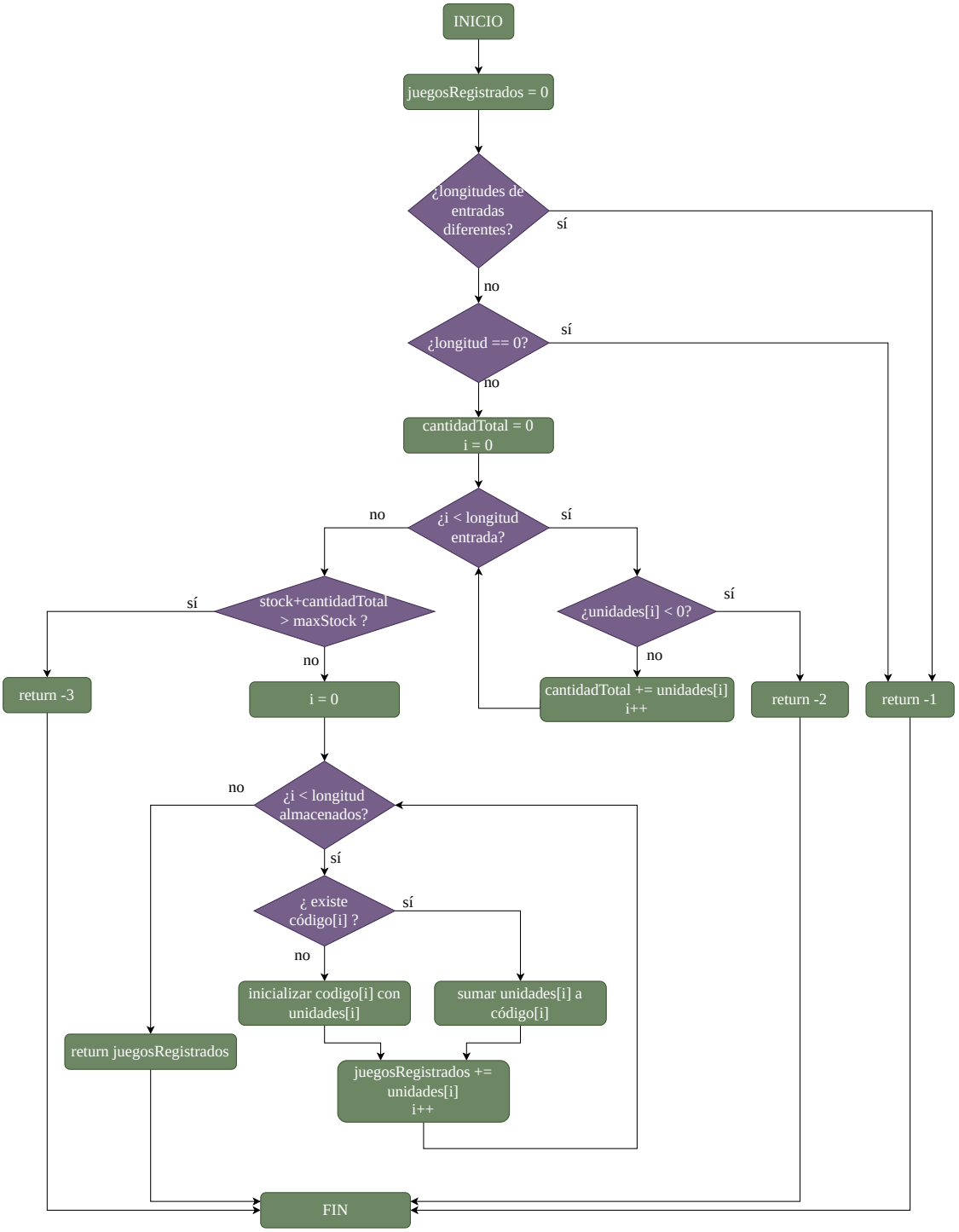
ED05 - Pruebas de Caja Blanca y Caja Negra. Propuesta de solución

A continuación se presenta la propuesta de solución para la actividad ED05 - Pruebas de Caja Blanca y Caja Negra. Se indican los pasos seguidos para la realización de las pruebas de caja blanca y caja negra, así como los casos de prueba diseñados para cada uno de los métodos.

Pruebas de Caja Blanca de `registrarLoteJuegos()`

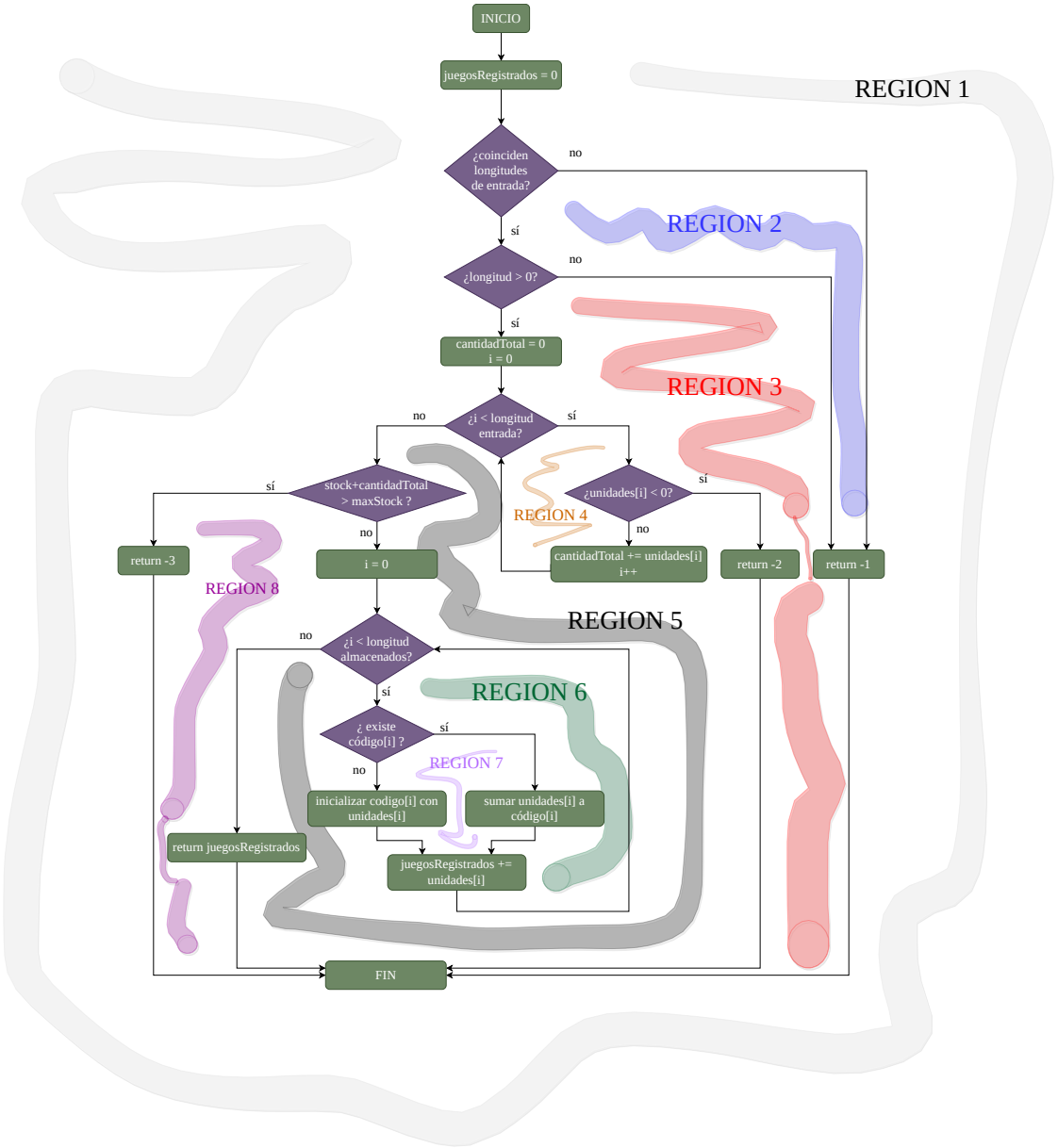
```java
// Metodo que registra un lote de juegos en el mapa de juegos de la tienda
public int registrarLoteJuegos(String[] codigos, int[] unidadesPorCodigo) {
    int juegosRegistrados = 0;

    //comprobamos que las longitudes de los arrays sean iguales
    if (codigos.length != unidadesPorCodigo.length) {
        return -1;
    }

    //Comprobamos que los valores de cantidad sean positivos y sino, la sumamos a cantidadTotal
    int cantidadTotal = 0;
    for (int cantidad : unidadesPorCodigo) {
        if (cantidad < 0) {
            return -2;
        } else {
            cantidadTotal += cantidad;
        }
    }

    //comprobamos que no se exceda el stock máximo
    if (obtenerStockActual() + cantidadTotal > maxStock) {
        return -3;
    }

    for (int i = 0; i < codigos.length; i++) {
        if (stockJuegos.containsKey(codigos[i])) {
            stockJuegos.put(codigos[i], stockJuegos.get(codigos[i]) + unidadesPorCodigo[i]);
        } else {
            stockJuegos.put(codigos[i], unidadesPorCodigo[i]);
        }
        juegosRegistrados += unidadesPorCodigo[i];
    }
    return juegosRegistrados;
}
```

1. Creación del grafo de flujo del método `registrarLoteJuegos`:



2. Cálculo de la complejidad ciclomática:

- $V(G) = a - n + 2 = 26 - 20 + 2 = 8$ siendo a el número de arcos y n el número de nodos.
- $V(G) = r = 8$ siendo r el número de regiones cerradas del grafo (incluyendo la externa).
- $V(G) = c + 1 = 7 + 1 = 8$ siendo c el número de nodos de condición.



Por lo tanto, podremos conseguir una cubrición de sentencias y condiciones del 100% con como mucho 8 casos de prueba.

3. Identificación de los caminos independientes:

   Se identifican 7 caminos, de los cuales 2 son irrealizables (los caminos que salen de la condición del for por el no en la primera iteración, lo cual es imposible por las condiciones previas).

4. Diseño de los casos de prueba para cada camino:

   - **Caso de Prueba CP1**: `codigos.length != unidadesPorCodigo.length`

        - Entrada: `codigos = {"COD1", "COD2"}, unidadesPorCodigo = {1}`
        - Salida esperada: -1
    - **Caso de Prueba CP2**: `codigos.length = 0`
        - Entrada: `codigos = {}, unidadesPorCodigo = {}`
        - Salida esperada: -1
    - **Caso de Prueba CP3**: `unidadesPorCodigo[i] < 0`
        - Entrada: `codigos = {"COD1", "COD2"}, unidadesPorCodigo = {1, -1}`
        - Salida esperada: -2
    - **Caso de Prueba CP4**: `obtenerStockActual() + cantidadTotal > maxStock`
        - Entrada: `codigos = {"COD1", "COD2"}, unidadesPorCodigo = {1, 1}`
        - Salida esperada: -3
    - **Caso de Prueba CP5**: `codigo` incluye valores ya registrados y valores nuevos
        - Entrada: `codigos = {"COD1", "COD2"}, unidadesPorCodigo = {1, 1}`
        - Salida esperada: 2
    - **Caso de Prueba CP6**: `codigos.length = 0` --> No viable, equivale a caso 2
        - Salida esperada: -1
    - **Caso de Prueba CP7**: `codigos.length = 0` --> No viable, equivale a caso 2
        - Salida esperada: -1

Pruebas de Caja Negra de `venderJuego()`

Para las pruebas de caja negra del método `venderJuego` nos centramos en la firma del método y en sus especificaciones, que son:

- El método recibe como parámetro la cantidad de juegos a vender
- La cantidad no podrá ser menor o igual a 0
- La cantidad no podrá ser mayor al stock actual
- El código del juego deberá tener 6 caracteres, los 3 primeros letras mayúsculas y los 3 últimos dígitos

Para definir las pruebas de caja negra, se pueden seguir los siguientes pasos:

1. **Particiones de equivalencia**:
    i. Para la cantidad de juegos a vender:
        - **Partición 1**: cantidad > 0 y cantidad &lt = stock actual
        - **Partición 2**: cantidad &lt = 0
        - **Partición 3**: cantidad > stock actual

    ii. Para el código del juego:
        - **Partición 1**: código válido
        - **Partición 2**: código con longitud distinta de 6
        - **Partición 3**: código con letras minúsculas
        - **Partición 4**: código con dígitos en las primeras 3 posiciones
        - **Partición 5**: código con letras en las últimas 3 posiciones

Condición de entrada	**Clase de equivalencia**	**Clases válidas**	**COD**	**Clases no válidas**	**COD**
cantidad	Rango	1 - stockDisponible	V1	&lt =	NV1
cantidad				> stockDisponible	NV2
código	Formato de string	"AAA000" - "ZZZ999"	V2	Nulo	NV3
código				longitud != 6	NV4
código				letras minúsculas	NV5
código				digitos en las 3 primeras posiciones	NV6
código				no dígitos en las 3 últimas posiciones	NV7

1. **Valores límite**:

    i. Para la cantidad de juegos a vender:
        - **Caso 1**: cantidad = 1
        - **Caso 2**: cantidad = stock disponible
        - **Caso 3**: cantidad = 0
        - **Caso 4**: cantidad = stock disponible + 1

    ◦ Para el código del juego:
        - **Caso 1**: código válido
        - **Caso 2**: código con longitud 5
        - **Caso 3**: código con longitud 7

- **Caso 4**: código con letras minúsculas
- **Caso 5**: código con dígitos en las primeras 3 posiciones
- **Caso 6**: código con no-dígitos en las últimas 3 posiciones
- **Caso 7**: código nulo

2. **Conjetura de errores**: aquí indicamos los errores retornados estimados. En este caso, lo hacemos para que concuerde con el código del método proporcionado, pero si no disponemos del código, debería de hacerse en base a la especificación del método.

    i. Si la cantidad de juegos a vender es menor o igual a 0, devuelve una excepción `IllegalArgumentException`
    ii. Si la cantidad de juegos a vender es mayor al stock actual, se devolverá -2
    iii. Si el código del juego es nulo, se devolverá una excepción `NullPointerException`
    iv. Si el código del juego no cumple con el formato especificado (no tiene 6 caracteres, o no tiene 3 letras mayúsculas en los primeros 3 caracteres, o no tiene 3 dígitos en los últimos 3 caracteres) se devolverá 0
    v. Si los datos son correctos, se devolverá la cantidad de juegos vendidos

3. Diseño de casos de prueba Combinamos los casos planteados por valores límite y particiones de equivalencia junto a la conjetura de errores planteada para diseñar los casos de prueba:

    - **Caso de Prueba CP1**: cantidad = 1, código válido
        - Entrada: cantidad = 1, código = "COD123"
        - Condición interna: cantidad &lt stock
        - Salida esperada: 1
    - **Caso de Prueba CP2**: cantidad = 0, código válido
        - Entrada: cantidad = 0, código = "COD123"
        - Salida esperada: 0
    - **Caso de Prueba CP3**: cantidad igual al stock para el código dado, código válido
        - Entrada: cantidad = 1, código = "COD12"
        - Condición interna: cantidad = stock
        - Salida esperada: 10
    - **Caso de Prueba CP4**: cantidad superior al stock para el código dado, código válido
        - Entrada: cantidad = 11, código = "COD123"

        - Condición interna: cantidad > stock
        - Salida esperada: -2
    - **Caso de Prueba CP5**: cantidad válida, código a null
        - Entrada: cantidad = 1, código = null
        - Condición interna: N/A
        - Salida esperada: 0
    - **Caso de Prueba CP6**: cantidad válida, código con longitud 5
        - Entrada: cantidad = 1, código = "COD12"
        - Condición interna: cantidad &lt = stock
        - Salida esperada: 0
    - **Caso de Prueba CP7**: cantidad válida, código con longitud 7
        - Entrada: cantidad = 1, código = "COD1234"
        - Condición interna: cantidad &lt = stock
        - Salida esperada: 0
    - **Caso de Prueba CP8**: cantidad válida, código con letras minúsculas
        - Entrada: cantidad = 1, código = "cod123"
        - Condición interna: cantidad &lt = stock
        - Salida esperada: 0
    - **Caso de Prueba CP9**: cantidad válida, código con no-letras en las primeras 3 posiciones
        - Entrada: cantidad = 1, código = "1AB123"
        - Condición interna: cantidad &lt = stock
        - Salida esperada: 0
    - **Caso de Prueba CP10**: cantidad válida, código con no-números en las últimas 3 posiciones
        - Entrada: cantidad = 1, código = "COD12A"
        - Condición interna: cantidad &lt = stock
        - Salida esperada: 0

Todos los casos de prueba diseñados en los dos apartados anteriores se implementan como test unitarios en JUnit para comprobar que los métodos `registrarLoteJuegos` y `venderJuego` cumple con las especificaciones (en el enunciado solo se pedía implementar los test de `registrarLoteJuegos`, pero se adjuntan los test de `venderJuego` para completar la solución). A continuación se muestra el código de los test unitarios:

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;

class GestorJuegosTest {
    GestorJuegos gestor;

    @BeforeEach
    void setUp() {
        // Inicialización de los objetos necesarios para los tests
        gestor = new GestorJuegos();
    }

    // Tests de caja blanca registrarLoteJuegos
    @Test
    void testRegistrarLoteJuegosCP1() {
        // Registrar juegos con longitudes de arrays diferentes
        String[] codigos = {"COD1", "COD2"};
        int[] unidadesPorCodigo = {1};
        assertEquals(-1, gestor.registrarLoteJuegos(codigos,
unidadesPorCodigo));
    }

    @Test
    void testRegistrarLoteJuegosCP2() {
        // Registrar juegos con arrays vacíos
        String[] codigos = {};
        int[] unidadesPorCodigo = {};
        assertEquals(-1, gestor.registrarLoteJuegos(codigos,
unidadesPorCodigo));
    }

    @Test
    void testRegistrarLoteJuegosCP3() {
        // Registrar juegos con cantidades negativas
        String[] codigos = {"COD1", "COD2"};
        int[] unidadesPorCodigo = {1, -1};
        assertEquals(-2, gestor.registrarLoteJuegos(codigos,
unidadesPorCodigo));
    }

    @Test
    void testRegistrarLoteJuegosCP4() {
        // Registro inicial para ocupar el stock máximo
        String[] codigos = {"COD1"};
```

```
        int[] unidadesPorCodigo = {GestorJuegos.MAX_STOCK};
        gestor.registrarLoteJuegos(codigos, unidadesPorCodigo);

        // Registrar juegos - Exceso de stock
        String[] codigosNuevos = {"COD1", "COD2"};
        int[] unidadesPorCodigoNuevas = {1, 1};
        assertEquals(-3, gestor.registrarLoteJuegos(codigosNuevos,
unidadesPorCodigoNuevas));
    }

    @Test
    void testRegistrarLoteJuegosCP5() {
        String[] codigos = {"COD1"};
        int[] unidadesPorCodigo = {5};
        gestor.registrarLoteJuegos(codigos, unidadesPorCodigo);

        // Registrar juegos con valores ya registrados y nuevos
        String[] codigosNuevos = {"COD1", "COD2"};
        int[] unidadesPorCodigoNuevas = {1, 1};
        assertEquals(2, gestor.registrarLoteJuegos(codigosNuevos,
unidadesPorCodigoNuevas));
    }

    // Tests de caja negra venderJuego
    @Test
    void testVenderJuegoCP1() {
        // Venta de un juego con stock suficiente
        gestor.registrarLoteJuegos(new String[]{"COD123"}, new int[]
{5});
        assertEquals(1, gestor.venderJuego("COD123", 1));
    }

    @Test
    void testVenderJuegoCP2() {
        // Venta de un juego con cantidad 0
        gestor.registrarLoteJuegos(new String[]{"COD123"}, new int[]
{5});
        assertEquals(0, gestor.venderJuego("COD123", 0));
    }

    @Test
    void testVenderJuegoCP3() {
        // Venta de un juego con cantidad igual al stock
        gestor.registrarLoteJuegos(new String[]{"COD123"}, new int[]
{10});
        assertEquals(10, gestor.venderJuego("COD123", 10));
    }
```

```
    @Test
    void testVenderJuegoCP4() {
        // Venta de un juego con cantidad superior al stock
        gestor.registrarLoteJuegos(new String[]{"COD123"}, new int[]
{5});
        assertEquals(-2, gestor.venderJuego("COD123", 10));
    }

    @Test
    void testVenderJuegoCP5() {
        // Venta de un juego con cantidad superior al stock
        gestor.registrarLoteJuegos(new String[]{"COD123"}, new int[]
{5});

        //Comprobar que gestor.venderJuego(null, 1) devuelve
nullpointerexception
        assertThrows(NullPointerException.class, () ->
gestor.venderJuego(null, 1));
    }

    @Test
    void testVenderJuegoCP6() {
        // Venta de un juego con código de longitud 5
        gestor.registrarLoteJuegos(new String[]{"COD12"}, new int[]
{5});
        assertEquals(0, gestor.venderJuego("COD12", 1));
    }

    @Test
    void testVenderJuegoCP7() {
        // Venta de un juego con código de longitud 7
        gestor.registrarLoteJuegos(new String[]{"COD1234"}, new int[]
{5});
        assertEquals(0, gestor.venderJuego("COD1234", 1));
    }

    @Test
    void testVenderJuegoCP8() {
        // Venta de un juego con código en minúsculas
        gestor.registrarLoteJuegos(new String[]{"cod123"}, new int[]
{5});
        assertEquals(0, gestor.venderJuego("cod123", 1));
    }

    @Test
    void testVenderJuegoCP9() {
        // Venta de un juego con dígitos en las primeras 3 posiciones
        gestor.registrarLoteJuegos(new String[]{"1AB123"}, new int[]
```

```
{5});
        assertEquals(0, gestor.venderJuego("1AB123", 1));
    }

    @Test
    void testVenderJuegoCP10() {
        // Venta de un juego con letras en las últimas 3 posiciones
        gestor.registrarLoteJuegos(new String[]{"COD12A"}, new int[]
{5});
        assertEquals(0, gestor.venderJuego("COD12A", 1));
    }
}
```