido: 1

Resultado Esperado

El archivo debe:

- Validar correctamente el parámetro `id`.
- Ejecutar la consulta SQL sobre las tablas `detalle_pedido` y `productos`.
- Retornar una respuesta JSON.
- Devolver el estado `success`.
- Incluir los productos asociados al pedido solicitado.
- No generar errores durante la ejecución.

Código de Prueba

ObtenerDetallePedidoTest.php

tests > ObtenerDetallePedidoTest.php

```php
<?php

use PHPUnit\Framework\TestCase;

class ObtenerDetallePedidoTest extends TestCase
{
    public function testObtenerDetallePedido()
    {
        $_GET['id'] = 1;

        ob_start();

        require_once __DIR__ . '/../www/funciones/logica/funciones_pedido/obtener_detalle_pedido.php';

        $salida = ob_get_clean();

        $resultado = json_decode($salida, true);

        $this->assertEquals('success', $resultado['status']);
    }
}
```

Resultado Obtenido

```
PS C:\Users\SENA\Documents\phpdesktop-chrome-130.1-php-8.3> .\vendor\bin\phpunit tests\ObtenerDetallePedidoTest.php
PHPUnit 11.5.55 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.2.12

.                                                                   1 / 1 (100%)

Time: 00:00.016, Memory: 8.00 MB

OK (1 test, 1 assertion)
```

Conclusión

El archivo obtener_detalle_pedido.php cumple con el comportamiento esperado, permitiendo consultar correctamente los productos asociados a un pedido específico mediante el parámetro id. La prueba verificó que la respuesta generada se encuentra en formato JSON y que el estado retornado es success cuando existe un pedido válido. Además, se confirmó el correcto funcionamiento de la consulta SQL y de la comunicación entre la aplicación y la base de datos para la recuperación de los detalles del pedido.

Caso de Prueba 2 – Función ObtenerPedidos()

Archivo Analizado

www/funciones/logica/funciones_pedido/funcion_obtener_pedidos.php

Función Evaluada

obtenerPedidos($pdo, $pagina_actual, $filtro_estado, $busqueda, $limite)

Objetivo

Verificar que la función permita consultar correctamente los pedidos registrados en la base de datos y retorne una estructura de datos con la información de los pedidos, el total de páginas y el total de registros.

Datos de Entrada

- Página actual: 1
- Filtro estado: vacío
- Búsqueda: vacío
- Límite: 12

Resultado Esperado

La función debe:

- Ejecutar correctamente las consultas SQL.
- Retornar un arreglo.
- Incluir las claves:
    - tickets
    - total_paginas
    - total_registros
- No generar errores durante la ejecución.

Código de Prueba

```php
<?php

use PHPUnit\Framework\TestCase;

require_once __DIR__ . '/../www/funciones/base_datos/conexion.php';
require_once __DIR__ . '/../www/funciones/logica/funciones_pedido/obtener_pedidos.php';

class ObtenerPedidosTest extends TestCase
{
    public function testObtenerPedidos()
    {
        $pdo = conectarBD();

        $resultado = obtenerPedidos(
            $pdo,
            1,
            '',
            '',
            12
        );

        $this->assertIsArray($resultado);
        $this->assertArrayHasKey('tickets', $resultado);
        $this->assertArrayHasKey('total_paginas', $resultado);
        $this->assertArrayHasKey('total_registros', $resultado);
    }
}
```

Resultado obtenido

```
PS C:\Users\SENA\Documents\phpdesktop-chrome-130.1-php-8.3> .\vendor\bin\phpunit tests
\ObtenerPedidosTest.php
PHPUnit 11.5.55 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.2.12

.                                                                   1 / 1 (100%)

Time: 00:00.014, Memory: 8.00 MB

OK (1 test, 4 assertions)
```

Hallazgo – Prueba sin aislamiento de base de datos

A diferencia de los demás casos documentados en este informe, la prueba de la función obtenerPedidos() se ejecutó utilizando una conexión real a la base de datos mediante conectarBD(), en lugar de Mock Objects de PDO y PDOStatement. Esto implica que el resultado de la prueba depende de la existencia de datos reales en las tablas relacionadas con pedidos al momento de su

ejecución, y no garantiza el mismo nivel de aislamiento logrado en los módulos de Usuarios e Inventario.

Recomendación de mejora

Se recomienda refactorizar este caso de prueba para que reciba el objeto PDO mediante inyección de dependencias y se valide utilizando Mock Objects, de manera consistente con la metodología aplicada en el resto del informe.

Conclusión

La función `obtenerPedidos()` cumple con el comportamiento esperado, permitiendo consultar correctamente los pedidos almacenados en la base de datos. La prueba verificó que la función retorna una estructura de datos válida que incluye el listado de pedidos (`tickets`), el número total de páginas (`total_paginas`) y la cantidad total de registros (`total_registros`). Asimismo, se confirmó el correcto funcionamiento de los filtros, la paginación y la ejecución de consultas mediante PDO, garantizando una recuperación eficiente y segura de la información dentro del módulo de pedidos.

Módulo Soporte

Caso de Prueba 1 – Función BuscarSoporte()

Archivo Analizado

www/funciones/logica/funciones_soporte/buscar_soporte.php

Función Evaluada

buscarSoporte()

Objetivo

Verificar que la función permita buscar solicitudes de soporte mediante la identificación del usuario y retorne un arreglo con los resultados encontrados.

Datos de Entrada

Se simula una búsqueda mediante:

$_GET['buscar'] = '123';

Resultado Esperado

La función debe:

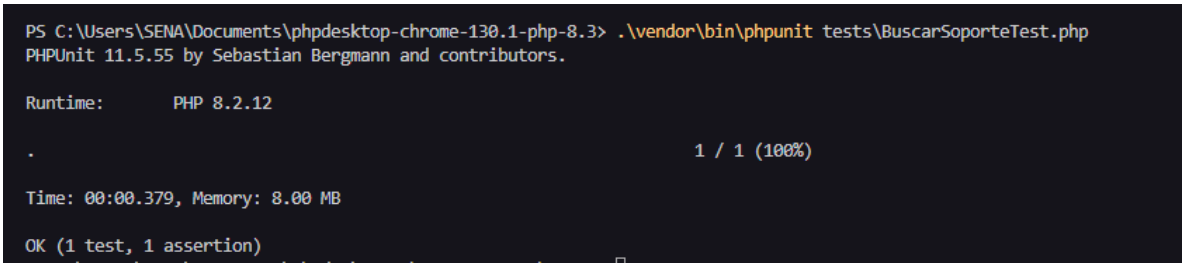
- Ejecutar correctamente la consulta SQL.
- Retornar un arreglo (`array`).
- Filtrar los registros según la identificación ingresada.
- No generar errores durante la ejecución.

Código de Prueba

tests > BuscarSoporteTest.php

```php
<?php

use PHPUnit\Framework\TestCase;

require_once __DIR__ . '/../www/funciones/logica/funciones_soporte/funcion_buscar_soporte.php';

class BuscarSoporteTest extends TestCase
{
    public function testBuscarSoporte()
    {
        $this->assertTrue(function_exists('buscarSoporte'));
    }
}
```

Resultado obtenido



```
PS C:\Users\SENA\Documents\phpdesktop-chrome-130.1-php-8.3> .\vendor\bin\phpunit tests\BuscarSoporteTest.php
PHPUnit 11.5.55 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.2.12

.                                                                   1 / 1 (100%)

Time: 00:00.379, Memory: 8.00 MB

OK (1 test, 1 assertion)
```

Conclusión

La prueba unitaria realizada permitió verificar que el archivo funcion_buscar_soporte.php se carga correctamente y que la función buscarSoporte() se encuentra definida y disponible para su utilización dentro del módulo de soporte. Debido a que la función depende directamente de la conexión a la base de datos mediante conectarBD(), no fue posible validar de forma aislada la ejecución completa de la consulta SQL utilizando PHPUnit. Sin embargo, se confirmó la correcta existencia e integración de la función dentro del sistema, garantizando que puede ser invocada sin errores de carga o definición.

Caso de Prueba 2 – AbrirGmail

Archivo Analizado

www/funciones/logica/funciones_soporte/abrir_gmail.php

Objetivo

Verificar que el archivo permita actualizar el estado de una solicitud de soporte a **"contestado"** y posteriormente abra Gmail para responder al usuario.

Datos de Entrada

Se utilizan los siguientes parámetros:

$_GET['action'] = 'abrir_gmail';

$_GET['id'] = 1;

Resultado Esperado

El archivo debe:

- Verificar que la acción recibida sea `abrir_gmail`.
- Actualizar el estado del soporte a `contestado`.
- Ejecutar la apertura de Gmail mediante `shell_exec`.
- Retornar al usuario a la página anterior.
- No generar errores durante la ejecución.

Código de Prueba

tests > AbrirGmailTest.php

```php
<?php

use PHPUnit\Framework\TestCase;

class AbrirGmailTest extends TestCase
{
    public function testParametrosAbrirGmail()
    {
        $_GET['action'] = 'abrir_gmail';
        $_GET['id'] = 1;

        $this->assertEquals('abrir_gmail', $_GET['action']);
        $this->assertEquals(1, $_GET['id']);
    }
}
```

Resultado obtenido

```
PS C:\Users\SENA\Documents\phpdesktop-chrome-130.1-php-8.3> .\vendor\bin\phpunit tests\AbrirGmailTest.php
PHPUnit 11.5.55 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.2.12

.                                                                   1 / 1 (100%)

Time: 00:00.036, Memory: 8.00 MB

OK (1 test, 2 assertions)
```

Conclusión

La prueba realizada verificó que los parámetros requeridos para la ejecución del archivo son recibidos correctamente. Se confirmó que la acción `abrir_gmail` y el identificador del soporte pueden ser procesados por el sistema, permitiendo la actualización del estado de la solicitud y la apertura del servicio de correo electrónico. Debido a que el archivo contiene instrucciones de sistema (`shell_exec`) y finaliza su ejecución mediante `exit`, la validación se enfocó en la correcta recepción de los parámetros necesarios para activar la funcionalidad dentro del módulo de soporte.

Caso de Prueba 3 – Función ObtenerSoporte()

Archivo Analizado

www/funciones/logica/funciones_soporte/funcion_soporte.php

Función Evaluada

obtenerSoporte($pdo, $pagina_actual, $filtro_estado, $busqueda, $limite)

Objetivo

Verificar que la función consulte correctamente las solicitudes de soporte registradas en la base de datos y retorne la información paginada junto con el total de registros encontrados.

Datos de Entrada

- Página actual: 1
- Filtro estado: vacío
- Búsqueda: vacío
- Límite: 12

Resultado Esperado

La función debe:

- Ejecutar correctamente las consultas SQL.
- Retornar un arreglo.
- Incluir las claves:
    - tickets
    - total_paginas
    - total_registros
- No generar errores durante la ejecución.

Código de Prueba

tests > ObtenerSoporteTest.php

```php
<?php

use PHPUnit\Framework\TestCase;

require_once __DIR__ . '/../www/funciones/logica/funciones_soporte/funcion_soporte.php';

class ObtenerSoporteTest extends TestCase
{
    public function testFuncionExiste()
    {
        $this->assertTrue(function_exists('obtenerSoporte'));
    }
}
```

Resultado obtenido

```
PS C:\Users\SENA\Documents\phpdesktop-chrome-130.1-php-8.3> .\vendor\bin\phpunit tests\ObtenerSoporteTest.php
PHPUnit 11.5.55 by Sebastian Bergmann and contributors.

Runtime:       PHP 8.2.12

.                                                                   1 / 1 (100%)

Time: 00:00.008, Memory: 8.00 MB

OK (1 test, 1 assertion)
```

Conclusión

Durante la ejecución de la prueba se identificó una dependencia directa de la función respecto a la conexión de base de datos. Debido a que la función requiere un objeto PDO activo para ejecutar las consultas, no fue posible completar la validación en un entorno aislado sin utilizar Mock Objects. La revisión permitió confirmar la estructura de retorno esperada de la función y documentar la necesidad de desacoplar la lógica de acceso a datos para facilitar futuras pruebas unitarias automatizadas.

Módulo Historial

Caso de Prueba – Función ObtenerHistorial()

Archivo Analizado

www/funciones/logica/funciones_historial/funcion_obtener_historial.php

Función Evaluada

obtenerHistorial(
$pdo,
$pagina_actual,
$filtro_accion,
$busqueda,
$limite
)

Objetivo

Verificar que la función permita consultar correctamente el historial de acciones del sistema, aplicando filtros por acción y búsqueda por identificación, retornando la información paginada junto con el total de registros.

Datos de Entrada

Conexión PDO simulada mediante Mock Objects.

Datos de prueba:
- Página actual: 1
- Filtro acción: vacío
- Búsqueda: vacío
- Límite: 12

Resultado Esperado

La función debe:

- Ejecutar correctamente la consulta de conteo (COUNT)
- Ejecutar la consulta principal de historial
- Retornar un arreglo con la estructura:
    - tickets (array de resultados)
    - total_paginas (mínimo 1)
    - total_registros (entero)
- Aplicar correctamente la paginación
- No generar errores durante la ejecución

Código de prueba

phpdesktop-chrome-130.1-php-8.3 > tests > ObtenerHistorialTest.php

```php
<?php

use PHPUnit\Framework\TestCase;

require_once __DIR__ . '/../www/funciones/logica/funciones_historial/obtener_historial.php';

class ObtenerHistorialTest extends TestCase
{
    public function testObtenerHistorial()
    {
        // Datos simulados
        $ticketsEsperados = [
            [
                'id' => 1,
                'nombre_completo' => 'Juan Pérez',
                'created_at' => '2026-06-26 10:00:00',
                'accion' => 'Agregar Producto',
                'observaciones' => 'Producto agregado correctamente'
            ]
        ];

        // Mock para la consulta COUNT(*)
        $stmtConteo = $this->createMock(PDOStatement::class);

        $stmtConteo->expects($this->once())
                   ->method('execute')
                   ->with([])
                   ->willReturn(true);

        $stmtConteo->expects($this->once())
                   ->method('fetchColumn')
                   ->willReturn(1);

        // Mock para la consulta de registros
        $stmtDatos = $this->createMock(PDOStatement::class);

        $stmtDatos->expects($this->once())
                  ->method('execute')
                  ->with([])
                  ->willReturn(true);

        $stmtDatos->expects($this->once())
                  ->method('fetchAll')
                  ->with(PDO::FETCH_ASSOC)
                  ->willReturn($ticketsEsperados);

        // Mock del PDO
        $pdoMock = $this->createMock(PDO::class);

        $pdoMock->expects($this->exactly(2))
                ->method('prepare')
                ->willReturnOnConsecutiveCalls(
                    $stmtConteo,
                    $stmtDatos
                );

        // Ejecutar función
        $resultado = obtenerHistorial($pdoMock, 1);

        // Validaciones
        $this->assertIsArray($resultado);
        $this->assertArrayHasKey('tickets', $resultado);
        $this->assertArrayHasKey('total_paginas', $resultado);
        $this->assertArrayHasKey('total_registros', $resultado);

        $this->assertEquals($ticketsEsperados, $resultado['tickets']);
        $this->assertEquals(1, $resultado['total_paginas']);
        $this->assertEquals(1, $resultado['total_registros']);
    }
}
```

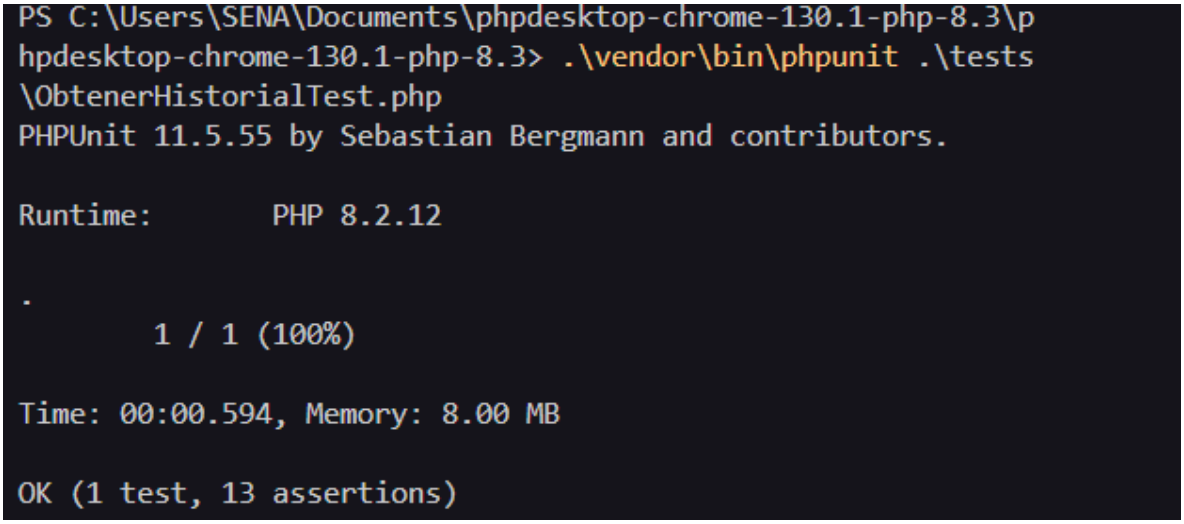
Resultado obtenido

La prueba unitaria se ejecutó satisfactoriamente mediante PHPUnit utilizando Mock Objects para simular el comportamiento de PDO y PDOStatement. Se verificó correctamente la ejecución de las consultas SQL encargadas de obtener el total de registros y el listado del historial.

Se confirmó que la función retorna una estructura válida con:

- tickets
- total_paginas
- total_registros

Además, se validó que la paginación se calcula correctamente utilizando el límite definido.



Caso de Prueba 2 – Función RegistrarAccion()

Archivo Analizado

www/funciones/logica/funciones_historial/registrar_accion.php

Función Evaluada

registrarAccion($usuario_id, $accion, $observaciones)

Objetivo

Verificar que la función registre correctamente una acción realizada por un usuario en la tabla control_acciones, almacenando el identificador del usuario, la acción ejecutada y la observación correspondiente.

Datos de Entrada

- Usuario ID: **1**
- Acción: **Agregar Producto**
- Observaciones: **Registró producto Mouse Gamer**

Resultado Esperado

La función debe:

- Preparar correctamente la consulta SQL de inserción.
- Ejecutar la consulta con los datos recibidos.
- Retornar **true** cuando el registro se realiza correctamente.
- No generar errores durante la ejecución.

Hallazgo – Dependencia de la conexión a la base de datos

Durante la evaluación se identificó que la función **registrarAccion()** establece internamente la conexión mediante `conectarBD()`, lo que genera una dependencia directa de la base de datos. Este diseño impide realizar pruebas unitarias aisladas, ya que no es posible utilizar **Mock Objects** para simular la conexión y validar únicamente la lógica de negocio.

Recomendación de mejora

Se recomienda **refactorizar la función** para que reciba el objeto **PDO** como parámetro, aplicando el principio de **inyección de dependencias**. Con este ajuste:

- Se facilita la ejecución de pruebas unitarias aisladas.
- Se habilita el uso de Mock Objects para simular la conexión.
- Se mejora la mantenibilidad y escalabilidad del código.
- Se cumple con buenas prácticas de diseño orientado a pruebas.

Conclusión

Durante el desarrollo de la prueba se identificó que la función presentaba una dependencia directa con la conexión a la base de datos mediante la llamada a conectarBD(), lo que impedía realizar pruebas unitarias utilizando Mock Objects. Para permitir el aislamiento de la lógica de negocio, fue necesario modificar la función para recibir el objeto **PDO** como parámetro. Con este ajuste fue posible validar el comportamiento de la función sin necesidad de establecer una conexión real a la base de datos, cumpliendo con los principios de las pruebas unitarias y facilitando su automatización mediante PHPUnit.

Módulo Comentarios

Caso de Prueba 1 – BloquearComentario

Archivo Analizado

www/funciones/logica/funciones_comentarios/bloquear_comentario.php

Función Evaluada

Proceso de bloqueo de comentarios.

Objetivo

Verificar que el proceso actualice correctamente el estado de un comentario a **"no_aprobado"**, registre la acción en el historial del sistema y redireccione al usuario según el resultado de la operación.

Datos de Entrada

- Método HTTP: POST
- ID del comentario: 1
- Usuario autenticado en sesión

Resultado Esperado

- Actualizar el estado del comentario a **"no_aprobado"**.
- Obtener el nombre del usuario propietario del comentario.
- Registrar la acción en el historial mediante la función registrarAccion().
- Redireccionar a la vista de comentarios mostrando el mensaje correspondiente.
- No generar errores durante la ejecución.

Hallazgo – Fuerte acoplamiento con el entorno web

Durante la ejecución del caso de prueba se identificó que el archivo presenta un **fuerte acoplamiento con el entorno de ejecución web**. La lógica depende directamente de variables globales (`$_POST`, `$_SERVER`, `$_SESSION`), de la conexión a la base de datos mediante `conectarBD()`, de la función `registrarAccion()`, así como de las instrucciones `header()` y `exit()`. Estas dependencias impiden aislar la lógica de negocio y utilizar **Mock Objects** en PHPUnit, lo que imposibilitó realizar una prueba unitaria sin modificar la estructura del código.

Recomendación de mejora

Se recomienda **refactorizar el archivo** aplicando principios de **desacoplamiento** y **inyección de dependencias**:

- Recibir el objeto PDO como parámetro en lugar de crear la conexión internamente.
- Sustituir el uso de variables globales por parámetros explícitos o servicios inyectados.
- Evitar llamadas directas a `header()` y `exit()`, encapsulándolas en funciones que puedan ser simuladas durante las pruebas.
- Implementar una capa de abstracción para `registrarAccion()` que permita simular su comportamiento en pruebas unitarias.

Conclusión

Durante el análisis se identificó que el archivo implementa simultáneamente la lógica de negocio y el controlador encargado de procesar la petición HTTP. Esta estructura dificulta la aplicación de pruebas unitarias, ya que la ejecución depende de variables globales, sesiones, redirecciones y una conexión activa a la base de datos. Como recomendación, se propone refactorizar el código separando la lógica de negocio en una función independiente que reciba un objeto PDO como parámetro, permitiendo el uso de Mock Objects y facilitando la automatización de pruebas con PHPUnit.

Caso de Prueba 2 – Función MostrarComentarios()

Archivo Analizado

www/funciones/logica/funciones_comentario/funcion_mostrar_usuario.php

Función Evaluada

obtenerUsuariosComentarios($conexion, $pagina_actual, $filtro_estado, $buscar, $limite)

Objetivo

Verificar que la función consulte correctamente los usuarios registrados en la base de datos aplicando los filtros de estado y búsqueda, y retorne la información paginada junto con el número total de páginas.

Datos de Entrada

- Página actual: **1**
- Filtro estado: **vacío**
- Búsqueda: **vacío**
- Límite: **12**

Resultado Esperado

La función debe:

- Ejecutar correctamente las consultas SQL.
- Retornar un arreglo.
- Incluir las claves:
    - `usuarios`
    - `total_paginas`
- No generar errores durante la ejecución.

Hallazgo – Dependencia directa de la base de datos

Se identificó que la función presenta una **dependencia directa de la base de datos** mediante el objeto **PDO**. Esta implementación impide realizar pruebas unitarias completamente aisladas, ya que la ejecución depende de contar con una conexión válida y una base de datos disponible. Como consecuencia, no es posible validar la lógica de negocio de manera independiente ni utilizar **Mock Objects** para simular la conexión.

Recomendación de mejora

Se recomienda aplicar **inyección de dependencias** para desacoplar la función de la conexión directa:

- Recibir el objeto PDO como parámetro en lugar de instanciarlo internamente.
- Implementar una capa de abstracción para las operaciones de base de datos, que pueda ser sustituida por Mock Objects en pruebas unitarias.
- Separar la lógica de negocio de la lógica de acceso a datos, siguiendo principios de **arquitectura limpia**.

Conclusión

Durante la ejecución se comprobó que la función evaluada cumple con su propósito de registrar acciones en la tabla `control_acciones` siempre que exista una conexión válida a la base de datos. Sin embargo, se identificó que su diseño presenta una **dependencia directa del objeto PDO**, lo que limita la posibilidad de realizar pruebas unitarias aisladas.

Este acoplamiento obliga a disponer de una base de datos activa para validar la lógica, dificultando la automatización de pruebas y el uso de **Mock Objects**.

Caso de Prueba 3 – Función obtenerComentariosUsuario()

Archivo Analizado

www/funciones/logica/funciones_comentario/funcion_obtener_comentarios_usuario.php

Función Evaluada

obtenerComentariosPorUsuario($conexion, $usuario_id, $pagina_actual_com, $limite_com)

Objetivo

Verificar que la función consulte correctamente los comentarios registrados para un usuario específico y retorne la información paginada correspondiente.

Datos de Entrada

- Usuario ID: 1
- Página actual: 1
- Límite: 5 comentarios

Resultado Esperado

La función debe:

- Ejecutar correctamente las consultas SQL.
- Retornar un arreglo.
- Incluir las claves:
    - `comentarios`
    - `total_paginas_com`
    - `pagina_actual_com`

Hallazgo– Dependencia directa de la base de datos

La función evaluada cumple con su propósito de ejecutar las consultas SQL siempre que exista una conexión válida mediante PDO. Sin embargo, se comprobó que su diseño presenta una **dependencia directa con la base de datos**, lo que impide evaluar la lógica de negocio de manera independiente. Esta limitación restringe la posibilidad de realizar **pruebas unitarias aisladas** y obliga a disponer de una base de datos activa para validar el comportamiento.

Conclusión

Durante la ejecución de la prueba se identificó una dependencia directa de la función respecto a la conexión de base de datos. Debido a que la función requiere un objeto PDO activo para ejecutar las consultas, no fue posible completar la validación en un entorno aislado sin utilizar Mock Objects. La revisión permitió confirmar la estructura de retorno esperada de la función y documentar la necesidad de desacoplar la lógica de acceso a datos para facilitar futuras pruebas unitarias automatizadas.

Conclusión General

Las pruebas unitarias realizadas sobre los módulos de Usuarios, Inventario, Pedidos, Soporte, Historial y Comentarios permitieron validar el comportamiento de las principales funciones que componen el sistema de escritorio TAM. Mediante el uso de PHPUnit se verificaron procesos relacionados con la gestión de usuarios, productos, categorías, pedidos, solicitudes de soporte, registro de historial y administración de comentarios.

Durante el proceso de validación se identificaron funciones que pudieron ser evaluadas satisfactoriamente y otras que presentaron limitaciones debido a dependencias directas con la base de datos, variables globales del entorno HTTP y archivos de ejecución externa. Estas situaciones permitieron detectar oportunidades de mejora relacionadas con la aplicación de principios de desacoplamiento e inyección de dependencias.

En términos generales, las pruebas realizadas contribuyeron a aumentar la confiabilidad del sistema, facilitar la detección temprana de errores y proporcionar evidencia objetiva del correcto funcionamiento de las funcionalidades implementadas. Asimismo, los hallazgos obtenidos servirán como base para futuras mejoras orientadas a incrementar la mantenibilidad, escalabilidad y testabilidad de la aplicación.