**Entrega 06**

**Documentación de Pruebas Unitarias**

**Módulo Control de Acceso**

## 1. Información general

| **Ítem** | **Detalle** |
|---|---|
| Aplicación | `apps.control_acceso` (Control de Acceso) |
| Framework de tests | `Django TestCase` (unittest) |
| Total de pruebas | 22 |
| Estado | Todas las pruebas pasan rata |
| Tiempo de corrida | 0.024 s |

Estas pruebas cubren tres bloques principales:

1. Asistencia / Control de acceso físico
2. Gestión de rutinas de entrenamiento
3. Validación de membresías

## 2. Bloque 1: Servicio de Asistencia (Control de Acceso)

**Archivo:** *`apps/control_acceso/tests/test_servicio_asistencia.py`*
**Clase:** *`AsistenciaServicioTest`*
**Módulo probado:** *`apps.control_acceso.servicios.asistencia_service`*

### 2.1 Resumen de casos

| **Test** | **Qué verifica** | **Resultado esperado** |
|---|---|---|

| | | |
|---|---|---|
| `test_registrar_entrada_exitosa` | Registro de entrada de un socio con membresía activa | Se crea un registro de `Asistencia` con entrada no nula, sin salida y con terminal de acceso correcto |
| `test_registrar_entrada_membresia_inactiva` | Bloqueo de entrada cuando la membresía no está activa | Se lanza `ValidationError` con mensaje que incluye “no está activa” y no se crea asistencia |
| `test_registrar_entrada_duplicada` | Evitar una segunda entrada con una asistencia activa sin salida | Se lanza `ValidationError` con mensaje que incluye “entrada activa” y solo existe un registro |
| `test_registrar_salida_exitosa` | Registro correcto de la salida del socio | `FechaHoraSalida` se completa y es mayor que `FechaHoraEntrada` |
| `test_registrar_salida_duplicada` | Evitar registrar salida dos veces sobre la misma asistencia | Se lanza `ValidationError` con mensaje del tipo “ya tiene una salida registrada” |
| `test_obtener_asistencia_activa` | Consulta de la asistencia activa de un socio (sin salida registrada) | Antes de la salida, devuelve la asistencia; después de registrar la salida, ya no devuelve registros |

**2.2 Casos límite cubiertos en este bloque**

- Solo se permite entrada si la membresía está ACTIVA.
- No se permite más de una entrada abierta por socio (entrada sin salida).

- El sistema garantiza que la hora de salida sea posterior a la de entrada.
- La consulta de asistencia activa diferencia entre:
    - Socio dentro del gimnasio (asistencia sin salida).
    - Socio fuera del gimnasio (todas las asistencias completadas).

### 3. Bloque 2: Servicio de Rutinas

**Archivo:** *apps/control_acceso/tests/test_servicio_rutina.py*
**Clase:** *RutinaServicioTest*
**Módulo probado:** *apps.control_acceso.servicios.rutina_service*

#### 3.1 Resumen de casos

| Test | Qué verifica | Resultado esperado |
|---|---|---|
| `test_crear_rutina_semanal_exitosa` | Creación de una rutina semanal personalizada para un socio | Se crea una `RutinaSemanal` asociada al socio, con nombre, días de entrenamiento y `EsPlantilla = False` |
| `test_crear_rutina_sin_nombre` | Validación de nombre obligatorio al crear una rutina | Se lanza `ValidationError` con mensaje similar a “el nombre de la rutina es obligatorio” |
| `test_crear_rutina_socio_inexistente` | Bloqueo al crear rutina para un ID de socio que no existe | Se lanza `ValidationError` indicando que el socio especificado no existe |

| | | |
|---|---|---|
| `test_asignar_ejercicio_a_rutina_exitoso` | Asignación de un ejercicio a un día específico de la rutina con sus parámetros completos | Se crea un `DiaRutinaEjercicio` con día correcto, series, repeticiones, tempo y peso objetivo (p.ej. 4x8 a 80 kg) |
| `test_asignar_ejercicio_dia_invalido` | Validación de que el día de la semana está dentro del rango permitido (0–6) | Se lanza `ValidationError` con mensaje que indica que el día debe estar entre 0 y 6 |
| `test_asignar_ejercicio_valores_negativos` | Rechazo de ejercicios con valores negativos en series o repeticiones | Se lanza `ValidationError` con mensaje del tipo “las series deben ser un número positivo” |
| `test_asignar_ejercicio_duplicado_mismo_dia` | Prevención de asignar el mismo ejercicio dos veces el mismo día en la misma rutina | Se lanza `ValidationError` con mensaje tipo “Ya existe este ejercicio asignado” |
| `test_obtener_ejercicios_por_dia` | Consulta de ejercicios asignados por día de la semana | Lunes devuelve 2 ejercicios, miércoles 1 y martes 0, de acuerdo con la configuración de prueba |

**3.2 Casos límite cubiertos en este bloque**

- Nombre de rutina obligatorio (vacío, espacios o nulo no son válidos).
- Integridad referencial: no se permiten rutinas para socios que no existen.
- Rango de día de semana estricto `0−6` (Lunes–Domingo).

- Parámetros de ejercicios (series, repeticiones, peso) deben ser positivos.
- Se permite:
    - Mismo ejercicio en días distintos.
    - Varios ejercicios el mismo día, mientras no se duplique el par (rutina, día, ejercicio).

### 4. Bloque 3: Validaciones de Asistencia / Membresías

**Archivo:** *apps/control_acceso/tests/test_validaciones_asistencia.py*
**Clase:** *ValidacionesAsistenciaTest*
**Módulo probado:** *apps.control_acceso.servicios.asistencia_service*

Aquí se prueban tanto funciones de validación de membresía como métodos del modelo `SocioMembresia`.

#### 4.1 Resumen de casos

| Test | Qué verifica | Resultado esperado |
|---|---|---|
| `test_validacion_membresia_activa_exitosa` | Que una membresía con estado ACTIVA pase la validación | No se lanza `ValidationError` |
| `test_validacion_membresia_morosa_falla` | Que una membresía en estado MOROSA no pase la validación | Se lanza `ValidationError` con mensaje que incluye “no está activa” |
| `test_validacion_membresia_expirada_falla` | Que una membresía en estado EXPIRADA no pase la validación | Se lanza `ValidationError` con mensaje que incluye “no está activa” |
| `test_is_active_method_membresia_activa` | Comportamiento del método `is_active()` para membresías activas | `is_active()` retorna `True` |

| | | |
|---|---|---|
| `test_is_active_method_membresia_morosa` | Comportamiento de `is_active()` para membresías morosas | `is_active()` retorna `False` |
| `test_is_active_method_membresia_expirada` | Comportamiento de `is_active()` para membresías expiradas | `is_active()` retorna `False` |
| `test_remaining_days_membresia_activa` | Cálculo de días restantes de una membresía activa | Para una membresía con `FechaFin = hoy + 355 días`, `remaining_days()` devuelve 355 |
| `test_remaining_days_membresia_expirada` | Cálculo de días restantes en membresías vencidas | Para una membresía con `FechaFin` en el pasado, `remaining_days()` devuelve un valor negativo |

**4.2 Casos límite cubiertos en este bloque**

- Diferentes estados de membresía: `ACTIVA`, `MOROSA`, `EXPIRADA`.
- Membresías:
    - Recién iniciadas.
    - Próximas a vencer.
    - Recién vencidas o vencidas hace tiempo.
- Interpretación de días negativos en `remaining_days()` para que la interfaz pueda mostrar claramente membresías vencidas.

**5. Conclusión**

La suite de pruebas del módulo control_acceso verifica de forma consistente que:

1. El control de acceso al gimnasio solo permite entradas y salidas válidas, ligadas a membresías activas y sin duplicar registros.

2. La gestión de rutinas garantiza datos coherentes (socio existente, nombre obligatorio, parámetros positivos, días válidos y sin ejercicios duplicados en el mismo día).

3. La validación de membresías (estado e información temporal) funciona de manera alineada con las reglas de negocio del sistema.

**Modulo Socios**

**1. Información General**

| Ítem | Detalle |
|---|---|
| **Aplicación** | apps.socios (Módulo de Gestión de Socios) |
| **Framework de tests** | Django TestCase (unittest) |
| **Total de pruebas** | 10 |
| **Estado** | Todas las pruebas pasan |
| **Tiempo de corrida** | < 0.1 s |

Las pruebas unitarias y de integración cubren tres bloques principales, enfocándose en la funcionalidad esencial del sistema: la validación de datos, la creación de registros y el flujo de la interfaz (vista).

**2. Bloque 1: Pruebas de Lógica de Negocio (Casos Límite)**

Archivo: apps/socios/tests/test_validaciones.py

Clase: ValidacionSocioTest

Módulo probado: apps.socios.servicios.registro_db.validate_socio_data

**2.1 Resumen de Casos**

| Test | Qué verifica | Resultado esperado |
|---|---|---|

| | | |
|---|---|---|
| test_falla_password_corto | Validación de la longitud mínima de contraseña (8 caracteres). | Se lanza ValidationError con un mensaje de longitud. |
| test_falla_identificacion_longitud | Validación de la longitud de la identificación (10 dígitos). | Se lanza ValidationError indicando la longitud incorrecta (para 9 dígitos). |
| test_falla_identificacion_no_numerica | Identificación contiene caracteres no numéricos. | Se lanza ValidationError indicando formato numérico. |
| test_falla_telefono_prefijo | Validación de formato de teléfono (debe iniciar con +57 3). | Se lanza ValidationError indicando el prefijo incorrecto. |
| test_validacion_exitosa | Prueba el **Happy Path** con todos los datos correctos. | No se lanza ninguna excepción (ValidationError). |

**2.2 Casos Límite Cubiertos en este Bloque**

Este bloque se centra en probar valores en los extremos o justo fuera del rango permitido (casos límite y de borde).

- **Longitud de Contraseña:** Se prueba el valor de borde (7 caracteres) y el valor límite (8 caracteres) para asegurar la transición de fallo a éxito.
- **Identificación:** Se prueba el valor justo fuera del rango de 10 dígitos (9 y 11 dígitos, o caracteres no válidos).
- **Teléfono:** Se prueba el patrón de inicio (Ej: se usa 300... o +57 4... en lugar de +57 3...) para asegurar la lógica de formato.

### 3. Bloque 2: Pruebas del Servicio y Base de Datos

Archivo: apps/socios/tests/test_servicios.py

Clase: ServicioCreacionSocioTest

Módulo probado: apps.socios.servicios.registro_db.create_socio_from_dict

#### 3.1 Resumen de Casos

| Test | Qué verifica | Resultado esperado |
|---|---|---|
| test_creacion_exitosa_socio | Creación completa de un nuevo socio en la base de datos (DB). | Se crea un objeto Socio y Socio.objects.count() es 1. |
| test_falla_email_duplicado | Bloqueo al intentar registrar un socio con un email ya existente. | Se lanza ValidationError y Socio.objects.count() se mantiene igual. |
| test_falla_identificacion_duplicada | Bloqueo al intentar registrar un socio con identificación ya existente. | Se lanza ValidationError y la DB no cambia. |

#### 3.2 Casos Límite Cubiertos en este Bloque

Este bloque valida la Integridad Referencial y la Unicidad de los datos, que son casos límite críticos a nivel de servicio y BD.

- **Duplicidad:** Probar la creación de un segundo registro con el mismo email o ID es el caso límite de la regla de unicidad.
- **Transaccionalidad:** Se verifica que si falla la validación (por duplicidad), la creación se aborta completamente, manteniendo la consistencia de la base de datos.

### 4. Bloque 3: Pruebas de Flujo HTTP (Vista de Registro)

Archivo: apps/socios/tests/test_vistas.py

Clase: RegisterViewTest

Módulo probado: apps.socios.views.register_view

**4.1 Resumen de Casos**

| **Test** | **Qué verifica** | **Resultado esperado** |
|---|---|---|
| test_get_register_page | Acceso a la vista (GET). | HTTP 200 y se usa la plantilla correcta. |
| test_post_register_success | Registro exitoso (POST con datos válidos). | HTTP 302 (Redirección) a login y se crea un socio. |
| test_post_password_mismatch | Falla por contraseñas diferentes. | HTTP 200, re-renderiza la plantilla y muestra un mensaje de error. |
| test_post_service_validation_error | Falla por error de validación del servicio (ej. contraseña corta). | HTTP 200, re-renderiza y muestra el mensaje de error de la excepción. |

**4.2 Casos Límite Cubiertos en este Bloque**

Esta sección valida los **comportamientos de la interfaz** bajo condiciones límite (éxito/fallo):

- **Transiciones de Estado:** Se verifica que en el caso límite de éxito, la vista transiciona a una redirección (302) y crea un registro. En el caso límite de fallo, se verifica que la vista no crea el registro y se mantiene en la misma página (200) mostrando el error.
- **Manejo de Excepciones:** Se prueba el flujo de manejo de errores donde la vista captura una ValidationError (lanzada desde la capa de servicio) y la convierte correctamente en un mensaje de error para el usuario.

**5. Conclusión**

La suite de pruebas implementada en el módulo socios verifica de forma consistente que las funcionalidades esenciales (como lo requiere la entrega) y las reglas de negocio se cumplen rigurosamente a través de las diferentes capas de la aplicación.

1. **Validación de Datos (Capa de Servicio):** Se garantiza que la función validate_socio_data maneje correctamente todos los casos límite y de borde (contraseñas cortas, identificaciones con longitud incorrecta, formatos de teléfono inválidos) antes de intentar cualquier operación de base de datos.
2. **Integridad de la Base de Datos (Capa de Servicio):** El servicio de creación (create_socio_from_dict) asegura que no se puedan crear registros duplicados (por Email o Identificación) o inconsistentes, manteniendo la unicidad de los datos.
3. **Flujo de la Interfaz (Capa de Vista):** La vista register_view está validada para manejar correctamente las transiciones de estado:
    - Redirige con éxito y con un mensaje afirmativo tras el registro válido.
    - Muestra mensajes de error claros al usuario y permanece en la página de registro (re-renderiza) ante fallos de la vista (Ej: contraseñas no coinciden) o fallos de la capa de servicio (ValidationError).

**Módulo Seguridad**

**1. Información General**

| Ítem | Detalle |
|---|---|
| **Aplicación** | apps.seguridad (Módulo de autenticación y auditoría) |
| **Framework de tests** | Django TestCase (unittest) |
| **Total de pruebas** | 7 |
| **Estado** | Todas las pruebas pasan (OK) |
| **Tiempo de corrida** | 0.02 s |

Las pruebas unitarias del módulo seguridad abarcan tres bloques principales, verificando la funcionalidad esencial del sistema de autenticación y auditoría.

Los bloques verificados incluyen:

1. Lógica de Autenticación
2. Registro de Auditoría
3. Gestión de Cierre de Sesión (Logout)

**2. Bloque 1: Pruebas de Lógica de Autenticación (Casos Límite)**

**Archivo:** `apps/seguridad/tests/test_autenticacion_unit.py`

**Clase:** `AutenticacionUnitTests`

**Módulo probado:** `apps.seguridad.servicios.autenticacion.autenticar_usuario`

### 2.1 Resumen de Casos

| Test | Qué verifica | Resultado esperado |
|---|---|---|
| test_login_exitoso | El flujo completo de autenticación cuando el usuario existe y la contraseña es correcta. | Devuelve el usuario mockeado, se actualiza `UltimoAcceso`, y se registra un evento `LOGIN_EXITOSO`. |
| test_login_usuario_inexistente | Comportamiento cuando `Usuario.objects.get` lanza una excepción de inexistencia. | Devuelve `None`, se registra auditoría `LOGIN_FALLIDO` con usuario `None`. |
| test_login_contraseña_incorrecta | Flujo cuando la contraseña no coincide. | Devuelve `None` y se registra `LOGIN_FALLIDO` asociado al usuario. |

### 2.2 Casos Límite Cubiertos en este Bloque

Los test de autenticación cubren múltiples escenarios límite:

- **Usuario inexistente**: Excepción simulada con mock (`DoesNotExist`)
- **Contraseña incorrecta**: Transición de estado correcta hacia error
- **Autenticación exitosa**: Comportamiento completo: actualización de último acceso + auditoría
- **Validación del flujo interno** sin BD: llamadas correctas a `Usuario.objects.get`, `check_password`, `.save()`, y `registrar_evento_auditoria`.

## 3. Bloque 2: Pruebas de Registro de auditoria

**Archivo:** `apps/seguridad/tests/test_auditoria_unit.py`

**Clase:** `AuditoriaUnitTests`

**Módulo probado:** `apps.seguridad.servicios.autenticacion.registrar_evento_auditoria`

### 3.1 Resumen de Casos

| Test | Qué verifica | Resultado esperado |
|---|---|---|
| test_registrar_evento_con_usuario | Registro correcto de auditoría cuando se envía un usuario válido. | `RegistroAuditoria.objects.create` es llamado con los argumentos esperados. |
| test_registrar_evento_sin_usuario | Manejo del caso en que el usuario es `None` (ej. login fallido). | Se crea un registro con `UsuarioID=None`, tipo y detalle correctos. |

### 3.2 Casos Límite Cubiertos en este Bloque

Este bloque garantiza que el sistema registre adecuadamente las acciones críticas, incluso en escenarios límite como fallos de autenticación.

- **Usuario válido vs usuario inexistente (None)**
- **Integridad de parámetros enviados a auditoría**
- **Simulación de la capa de modelo sin BD**

## 4. Bloque 3: Pruebas de Logout

**Archivo:** `apps/seguridad/tests/test_logout_unit.py`

**Clase:** `LogoutUnitTests`

**Módulo probado:** `apps.seguridad.servicios.autenticacion.registrar_logout`

### 4.1 Resumen de Casos

| Test | Qué verifica | Resultado esperado |
|---|---|---|
| test_logout_registra_evento | Que un logout con usuario válido genera un registro de auditoría. | Se llama a `registrar_evento_auditoria` con tipo `LOGOUT`.. |
| test_logout_sin_usuario_no_registra_evento | Caso límite en el que `registrar_logout` recibe `None`. | No se registra ningún evento. |

### 4.2 Casos Límite Cubiertos en este Bloque

Los test de autenticación cubren múltiples escenarios límite:

- **Logout normal (usuario logueado)**
- **Logout inválido (usuario None)**
- Garantiza que no se generen auditorías incorrectas
- Revisa exclusivamente la lógica de negocio, sin dependencias externas

## 5. Conclusión

La suite de pruebas de este módulo válida que las funcionalidades esenciales del sistema autenticación, registro de auditoría y gestión del cierre de sesión cumplen rigurosamente con las reglas establecidas.

1. **Lógica de Autenticación (Capa de Servicio):**
   Nos aseguramos que la función `autenticar_usuario` maneja correctamente los principales casos límite del proceso de inicio de sesión. Las pruebas cubren credenciales válidas, usuario inexistente y contraseñas incorrectas, asegurando que la lógica responda adecuadamente en cada situación. Asimismo, validamos que la

actualización del último acceso y el registro de auditoría se ejecuten solo cuando corresponde.

2. **Registro de Auditoría (Capa de Servicio):**
   Las pruebas de `registrar_evento_auditoria` aseguran que el sistema mantenga la lógica de las acciones críticas. Verificamos que se generen registros tanto para eventos asociados a usuarios válidos como para fallos de autenticación, preservando el correcto flujo.
3. **Gestión de Logout (Capa de Servicio):**
   Las pruebas aplicadas a `registrar_logout` confirman que el sistema maneja correctamente la transición de cierre de sesión. Validamos que se registre un evento de salida únicamente cuando hay un usuario activo, evitando auditorías inválidas al recibir valores nulos.

Estos tests nos ayudan a asegurar que la lógica central del módulo Seguridad funciona de manera confiable y precisa en escenarios de éxito y fallo, cumpliendo con los criterios de requeridos para este componente del sistema.

**Módulo Pagos**

**1. Información General**

| **Ítem** | **Detalle** |
|---|---|
| Aplicación | apps.pagos (Módulo de pagos y membresías) |
| Framework de tests | Django TestCase (unittest) |
| Total de pruebas | 7 |
| Estado | Todas las pruebas pasan (OK) |
| Tiempo de corrida | ≈ 0.058s (en entorno local de desarrollo) |

Las pruebas unitarias del módulo **Pagos** se enfocan en tres bloques principales:

1. Creación y validación de **membresías de socios**.
2. Registro de **pagos** asociados a una membresía y actualización de su estado.
3. Generación de **alertas de morosidad** para membresías con pagos pendientes.

**2. Bloque 1: Servicio de Membresías**

**Archivo:** apps/pagos/tests/test_membresia_servicios.py
**Clase:** CrearMembresiaServicioTest
**Módulo probado:** apps.pagos.servicios.pagos_service.crear_membresia_para_socio

**2.1 Resumen de Casos**

| Test | Qué verifica | Resultado esperado |
|---|---|---|
| test_crear_membresia_calcula_fechas_y_estado_activa | Creación de una membresía para un socio a partir de un plan de membresía | Se crea un SocioMembresia con FechaInicio dada, FechaFin = FechaInicio + DuracionDias y Estado = ACTIVA |
| test_no_crea_segunda_membresia_activa_solapada | Restricción de crear una segunda membresía activa que se solape en fechas | Se lanza ValidationError indicando que el socio ya tiene una membresía activa en ese período y no se crea un nuevo registro |

**2.2 Casos Límite Cubiertos en este Bloque**

- Solo se permite crear una **membresía activa vigente** para un socio en un rango de fechas determinado (no se permiten solapamientos).
- Validación de existencia de los objetos referenciados:

  El socio (Socio) debe existir.

  El plan de membresía (PlanMembresia) debe existir.

- Cálculo correcto de las fechas:

  FechaFin se calcula como FechaInicio + DuracionDias.

  La membresía inicia en estado **ACTIVA** cuando se crea exitosamente.

**3. Bloque 2: Servicio de Pagos**

**Archivo:** apps/pagos/tests/test_pagos_servicios.py
**Clase:** RegistrarPagoServicioTest
**Módulo probado:** apps.pagos.servicios.pagos_service.registrar_pago_membresia

### 3.1 Resumen de Casos

| Test | Qué verifica | Resultado esperado |
|---|---|---|
| test_pago_parcial_actualiza_monto_pendiente_y_estado_morosa | Registro de un **pago parcial** para una membresía con saldo pendiente | Se crea un Pago con Monto correcto, MontoPendiente > 0 y la membresía queda en estado **MOROSA** |
| test_pago_completo_deja_sin_saldo_y_estado_activa | Registro de un pago que **cubre el valor total del plan de membresía** | MontoPendiente = 0.00 y la membresía pasa a estado **ACTIVA** |
| test_monto_negativo_lanza_validation_error | Validación de que no se aceptan montos de pago negativos o iguales a cero | Se lanza ValidationError y **no** se crea ningún registro de Pago en la base de datos |

### 3.2 Casos Límite Cubiertos en este Bloque

- **Montos válidos de pago:**

  Solo se aceptan montos **positivos** (monto > 0).

- **Control de saldo pendiente:**

  La suma de todos los pagos (Pago) asociados a una misma membresía **no puede superar** el precio (Precio) del PlanMembresia.

  MontoPendiente nunca es negativo.

- **Cambio de estado de la membresía según el pago:**

Si aún hay saldo pendiente (MontoPendiente > 0), la membresía se mantiene en estado **MOROSA**.

Cuando el saldo pendiente llega a 0, la membresía queda en estado **ACTIVA**.

- Uso de transacciones (transaction.atomic) para garantizar la consistencia entre:

  el registro del pago y la actualización del estado de la membresía.

## 4. Bloque 3: Alertas de Pagos / Morosidad

**Archivo:** apps/pagos/tests/test_alertas_servicios.py
**Clase:** AlertasPagoServicioTest
**Módulo probado:** apps.pagos.servicios.pagos_service.generar_alerta_morosidad

### 4.1 Resumen de Casos

| Test | Qué verifica | Resultado esperado |
|---|---|---|
| test_generar_alerta_para_membresia_morosa_sin_duplicar | Generación de alertas de pago pendiente para una membresía en estado **MOROSA** | Se crea una única AlertaPago por membresía morosa; llamadas repetidas reutilizan la misma alerta (idempotencia) y VistaEnPanel = False al inicio |
| test_no_generar_alerta_si_membresia_no_morosa | Bloqueo de generación de alertas cuando la membresía **no** está en estado moroso | Se lanza ValidationError y no se crea ninguna AlertaPago en la base de datos |

### 4.2 Casos Límite Cubiertos en este Bloque

- Validación estricta del **estado de la membresía**:

Solo se permiten alertas para membresías en estado **MOROSA**.

No se generan alertas para membresías **ACTIVAS** o **EXPIRADAS**.

- **Idempotencia en la creación de alertas:**

  Se utiliza get_or_create para evitar crear alertas duplicadas para la misma membresía morosa.

- Estado inicial de la alerta:

  Las alertas nuevas se crean con VistaEnPanel = False, permitiendo que la interfaz identifique alertas **pendientes de revisar**.

**5. Conclusión**

La suite de pruebas del **módulo Pagos** verifica de manera consistente que:

1. **Gestión de membresías**
   La creación de SocioMembresia respeta las reglas de negocio:
   - Solo se permite una membresía **activa y vigente** por socio en un mismo período.
   - Las fechas de inicio y fin se calculan de forma coherente con la duración del plan.
2. **Registro de pagos y estados de membresía**
   El servicio registrar_pago_membresia garantiza que:
   - Los montos de pago sean válidos (mayores a cero).
   - No se puedan registrar pagos que superen el valor del plan.
   - El estado de la membresía (ACTIVA/MOROSA) se actualice de forma automática según el saldo pendiente.
3. **Alertas de morosidad**
   El servicio generar_alerta_morosidad asegura que:
   - Solo se generen alertas para membresías realmente **morosas**.
   - No se creen alertas duplicadas para la misma situación, manteniendo una gestión limpia de notificaciones.
   - La interfaz pueda identificar alertas pendientes de revisión a través del campo VistaEnPanel.

En conjunto, las pruebas del módulo **Pagos** refuerzan la consistencia entre las reglas de negocio de membresías, el registro de pagos y la notificación de morosidad.