Prueba de Caja Blanca

"SKIPUR"

Integrantes:

Chavez Oscullo Klever Enrique

Guacan Rivera Alexander David

Trejo Duque Alex Fernando

Fecha 2025-07-08

Prueba caja blanca para el Requisito Funcional: Reservar una Cita

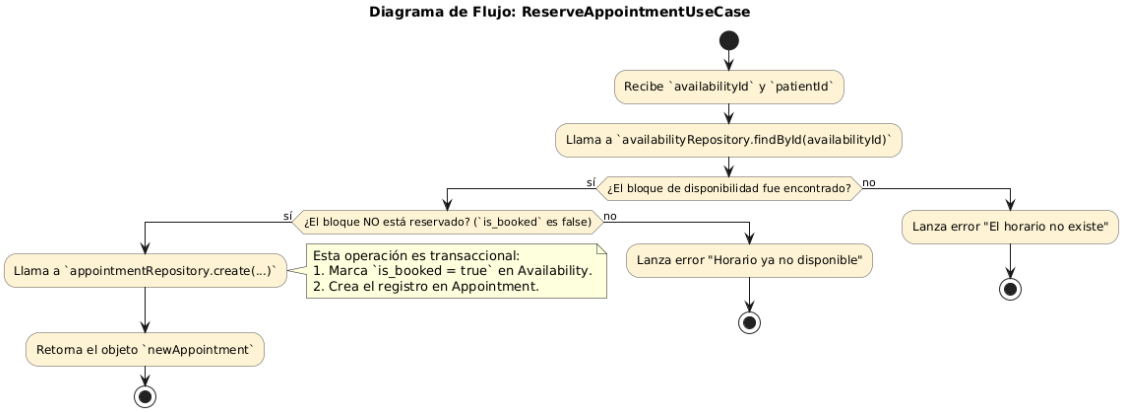
El propósito de esta prueba es analizar la estructura interna del código responsable de la funcionalidad de reserva de citas. Se busca garantizar que todas las rutas lógicas, condiciones y flujos de control dentro del módulo ReserveAppointmentUseCase sean evaluadas para identificar posibles errores y asegurar su correcto funcionamiento.

1. **CÓDIGO FUENTE**

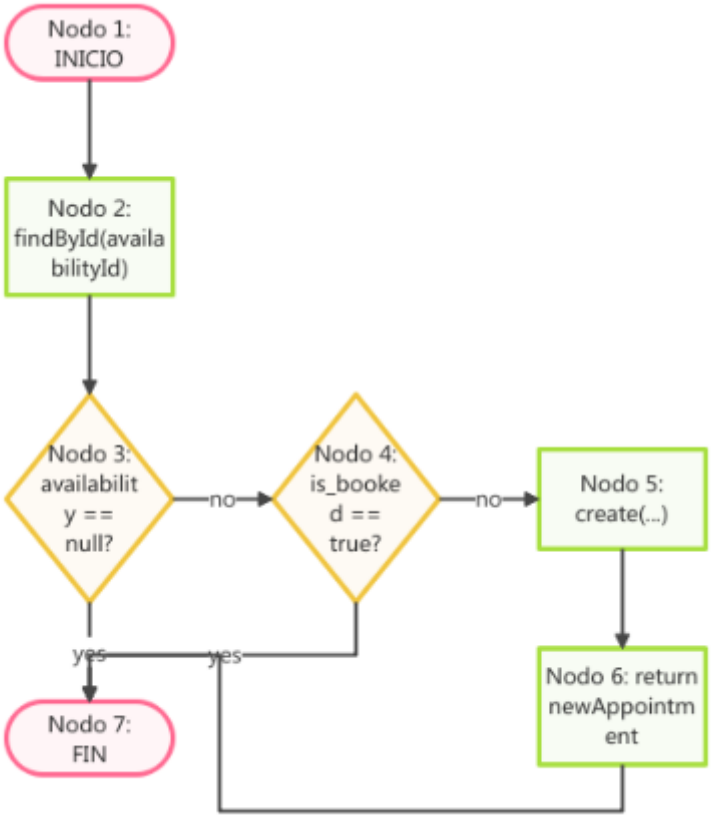
El fragmento de código seleccionado para el análisis corresponde al método execute de la clase ReserveAppointmentUseCase. Esta clase encapsula la lógica de negocio principal para crear una reserva, verificando la disponibilidad y garantizando que un horario no sea reservado más de una vez.

```
// src/application/use-cases/appointment/reserve-appointment.use-case.ts

import { IAvailabilityRepository } from "../../../domain/repositories/availability.repository";
import { IAppointmentRepository } from "../../../domain/repositories/appointment.repository";

interface ReserveAppointmentInput {
   availabilityId: string;
   patientId: string;
}

export class ReserveAppointmentUseCase {
  constructor(
    private readonly availabilityRepository: IAvailabilityRepository,
    private readonly appointmentRepository: IAppointmentRepository
  ) {}

  async execute(input: ReserveAppointmentInput) {
    const { availabilityId, patientId } = input;

    // 1. Verificar que el bloque de disponibilidad exista
    const availability = await this.availabilityRepository.findById(availabilityId);
    if (!availability) {
      throw new Error("El horario seleccionado no existe.");
    }

    // 2. Verificar que el bloque no esté ya reservado
    if (availability.is_booked) {
      throw new Error("Este horario ya no está disponible. Por favor, seleccione otro.");
    }

    // 3. Crear la cita. El repositorio se encargará de la transacción.
    const newAppointment = await this.appointmentRepository.create(
        {
          patient_id: patientId,
          specialist_id: availability.specialist_id
        },
        availabilityId
    );

    return newAppointment;
  }
}
```

2. DIAGRAMA DE FLUJO (DF)



3. GRAFO DE FLUJO (GF)



4. IDENTIFIACCIÒN DE LAS RUTAS (Camino básico)

En base al Grafo de Flujo, se identifican los caminos linealmente independientes que atraviesan el código desde el nodo de inicio hasta el nodo final.

- **R1 (Camino de Éxito):** 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7
  - *Descripción:* Se encuentra la disponibilidad, no está reservada, se crea la cita y se retorna exitosamente.
- **R2 (Error - Disponibilidad no encontrada):** 1 -> 2 -> 3 -> 7
  - *Descripción:* El availabilityId proporcionado no corresponde a ningún registro en la base de datos.
- **R3 (Error - Disponibilidad ya reservada):** 1 -> 2 -> 3 -> 4 -> 7
  - *Descripción:* Se encuentra la disponibilidad, pero su estado is_booked ya es true.

5. COMPLEJIDAD CICLOMÁTICA

La Complejidad Ciclomática V(G) mide el número de caminos independientes en el grafo y nos da una idea de la complejidad del código y el número mínimo de pruebas necesarias para cubrir todas las rutas.

Se calcula de las siguientes formas:

- **V(G) = P + 1** (Número de nodos predicado + 1)
  - Los nodos predicado (decisiones) en el grafo son el **Nodo 3** (availability == null?) y el **Nodo 4** (is_booked == true?).
  - P = 2
  - **V(G) = 2 + 1 = 3**
- **V(G) = A – N + 2** (Número de aristas - Número de nodos + 2)
  - Número de Nodos (N) = 7 (nodos del 1 al 7).
  - Número de Aristas (A) = 8 (las flechas en el grafo: 1->2, 2->3, 3->4, 3->7, 4->5, 4->7, 5->6, 6->7).
  - V(G) = 8 - 7 + 2 = 1 + 2
  - **V(G) = 3**

Prueba de Caja Blanca para el Requisito Funcional: Crear Disponibilidad Horaria

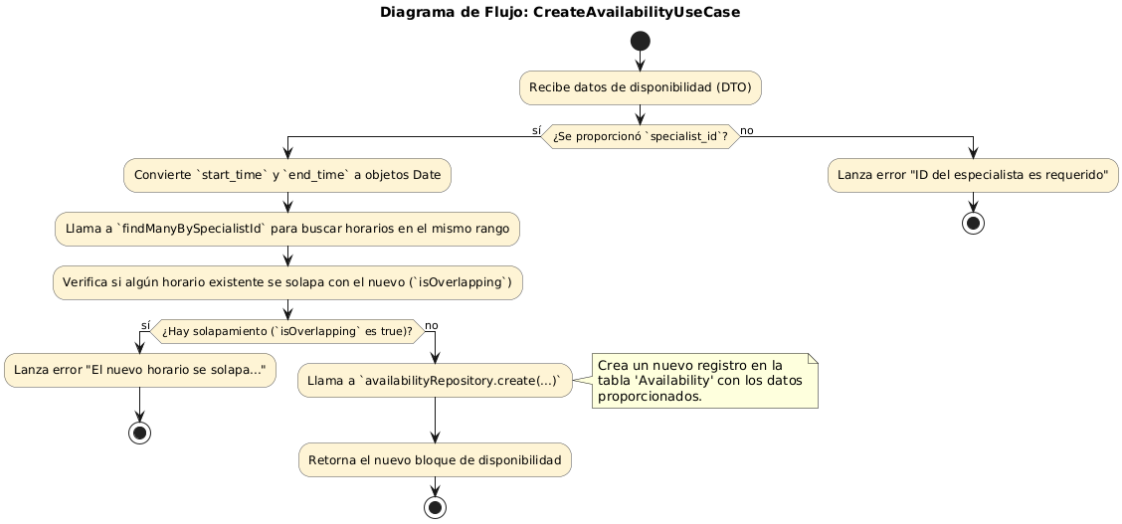
El propósito de esta prueba es analizar la estructura interna del código responsable de crear un nuevo bloque de disponibilidad para un especialista. El enfoque principal es validar el flujo de control que previene la creación de horarios que se solapan con bloques ya existentes, garantizando la integridad de la agenda del especialista.

1. CÓDIGO FUENTE

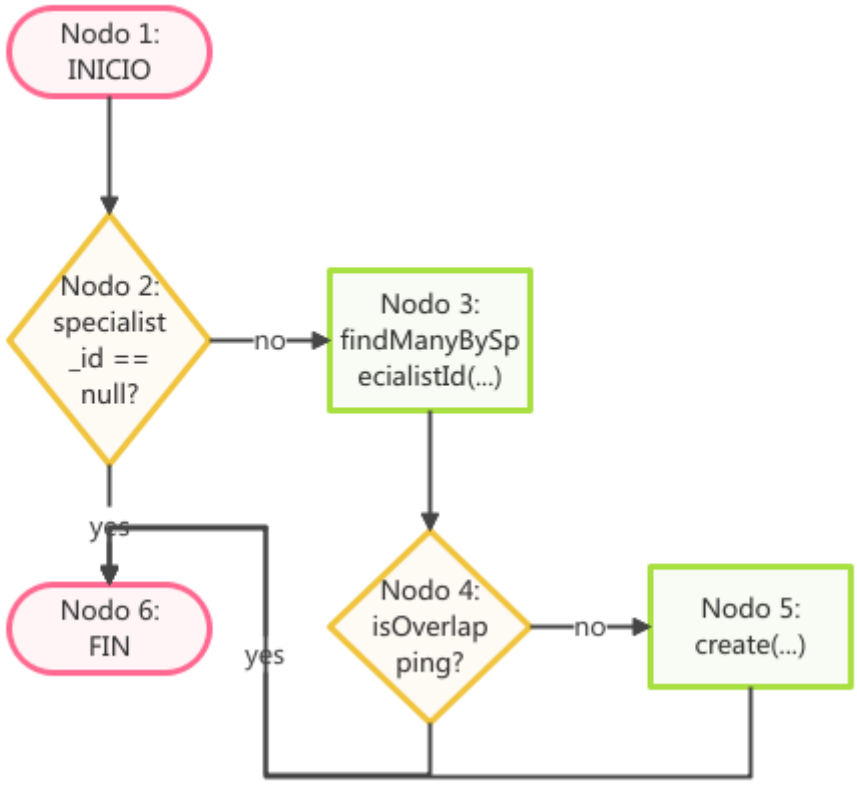
El fragmento de código seleccionado es el método execute de la clase CreateAvailabilityUseCase. Este método contiene la lógica para verificar si un nuevo bloque de tiempo entra en conflicto con otros antes de proceder con su creación.

```
// src/application/use-cases/availability/create-availability.use-case.ts

import { IAvailabilityRepository } from "../../../domain/repositories/availability.repository";
import { CreateAvailabilityDto } from "../../dtos/availability.dtos";

export class CreateAvailabilityUseCase {
  constructor(private readonly availabilityRepository: IAvailabilityRepository) {}

  async execute(dto: CreateAvailabilityDto) {
    const { specialist_id, start_time, end_time } = dto;
    if (!specialist_id) {
       throw new Error("El ID del especialista es requerido.");
    }

    // 1. Convertir fechas a objetos Date
    const startDate = new Date(start_time);
    const endDate = new Date(end_time);

    // 2. Verificar si ya existe un bloque que se solape con el nuevo
    const existingAvailabilities = await
this.availabilityRepository.findManyBySpecialistId(specialist_id, startDate, endDate);

    const isOverlapping = existingAvailabilities.some(existing =>
      (startDate < new Date(existing.end_time)) && (endDate > new Date(existing.start_time))
    );

    if (isOverlapping) {
      throw new Error("El nuevo horario se solapa con un bloque de disponibilidad existente.");
    }

    // 3. Crear el nuevo bloque de disponibilidad
    return this.availabilityRepository.create({
       specialist_id,
       start_time,
       end_time,
    });
  }
}
```

2. DIAGRAMA DE FLUJO (DF)



3. GRAFO DE FLUJO (GF)



4. IDENTIFIACCIÒN DE LAS RUTAS (Camino basico)

Se identifican los caminos linealmente independientes a través del Grafo de Flujo.

- **R1 (Camino de Éxito):** 1 -> 2 -> 3 -> 4 -> 5 -> 6
  - *Descripción:* Se provee un specialist_id, no se encuentra ningún horario que se solape, y el nuevo bloque de disponibilidad se crea exitosamente.
- **R2 (Error - Horario se solapa):** 1 -> 2 -> 3 -> 4 -> 6
  - *Descripción:* Se provee un specialist_id, pero la búsqueda en la base de datos encuentra al menos un bloque de horario existente que entra en conflicto con las nuevas fechas.
- **R3 (Error - Falta ID de especialista):** 1 -> 2 -> 6
  - *Descripción:* No se proporciona el specialist_id en los datos de entrada, por lo que la operación se detiene antes de consultar la base de datos.

5. COMPLEJIDAD CICLOMÁTICA

Se calcula la Complejidad Ciclomática V(G) para determinar la complejidad del código.

- **V(G) = P + 1** (Número de nodos predicado + 1)
    - Nodos predicado (decisiones): **Nodo 2** (specialist_id == null?) y **Nodo 4** (isOverlapping?).
    - P = 2
    - **V(G) = 2 + 1 = 3**
- **V(G) = A – N + 2** (Número de aristas - Número de nodos + 2)
    - Número de Nodos (N) = 6 (nodos del 1 al 6).
    - Número de Aristas (A) = 7 (las flechas en el grafo).
    - V(G) = 7 - 6 + 2 = 1 + 2
    - **V(G) = 3**

Prueba de Caja Blanca para el Requisito Funcional: Desactivar una Especialidad

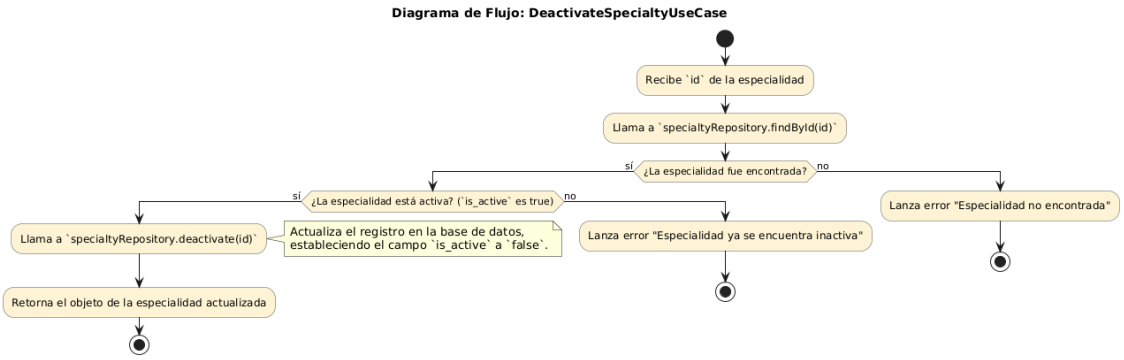
El objetivo de esta prueba es analizar la lógica interna del caso de uso encargado de desactivar una especialidad. El análisis se centra en las validaciones previas que aseguran que la especialidad exista y que no se intente desactivar una que ya está inactiva, garantizando así la consistencia del estado del sistema.

1. CÓDIGO FUENTE

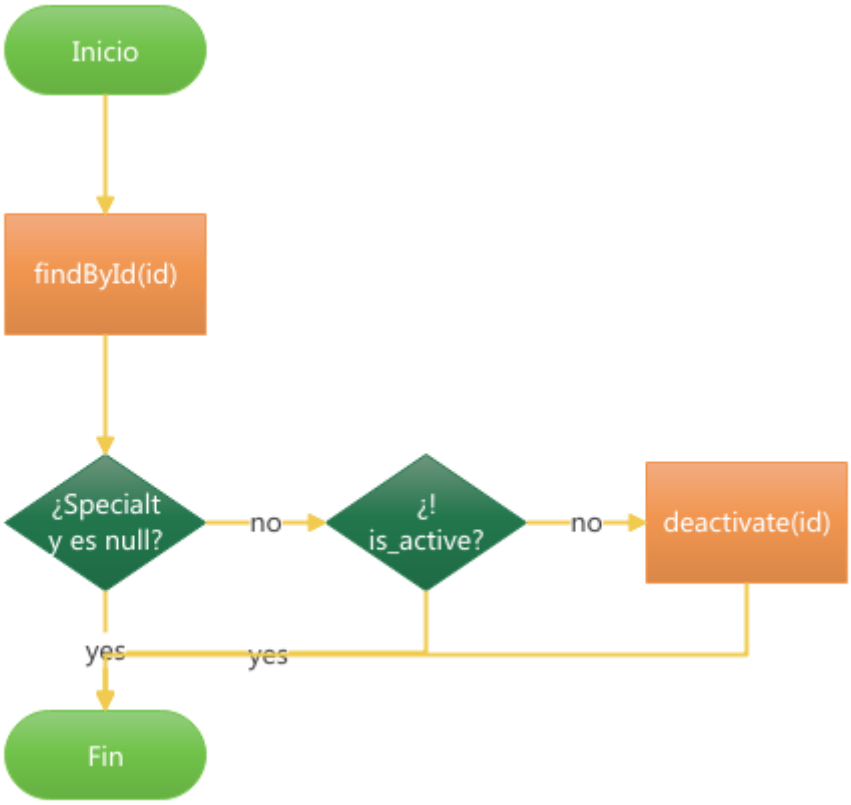
El fragmento de código a analizar es el método execute de la clase DeactivateSpecialtyUseCase.

```
// src/application/use-cases/specialty/deactivate-specialty.use-case.ts

import { ISpecialtyRepository } from "../../../domain/repositories/specialty.repository";

export class DeactivateSpecialtyUseCase {
  constructor(private readonly specialtyRepository: ISpecialtyRepository) {}

  async execute(id: string) {
    // 1. Verificar que la especialidad exista
    const specialty = await this.specialtyRepository.findById(id);
    if (!specialty) {
      throw new Error("Especialidad no encontrada.");
    }

    // 2. Verificar que no esté ya inactiva
    if (!specialty.is_active) {
      throw new Error("Esta especialidad ya se encuentra inactiva.");
    }

    // 3. Llamar al repositorio para realizar la desactivación (actualización)
    return this.specialtyRepository.deactivate(id);
  }
}
```

2. DIAGRAMA DE FLUJO (DF)



3. GRAFO DE FLUJO (GF)



4. IDENTIFIACCIÒN DE LAS RUTAS (Camino basico)

Los caminos linealmente independientes a través del Grafo de Flujo son:

- **R1 (Camino de Éxito):** 1 -> 2 -> 3 -> 4 -> 5 -> 6
  - *Descripción:* Se encuentra la especialidad, está activa, se procede a desactivarla y la operación finaliza con éxito.
- **R2 (Error - Especialidad no encontrada):** 1 -> 2 -> 3 -> 6
  - *Descripción:* El id proporcionado no corresponde a ninguna especialidad en la base de datos.
- **R3 (Error - Especialidad ya inactiva):** 1 -> 2 -> 3 -> 4 -> 6
  - *Descripción:* Se encuentra la especialidad, pero su estado is_active ya es false.

5. COMPLEJIDAD CICLOMÁTICA

Se calcula la Complejidad Ciclomática V(G) del fragmento de código.

- **V(G) = P + 1** (Número de nodos predicado + 1)
  - Nodos predicado (decisiones): **Nodo 3** (specialty == null?) y **Nodo 4** (!is_active?).
  - P = 2
  - **V(G) = 2 + 1 = 3**
- **V(G) = A – N + 2** (Número de aristas - Número de nodos + 2)
  - Número de Nodos (N) = 6.
  - Número de Aristas (A) = 7.
  - V(G) = 7 - 6 + 2 = 1 + 2
  - **V(G) = 3**

Prueba de Caja Blanca para el Requisito Funcional: Iniciar Sesión de Usuario

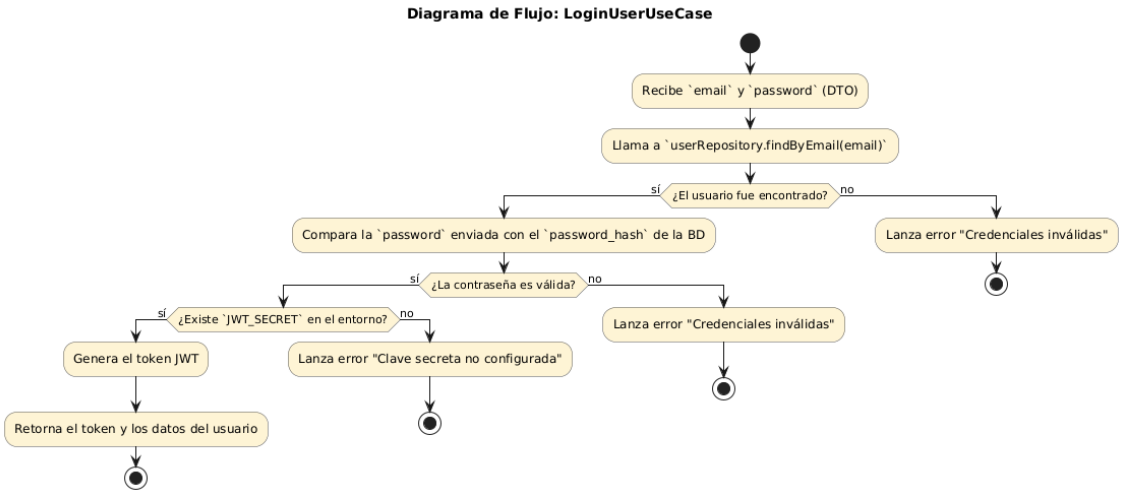
El objetivo de esta prueba es analizar el flujo lógico del caso de uso responsable de la autenticación de usuarios. Se examinarán las validaciones de credenciales, incluyendo la verificación de la existencia del usuario por su correo electrónico y la comparación de la contraseña proporcionada con su hash almacenado, para garantizar un proceso de inicio de sesión seguro y correcto.

1. CÓDIGO FUENTE

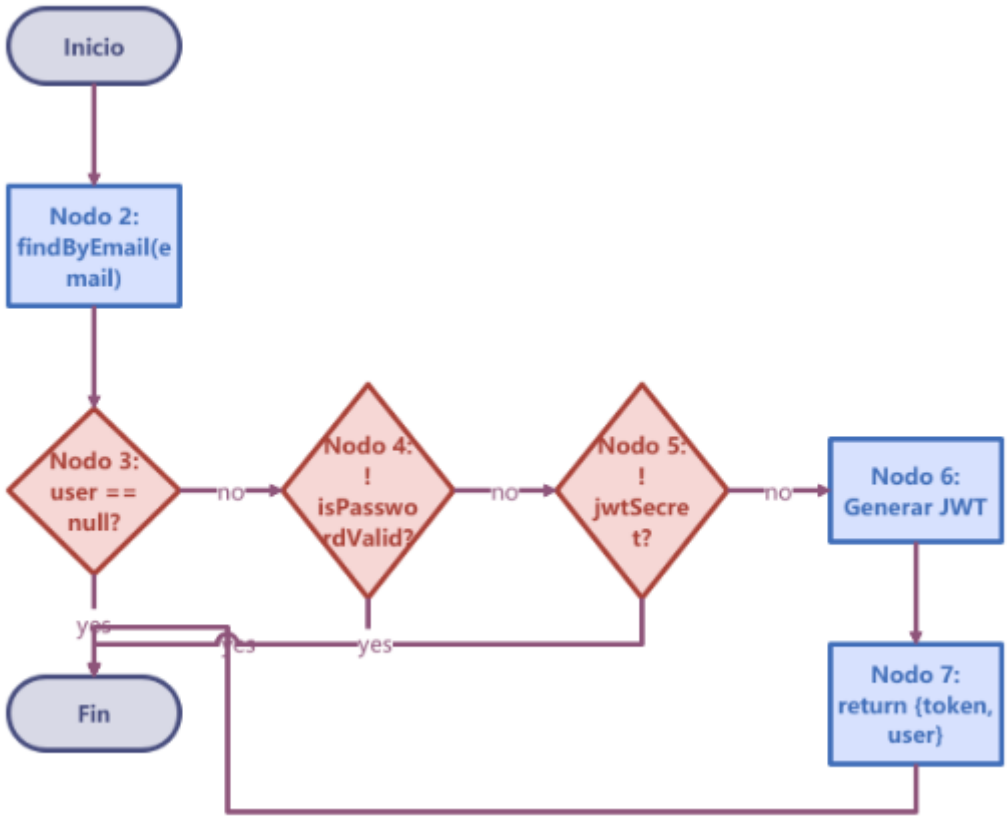
El fragmento de código a analizar es el método execute de la clase LoginUserUseCase.

```
// src/application/use-cases/auth/login-user.use-case.ts

import { IUserRepository } from "../../../domain/repositories/user.repository";
import { LoginUserDto } from "../../dtos/auth.dtos";
import * as bcrypt from 'bcryptjs';
import * as jwt from 'jsonwebtoken';

export class LoginUserUseCase {
  constructor(private readonly userRepository: IUserRepository) {}

  async execute(dto: LoginUserDto): Promise<{ token: string; user: { id: string, email: string, full_name: string, role: string } }> {
    // 1. Buscar al usuario por su email
    const user = await this.userRepository.findByEmail(dto.email);
    if (!user) {
      throw new Error("Credenciales inválidas."); // Mensaje genérico por seguridad
    }

    // 2. Comparar la contraseña enviada con el hash guardado en la BD
    const isPasswordValid = await bcrypt.compare(dto.password, user.password_hash);
    if (!isPasswordValid) {
      throw new Error("Credenciales inválidas."); // Mismo mensaje genérico
    }

    // 3. Generar el JSON Web Token (JWT)
    const jwtSecret = process.env.JWT_SECRET;
    if (!jwtSecret) {
      throw new Error("La clave secreta para JWT no está configurada.");
    }

    const payload = { id: user.id, email: user.email, role: user.role };
    const token = jwt.sign(payload, jwtSecret, { expiresIn: '7d' });

    // 4. Devolver el token y la información básica del usuario
    return {
      token,
      user: { id: user.id, email: user.email, full_name: user.full_name, role: user.role }
    };
  }
}
```

2. DIAGRAMA DE FLUJO (DF)



3. GRAFO DE FLUJO (GF)



4. IDENTIFIACCIÒN DE LAS RUTAS (Camino basico)

Los caminos linealmente independientes a través del Grafo de Flujo son:

- **R1 (Camino de Éxito):** 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8
  - *Descripción:* El usuario existe, la contraseña es correcta, la clave JWT está configurada y se genera y devuelve el token exitosamente.
- **R2 (Error - Usuario no encontrado):** 1 -> 2 -> 3 -> 8
  - *Descripción:* El correo electrónico proporcionado no corresponde a ningún usuario en la base de datos.
- **R3 (Error - Contraseña incorrecta):** 1 -> 2 -> 3 -> 4 -> 8
  - *Descripción:* El usuario existe, pero la contraseña proporcionada no coincide con el hash almacenado.
- **R4 (Error - Clave JWT no configurada):** 1 -> 2 -> 3 -> 4 -> 5 -> 8
  - *Descripción:* Las credenciales son válidas, pero el servidor no tiene configurada la variable de entorno JWT_SECRET, lo que impide la creación del token.

5. COMPLEJIDAD CICLOMÁTICA

Se calcula la Complejidad Ciclomática V(G) del fragmento de código.

- **V(G) = P + 1** (Número de nodos predicado + 1)
  - Nodos predicado (decisiones): **Nodo 3** (user == null?), **Nodo 4** (!isPasswordValid?) y **Nodo 5** (!jwtSecret?).
  - P = 3
  - **V(G) = 3 + 1 = 4**
- **V(G) = A – N + 2** (Número de aristas - Número de nodos + 2)
  - Número de Nodos (N) = 8.
  - Número de Aristas (A) = 10.
  - V(G) = 10 - 8 + 2 = 2 + 2
  - **V(G) = 4**

Prueba caja blanca de describa el requisito funcional

1. **CÓDIGO FUENTE**

Pegar el trozo de código fuente que se requiere para el caso de prueba

2. **DIAGRAMA DE FLUJO (DF)**

Realizar un DF del código fuente del numeral 1

3. **GRAFO DE FLUJO (GF)**

Realizar un GF en base al DF del numeral 2

4. **IDENTIFIACCIÒN DE LAS RUTAS (Camino basico)**

Determinar en base al GF del numeral 4
RUTAS

R1: 1
R2:

5. **COMPLEJIDAD CICLOMÁTICA**

Se puede calcular de las siguientes formas:

- V(G) = número de nodos predicados(decisiones)+1
  V(G)=

- V(G) = A – N + 2
  V(G)=

DONDE:
P: Número de nodos predicado
A: Número de aristas
N: Número de nodos