Escuela Colombiana de Ingeniería Julio Garavito

Imagen en blanco y negro

Descripción generada automáticamente con confianza baja

**Análisis dinámico**

Presentado por:

Camilo Andrés Fernández Diaz

Andrés Jacobo Sepúlveda Sánchez

Asignatura:

Programación orientada a objetos

Grupo:

ISIS POOB-4

Docente:

Juan Sebastián Frasica Galeano

María Irma Diaz Rozo

El análisis dinámico de cobertura de código se realizó en IntelliJ IDEA utilizando las clases de prueba JUnit creadas para el proyecto. La herramienta de cobertura permite visualizar las instrucciones cubiertas y las instrucciones faltantes en cada clase. Esto proporciona una visión clara de la eficiencia del código en términos de cobertura y muestra las áreas que necesitan mejoras o validaciones adicionales.

**Resumen General de Cobertura:**

El proyecto alcanzó un nivel de cobertura de código del 87,6%, lo cual indica una implementación sólida con errores mínimos en términos de pruebas y cobertura. A continuación, se detalla el porcentaje de cobertura de cada clase en el proyecto.

**Resumen de Cobertura de Código:**

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

Ahora se mostrará en cuanto porcentaje cada clase dentro de la cobertura mostrada:

Tabla

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación, Tabla

Descripción generada automáticamente con confianza media

**Cobertura de Código por Clase:**

Una vez mostrado cada porcentaje simplemente nos enfocaremos en este documento en las clases de pruebas que se realizaron a lo largo de este proyecto, y se dejaran de ultimas las pruebas de aceptación hechas por cada uno:

* BrainManagerTest: La clase BrainManagerTest tiene como objetivo realizar pruebas unitarias para validar el comportamiento de la clase BrainManager. BrainManager parece ser una clase que utiliza el patrón Singleton, maneja un contador de cerebros (puntos) y actualiza un JLabel periódicamente con los puntos acumulados.

**Configuración Inicial:**

**@BeforeEach setUp()**: Antes de ejecutar cada prueba, se crea una nueva instancia de BrainManager usando el constructor de la clase.

**Descripción de las Pruebas**

1. **testSingletonInstance():**
   1. **Objetivo**: Verificar que BrainManager implementa correctamente el patrón Singleton.
   2. **Prueba**: Se obtienen dos instancias de BrainManager mediante el método getInstance() y se verifica que ambas referencias apuntan a la misma instancia.
   3. **Resultado esperado**: Ambas instancias deben ser la misma (singleton).
2. **testSetBrainCounterLabel()**
   1. **Objetivo**: Verificar que al configurar el JLabel mediante setBrainCounterLabel(), el texto del JLabel se inicializa correctamente.
   2. **Prueba**: Se asigna un JLabel vacío al método y se verifica que no sea nulo y que su valor inicial sea "50".
   3. **Resultado esperado**: El JLabel debe mostrar "50" al inicio.
3. **testAddBrainsUpdatesLabel()**
   1. **Objetivo**: Verificar que el puntaje aumenta automáticamente después de 10 segundos mediante un timer.
   2. **Prueba**: Se espera 10 segundos utilizando Thread.sleep(10000) y se verifica que el texto del JLabel sea "100".
   3. **Resultado esperado**: El JLabel debe mostrar "100" después de 10 segundos.
4. **testMultipleTimerTicks()**
   1. **Objetivo**: Verificar que el puntaje se acumula correctamente después de múltiples intervalos del timer.
   2. **Prueba**: Se espera 31 segundos (tres ciclos de 10 segundos) y se verifica que el texto del JLabel sea "200".
   3. **Resultado esperado**: El JLabel debe mostrar "200" después de 30 segundos.
5. **testInitialBrainPoints()**
   1. **Objetivo**: Verificar que el puntaje inicial del BrainManager es correcto.
   2. **Prueba**: Se verifica que el puntaje inicial devuelto por getTotalGetPoints() sea igual a 50.
   3. **Resultado esperado**: El puntaje inicial debe ser "50".

* BrainsteinTest: La clase BrainsteinTest tiene como objetivo probar el comportamiento de la clase Brainstein, específicamente su funcionalidad para generar puntos ("cerebros") de forma pasiva a lo largo del tiempo
  1. **testGeneracionCerebros()**
     1. **Objetivo**: Verificar que la clase Brainstein genere correctamente 50 cerebros después de un tiempo determinado.
     2. **Prueba**:

Se crea una instancia de Brainstein.

Se llama al método performPassiveAction(), el cual inicia la generación de cerebros.

Se simula el paso del tiempo utilizando Thread.sleep(20100) (20.1 segundos), para permitir que el proceso de generación se complete.

Se llama al método stopGenerating() para detener el proceso de generación.

Se verifica que la cantidad de cerebros generados sea exactamente 50 usando getCerebrosGenerados().

* + 1. **Resultado esperado**: La cantidad de cerebros generados debe ser igual a **50**.
* ECIEvolutionTest: La clase ECIEvolutionTest se enfoca en validar el comportamiento de la clase ECIEvolution, una planta que evoluciona automáticamente a lo largo del tiempo, mejorando sus habilidades de ataque y su intervalo de disparo. La clase contiene pruebas que verifican el progreso a través de sus diferentes fases evolutivas y su capacidad de infligir daño.

1. **testEvolucionarPrimeraFase()**
2. **Objetivo**: Verificar la evolución de la planta después de **15 segundos**.
3. **Prueba**:

Se comprueba que la planta inicia con **500 de salud** y **0 de daño**.

Después de esperar 16 segundos, se valida que:

* + - * El intervalo de disparo es **3 segundos**.
      * El daño infligido es de **70 puntos**.

1. **Resultado esperado**: La planta evoluciona correctamente a la segunda fase.
2. **testEvolucionarSegundaFase()**
3. **Objetivo**: Verificar la evolución completa de la planta después de **40 segundos**.
4. **Prueba**:

Se espera 40 segundos para que la planta evolucione a su fase final.

Se verifica que:

* + - * El intervalo de disparo es de **1 segundo**.
      * El daño infligido es de **100 puntos**.

1. **Resultado esperado**: La planta alcanza su última fase evolutiva con las estadísticas correctas.
2. **testHacerDanioConCadaFase()**
3. **Objetivo**: Validar el daño infligido por la planta en cada fase evolutiva.
4. **Prueba**:

**Fase 1**: Se verifica que no causa daño inicial.

**Fase 2**: Después de 15 segundos, se verifica que causa **70 de daño**.

**Fase Final**: Después de 35 segundos adicionales, se verifica que causa **100 de daño**.

1. **Resultado esperado**: El daño de la planta aumenta progresivamente con cada fase.
2. **testNoEvolucionaMasAllaDeFaseFinal()**
3. **Objetivo**: Asegurarse de que la planta no evoluciona más allá de la fase final después de **60 segundos**.
4. **Prueba**:

Se espera 1 minuto para confirmar que:

* + - * El intervalo de disparo permanece en **1 segundo**.
      * El daño no supera los **100 puntos**.

1. **Resultado esperado**: La planta se mantiene en la fase final y no evoluciona más.
2. **testEvolucionarSinAccionesPrevias()**
3. **Objetivo**: Confirmar que la planta evoluciona automáticamente sin interacción directa.
4. **Prueba**:

Se llama al método performAction() y se verifica que la planta evoluciona correctamente con el tiempo.

1. **Resultado esperado**: La evolución ocurre automáticamente sin intervención manual.

* ECIPlantTest: La clase ECIPlantTest se centra en probar la funcionalidad específica de la planta ECIPlant. Esta planta tiene la capacidad especial de generar soles de 50 unidades, lo cual se valida en la prueba implementada.

1. **testGeneracionSolEspecial()**
2. **Objetivo**: Verificar que la planta ECIPlant genera soles especiales correctamente cuando realiza su acción pasiva.
3. **Prueba**:

Se crea una instancia de ECIPlant.

Se llama al método performPassiveAction() que ejecuta la generación de soles especiales.

Se compara el resultado esperado con el mensaje simulado de generación de soles:  
**"ECIPlant generó un sol especial de 50 unidades."**.

1. **Resultado esperado**: La generación de soles especiales de **50 unidades** se realiza correctamente.

* ECIZombieTest: La clase ECIZombieTest tiene como propósito verificar el comportamiento del ataque del ECIZombie. Esta prueba se enfoca en asegurarse de que el zombi pueda atacar correctamente a una planta y reducir su salud.

1. **testAtaqueECIZombie()**
2. **Objetivo**: Validar que el ECIZombie puede atacar a una planta (Guisante) y reducir su salud.
3. **Prueba**:

Se crea una instancia de ECIZombie.

Se crea una instancia de Guisante (planta con salud inicial de **300**).

Se llama al método attack del ECIZombie, pasándole la planta como objetivo.

Se verifica que la salud del Guisante es **menor a 300** después del ataque, lo que significa que el ataque fue exitoso.

1. **Resultado esperado**: La salud del Guisante disminuye tras el ataque del ECIZombie.

* GirasolTest: La clase GirasolTest tiene como objetivo verificar la generación de soles por parte del Girasol. Esta prueba evalúa que el girasol genere soles después de un tiempo determinado, validando la correcta ejecución del temporizador y su funcionalidad pasiva.

1. **testGeneracionSoles()**
2. **Objetivo**: Comprobar que el Girasol genera **50 puntos de soles** tras ejecutar su acción pasiva durante un tiempo determinado.
3. **Prueba**:
   * + Se crea una instancia del Girasol.
     + Se llama al método **performPassiveAction**, que inicia la generación de soles mediante un temporizador.
     + Se simula el paso del tiempo usando **Thread.sleep(20000)** (20 segundos), tiempo suficiente para generar una unidad de soles (50 puntos).
     + Se llama al método **stopGeneratingSun** para detener el temporizador antes de la validación.
     + Se verifica que el **total de soles generados** sea igual a **50** usando assertEquals.
4. **Resultado esperado**: El girasol genera exactamente **50 soles** después del tiempo especificado.

* GuisanteTest: La clase GuisanteTest está diseñada para verificar el ataque combinado de múltiples plantas Guisante hacia un Zombi. Esta prueba evalúa la capacidad de las plantas de infligir daño acumulativo al atacar a un objetivo común, asegurando que la lógica de ataque funcione correctamente.

1. **testAtaqueCombinadoPlantas()**
2. **Objetivo**: Validar que dos instancias de la planta Guisante infligen **daño acumulativo** al atacar al mismo ZombieBasico.
3. **Prueba**:
   * + Se crean **dos plantas Guisante** y un **ZombieBasico**.
     + Se simula el disparo de ambas plantas utilizando **performAction** y aplicando el daño correspondiente al zombi mediante **decreaseHealth**.
     + Se verifica que la **salud del zombi** sea menor a su salud inicial (100).
4. **Resultado esperado**: El zombi debe tener **menos de 100 puntos de salud** después de recibir los ataques combinados de ambas plantas.

* JugadorTest: La clase JugadorTest contiene pruebas unitarias para la clase Jugador, encargada de manejar las acciones de los jugadores en el juego. Se validan operaciones como la asignación de nombres, la gestión de recursos (soles) y las restricciones al realizar compras según el tipo de jugador.

1. **testNombreJugador()**
2. **Objetivo**: Verificar que los nombres de los jugadores se asignen correctamente al inicializarlos.
3. **Prueba**:
   * + Se crean dos jugadores: uno de plantas y otro de zombis.
     + Se comparan los nombres obtenidos mediante **getNombre** con los valores esperados.
4. **Resultado esperado**: Los nombres coinciden con los asignados al crear cada jugador.
5. **testIncrementarSoles()**
6. **Objetivo**: Validar que los soles disponibles de un jugador se incrementan correctamente.
7. **Prueba**:
   * + Se incrementan los soles disponibles del **jugador de plantas** usando el método **incrementarSoles**.
     + Se verifica que el nuevo valor de soles sea correcto.
8. **Resultado esperado**: Los soles se incrementan adecuadamente (de 100 a 150).
9. **testComprarPlantaFalloPorEsZombi()**
10. **Objetivo**: Confirmar que un jugador del tipo "zombi" **no puede comprar plantas**.
11. **Prueba**:
    * + Se intenta que el jugador zombi compre una planta (Girasol).
      + Se valida que el método **comprarPlanta** retorne false y que los **soles disponibles** no se modifiquen.
12. **Resultado esperado**:
    * 1. El método **comprarPlanta** retorna false.
      2. Los soles permanecen sin cambios.
13. **testComprarZombieFalloPorNoSerZombi()**
14. **Objetivo**: Verificar que un jugador del tipo "plantas" **no puede comprar zombis**.
15. **Prueba**:
    * + Se intenta que el jugador de plantas compre un zombi (ZombieBasico).
      + Se valida que el método **comprarZombie** retorne false y que los **soles disponibles** permanezcan igual.
16. **Resultado esperado**:
    * 1. El método **comprarZombie** retorna false.
      2. Los soles no cambian.

* NivelTest: La clase NivelTest contiene pruebas unitarias para la clase Nivel, la cual gestiona la lógica de un nivel del juego, incluyendo el tiempo restante, el despliegue de zombis y las condiciones de victoria o derrota.

1. **testActualizarNivel\_DisminuyeTiempo()**
   * **Objetivo**: Validar que el tiempo restante del nivel disminuye correctamente cuando se actualiza el nivel.
   * **Prueba**:
     + Se obtiene el tiempo inicial.
     + Se llama al método **actualizarNivel** para reducir el tiempo restante.
     + Se compara el tiempo restante con el valor esperado (tiempoInicial - 1).
   * **Resultado esperado**: El tiempo disminuye correctamente en cada actualización.
2. **testActualizarNivel\_ColocaZombiEnTablero()**
   * **Objetivo**: Confirmar que se colocan zombis en el tablero a medida que avanza el tiempo y se reducen en la cola.
   * **Prueba**:
     + Se avanza el tiempo en múltiplos de 10 segundos usando **actualizarNivel**.
     + Se verifica que la lista de **zombisEnCola** contiene menos zombis.
     + Se revisa si un zombi se colocó en el tablero.
   * **Resultado esperado**:
     + Los zombis son removidos de la cola.
     + Un zombi aparece en la posición esperada del tablero.
3. **testVerificarVictoria\_SinZombis()**
   * **Objetivo**: Validar la condición de victoria cuando no hay zombis en la cola ni en el tablero.
   * **Prueba**:
     + Se vacía la lista de **zombisEnCola**.
     + Se llama a **verificarVictoria** para confirmar si el nivel se considera ganado.
   * **Resultado esperado**: El método devuelve true indicando victoria.
4. **testVerificarDerrota\_SinZombisEnCasa()**
   * **Objetivo**: Confirmar que no se declara derrota si los zombis no han alcanzado la casa (posición final).
   * **Prueba**:
     + Se verifica si el método **verificarDerrota** devuelve false cuando no hay zombis en la posición final.
   * **Resultado esperado**: La derrota no se declara prematuramente.

* NuezTest: La clase NuezTest contiene pruebas unitarias para la clase Nuez, la cual representa una planta defensiva en el juego. Las pruebas verifican su capacidad para bloquear ataques de los zombis y manejar su salud.

1. **testBloqueoPorNuez()**
   * **Objetivo**: Validar que la **Nuez** puede bloquear el avance de un zombi mientras su salud disminuye tras recibir ataques.
   * **Prueba**:
     + Se coloca una **Nuez** y un **ZombieBasico** en el tablero en posiciones adyacentes.
     + Se simula un ataque del zombi a la nuez.
     + Se verifica que:
       - La salud de la nuez disminuye tras el ataque.
       - La nuez sigue viva después de recibir el daño.
   * **Resultado esperado**:
     + La salud de la nuez es menor a su valor inicial de 4000, pero no se destruye tras un solo ataque.
2. **testNuezRecibeDaño()**
   * **Objetivo**: Confirmar que la salud de la **Nuez** disminuye correctamente al recibir daño.
   * **Prueba**:
     + Se reduce la salud de la **Nuez** en 100 puntos usando el método **decreaseHealth**.
     + Se verifica que la salud actual sea 3900.
   * **Resultado esperado**:
     + La salud de la nuez refleja correctamente el daño recibido.
3. **testNuezEsDestruida()**
   * **Objetivo**: Validar que la **Nuez** se destruye correctamente cuando su salud llega a 0.
   * **Prueba**:
     + Se reduce la salud de la **Nuez** en 4000 puntos usando el método **decreaseHealth**.
     + Se verifica que:
       - La salud es igual a 0.
       - El método **isAlive** devuelve false.
   * **Resultado esperado**:
     + La nuez está destruida después de recibir daño total.

* PapaExplosivaTest: La clase PapaExplosivaTest contiene pruebas unitarias para la clase PapaExplosiva, una planta especial en el juego que, tras un tiempo de preparación, explota y elimina a los zombis cercanos. Las pruebas validan tanto su ciclo de activación como la efectividad de su explosión y comportamiento general.

1. **testExplosiónPapaExplosiva()**
   * **Objetivo**: Validar que la **PapaExplosiva** explota correctamente al estar activa, destruyendo un zombi y a sí misma.
   * **Prueba**:
     + Se reduce el tiempo de activación para preparar la explosión.
     + Se verifica que:
       - La papa está activa después de 14 turnos.
       - Al explotar, la salud del zombi es 0.
       - La papa ya no está viva tras explotar.
   * **Resultado esperado**:
     + La **PapaExplosiva** elimina al zombi y se destruye al explotar.
2. **testPapaExplosivaExplosion()**
   * **Objetivo**: Verificar que la **PapaExplosiva** marca correctamente su estado como "explotada" y elimina al zombi.
   * **Prueba**:
     + Se activa la **PapaExplosiva** tras 14 turnos y se la hace explotar.
     + Se verifica que:
       - La salud del zombi es 0 tras la explosión.
       - El estado **hasExploded** de la papa es true.
   * **Resultado esperado**:
     + La papa elimina al zombi y su estado refleja que ha explotado.
3. **testPapaExplosivaNoExplota()**
   * **Objetivo**: Validar que una **PapaExplosiva** recién colocada no explota automáticamente.
   * **Prueba**:
     + Se verifica el estado inicial de la **PapaExplosiva** (sin reducir su tiempo de activación).
     + Se comprueba que:
       - El estado **hasExploded** es false.
   * **Resultado esperado**:
     + La papa no explota inmediatamente después de ser colocada.
4. **testPapaExplosivaNoActiva()**
   * **Objetivo**: Confirmar que la **PapaExplosiva** no está activa ni destruida inicialmente.
   * **Prueba**:
     + Se verifica que:
       - La papa no está activa al ser colocada.
       - La papa está viva (isAlive() == true).
   * **Resultado esperado**:
     + La papa debe estar en estado inactivo y viva.
5. **testPapaExplosivaExplotaCorrectamente()**
   * **Objetivo**: Validar que la **PapaExplosiva** pasa correctamente por todo su ciclo de vida:  
     activación → explosión → destrucción.
   * **Prueba**:
     + Se activa la papa a través del método **performAction** en 14 turnos.
     + Se verifica que:
       - La papa está activa después de 14 acciones.
       - La salud del zombi es 0 tras la explosión.
       - La papa ya no está viva después de explotar.
   * **Resultado esperado**:
     + La papa elimina al zombi al explotar y se destruye.

* PlantaTest: La clase PlantaTest contiene pruebas unitarias diseñadas para validar el comportamiento básico de las plantas en el juego. La clase Planta es una entidad genérica que representa a las plantas, y estas pruebas aseguran que maneja correctamente su salud, incluyendo la reducción y límites en su valor.

1. **testPlantaDisminuirSalud()**
   * **Objetivo**: Verificar que el método **decreaseHealth** reduce correctamente la salud de la planta.
   * **Prueba**:
     + Se crea una instancia anónima de una planta llamada "Guisante" con 100 de salud.
     + Se llama al método **decreaseHealth(50)**.
     + Se verifica que:
       - La salud de la planta se reduce a 50.
   * **Resultado esperado**:
     + La salud de la planta debe disminuir correctamente al recibir daño.
2. **testPlantaNoSaludNegativa()**
   * **Objetivo**: Validar que la salud de una planta nunca sea negativa, incluso si recibe más daño que su salud actual.
   * **Prueba**:
     + Se crea una instancia anónima de una planta llamada "Girasol" con 50 de salud.
     + Se llama al método **decreaseHealth(100)**, un daño mayor que su salud actual.
     + Se verifica que:
       - La salud de la planta se establece en 0.
   * **Resultado esperado**:
     + La salud de la planta debe detenerse en 0, evitando valores negativos.

* ShovelTest: La clase ShovelTest contiene pruebas unitarias que validan el comportamiento de la herramienta Shovel (pala) dentro del juego. La pala se utiliza para remover plantas del tablero, y estas pruebas aseguran que su funcionalidad sea precisa, verificando tanto los casos exitosos como los errores.

1. **testRemovePlantSuccess()**
   * **Objetivo**: Verificar que la pala puede remover una planta correctamente de una celda válida del tablero.
   * **Prueba**:
     + Se coloca una planta en la posición (2, 2) del tablero.
     + Se verifica que la planta esté en la posición indicada.
     + Se utiliza la pala para remover la planta.
     + Se verifica que la celda quede vacía después de usar la pala.
   * **Resultado esperado**:
     + La pala remueve la planta exitosamente y la celda queda vacía.
2. **testRemovePlantFail\_NoPlant()**
   * **Objetivo**: Verificar que la pala no puede remover plantas de celdas vacías.
   * **Prueba**:
     + Se verifica que la celda (3, 3) esté vacía.
     + Se intenta usar la pala en la celda vacía.
     + Se verifica que el intento de remover una planta falle y no altere el tablero.
   * **Resultado esperado**:
     + La pala no realiza ninguna acción si no hay planta en la posición.
3. **testRemovePlantFail\_InvalidPosition()**
   * **Objetivo**: Asegurarse de que la pala maneje correctamente las posiciones inválidas (fuera de los límites del tablero).
   * **Prueba**:
     + Se intenta usar la pala en la posición (-1, -1), fuera de los límites del tablero.
     + Se verifica que se lance una excepción **IllegalArgumentException** con un mensaje claro que indique que la posición es inválida.
   * **Resultado esperado**:
     + La pala lanza una excepción y no realiza ninguna acción en posiciones inválidas.

* TableroTest: La clase TableroTest contiene pruebas unitarias para validar el comportamiento de la clase Tablero, que es el componente principal donde se desarrollan las interacciones del juego. Estas pruebas verifican la colocación de plantas y zombis, la interacción con elementos como soles y podadoras, y el movimiento de los personajes.

1. **testColocarPlanta()**
   * **Objetivo**: Verificar que se puedan colocar plantas en celdas válidas del tablero.
   * **Prueba**:
     + Se coloca una planta en una celda vacía y se verifica que sea exitoso.
     + Se intenta colocar otra planta en la misma celda y se espera un fallo.
   * **Resultado esperado**:
     + Las plantas solo pueden colocarse en celdas vacías.
2. **testColocarZombie()**
   * **Objetivo**: Verificar que se puedan colocar zombis en celdas válidas del tablero.
   * **Prueba**:
     + Se coloca un zombi en una celda vacía y se verifica que sea exitoso.
     + Se intenta colocar otro zombi en la misma celda y, **por error**, actualmente se permite.
   * **Nota Importante**:
     + La segunda validación está incorrecta, debería usarse **assertFalse** para evitar colocar zombis en celdas ocupadas.
3. **testMovimientoMultipleZombis()**
   * **Objetivo**: Verificar el movimiento independiente de múltiples zombis en el tablero.
   * **Prueba**:
     + Se colocan dos zombis en diferentes posiciones.
     + Se simula su movimiento y se verifica que sigan vivos después del desplazamiento.
   * **Resultado esperado**:
     + Los zombis deben moverse correctamente sin afectar su estado de vida.
4. **testRecoleccionSoles()**
   * **Objetivo**: Verificar que el tablero recolecte los soles generados por diferentes plantas.
   * **Prueba**:
     + Se colocan un **Girasol** y una **ECIPlant** en el tablero.
     + Se simula el paso del tiempo para que las plantas generen soles.
     + Se llama al método **recogerSoles** y se verifica que la suma de los soles sea correcta.
   * **Resultado esperado**:
     + Se deben recolectar 75 soles (50 de la **ECIPlant** y 25 del **Girasol**).
5. **testLawnMowerEliminaZombies()**
   * **Objetivo**: Validar que las podadoras eliminen a los zombis cuando son activadas.
   * **Prueba**:
     + Se coloca un zombi en una fila con una podadora.
     + Se activa la podadora y se verifica que la celda donde estaba el zombi quede vacía.
   * **Resultado esperado**:
     + El zombi debe ser eliminado de la celda.
6. **testLawnMowerNoReutilizable()**
   * **Objetivo**: Asegurarse de que las podadoras no sean reutilizables después de activarse.
   * **Prueba**:
     + Se activa una podadora sin zombis presentes.
     + Se verifica que el estado de la podadora indique que ya no es utilizable.
   * **Resultado esperado**:
     + Las podadoras solo deben usarse una vez.

* ZombieBasicoTest: La clase ZombieBasicoTest contiene pruebas unitarias para validar el comportamiento de la clase ZombieBasico, que representa un zombi estándar en el juego. Estas pruebas evalúan acciones clave como el ataque a plantas y el movimiento del zombi en el tablero.

1. **testAtaqueZombieBasico()**
   * **Objetivo**: Verificar que un **ZombieBasico** puede atacar a una planta y reducir su salud correctamente.
   * **Prueba**:
     + Se crea un zombi y una planta (**Girasol**).
     + El zombi ataca a la planta.
     + Se simula un tiempo de espera de 600 ms para asegurar que el ataque se registre (el intervalo de ataque del zombi es de 0.5 segundos).
     + Se verifica que la salud de la planta sea menor a 300, lo que indica que recibió daño.
   * **Resultado esperado**:
     + La salud de la planta debe disminuir correctamente después del ataque.
2. **testMovimientoZombieBasico()**
   * **Objetivo**: Validar que el **ZombieBasico** puede moverse en el tablero correctamente.
   * **Prueba**:
     + Se crea un zombi y un tablero de dimensiones 5x5.
     + El zombi se coloca en la posición (4,4) del tablero.
     + Se simula el movimiento del zombi llamando al método **move**.
     + Se verifica que el zombi siga vivo después de moverse.
   * **Resultado esperado**:
     + El zombi debe mantenerse vivo tras realizar su movimiento.

* ZombieConoTest: La clase ZombieConoTest contiene pruebas unitarias para validar el comportamiento de la clase ZombieCono, que es una variante del zombi básico con mayor resistencia inicial debido a su "cono protector". Estas pruebas cubren los aspectos clave de esta clase, incluyendo la resistencia adicional proporcionada por el cono, el manejo de daños, y su interacción con las plantas a través de ataques.

1. **testAtaqueZombieCono()**
   * **Objetivo**: Validar que el zombi con cono maneja correctamente la transición entre la salud del cono y la salud base.
   * **Prueba**:
     + Se simula la destrucción completa del cono y se valida que la salud base no se ve afectada.
     + Se verifica que, una vez destruido el cono, los daños afectan directamente la salud base del zombi.
   * **Resultado esperado**:
     + El zombi pierde el cono correctamente y la salud base responde adecuadamente al daño recibido.
2. **testZombieConoRecibirDanio()**
   * **Objetivo**: Verificar que el zombi con cono reduce su salud correctamente al recibir daño.
   * **Prueba**:
     + Se crea un zombi con cono, se aplica daño, y se valida que su salud se reduce de acuerdo al valor del daño.
   * **Resultado esperado**:
     + La salud se actualiza correctamente tras recibir daño.
3. **testZombieConoResistencia()**
   * **Objetivo**: Validar la resistencia adicional proporcionada por el cono y su interacción con el daño recibido.
   * **Prueba**:
     + Se aplican varios ataques que primero agotan la salud del cono y luego afectan la salud base del zombi.
   * **Resultado esperado**:
     + El zombi mantiene su resistencia inicial con el cono, y el daño sobrante se transfiere a la salud base tras romper el cono.
4. **testZombieConoSinCono()**
   * **Objetivo**: Asegurarse de que el zombi responde correctamente tras perder el cono.
   * **Prueba**:
     + Se aplica daño suficiente para destruir el cono y se valida que la salud base queda intacta.
   * **Resultado esperado**:
     + El zombi pierde el cono, pero conserva su salud base intacta hasta recibir más daño.
5. **testZombieConoRecibeDañoExcesivo()**
   * **Objetivo**: Verificar que el daño sobrante tras destruir el cono se aplica correctamente a la salud base.
   * **Prueba**:
     + Se aplica un daño superior al valor de la salud del cono y se valida que el exceso afecta la salud base del zombi.
   * **Resultado esperado**:
     + La salud base se reduce de forma precisa tras destruir el cono.
6. **testZombieConoAtacaPlanta()**
   * **Objetivo**: Validar que el zombi con cono puede atacar a una planta, causando daño en intervalos definidos.
   * **Prueba**:
     + Se utiliza un mecanismo de conteo para simular los ataques del zombi y verificar que se realizan en los intervalos esperados.
   * **Resultado esperado**:
     + La planta recibe daño acumulativo por cada ataque realizado dentro del tiempo establecido.
7. **testZombieConoGetCosto()**
   * **Objetivo**: Comprobar que el método **getCosto()** retorna el valor correcto para un zombi con cono.
   * **Prueba**:
     + Se crea un zombi con cono y se valida que el costo devuelto sea el esperado (150).
   * **Resultado esperado**:
     + El método retorna el costo correcto de 150.

* ZombieCubetaTest: La clase ZombieCubetaTest contiene pruebas unitarias para la clase ZombieCubeta, un tipo de zombi que cuenta con una cubeta como protección adicional. Estas pruebas evalúan aspectos clave como la resistencia adicional de la cubeta, la transferencia de daño a la salud base tras la pérdida de la cubeta, el costo del zombi y su comportamiento en ataques y movimientos.

1. **testDañoAcumuladoZombieCubeta()**
   * **Objetivo**: Verificar el comportamiento del zombi cuando recibe daño acumulado suficiente para destruir la cubeta y reducir su salud base.
   * **Prueba**:
     + Se aplica daño para destruir la cubeta y luego reducir la salud base.
     + Se valida que la cubeta se pierda y que el daño sobrante afecte correctamente la salud base.
   * **Resultado esperado**:
     + La cubeta se destruye correctamente y el daño sobrante se aplica a la salud base.
2. **testZombieCubetaResistenciaExtra()**
   * **Objetivo**: Comprobar que la resistencia de la cubeta absorbe daño parcial antes de transferirlo a la salud base.
   * **Prueba**:
     + Se aplica daño parcial que no supera la resistencia de la cubeta.
     + Se valida que la salud base no se vea afectada mientras la cubeta resiste.
   * **Resultado esperado**:
     + La cubeta conserva su vida restante y la salud base no se reduce.
3. **testZombieCubetaSinResistencia()**
   * **Objetivo**: Verificar que el zombi sea eliminado cuando recibe daño total que excede la suma de la vida de la cubeta y la salud base.
   * **Prueba**:
     + Se aplica un daño superior al total combinado de la cubeta y la salud base.
   * **Resultado esperado**:
     + El zombi queda sin salud y es eliminado.
4. **testZombieCubetaPierdeCubeta()**
   * **Objetivo**: Validar que el zombi pierda correctamente la cubeta tras recibir daño suficiente.
   * **Prueba**:
     + Se inflige exactamente la cantidad de daño necesaria para destruir la cubeta.
   * **Resultado esperado**:
     + La cubeta se destruye y la salud base queda intacta.
5. **testZombieCubetaDañoExcesivo()**
   * **Objetivo**: Verificar que el daño excedente tras destruir la cubeta se transfiera correctamente a la salud base.
   * **Prueba**:
     + Se aplica daño suficiente para destruir la cubeta y reducir la salud base.
   * **Resultado esperado**:
     + El daño sobrante se aplica correctamente a la salud base, y el zombi es eliminado si el daño supera la salud restante.
6. **testZombieCubetaGetCosto()**
   * **Objetivo**: Comprobar que el método **getCosto()** retorna el valor correcto para el ZombieCubeta.
   * **Prueba**:
     + Se valida que el costo del zombi sea igual al valor definido en la clase (200).
   * **Resultado esperado**:
     + El método devuelve el costo correcto de 200.
7. **testZombieCubetaMove()**
   * **Objetivo**: Validar que el método **move()** imprime correctamente el mensaje esperado al mover el zombi.
   * **Prueba**:
     + Se captura la salida estándar mientras se ejecuta el método.
     + Se compara el mensaje impreso con el esperado.
   * **Resultado esperado**:
     + El mensaje coincide con el formato definido en el método.
8. **testZombieCubetaAttack()**
   * **Objetivo**: Comprobar que el ZombieCubeta realiza ataques consecutivos a una planta, reduciendo su salud en intervalos definidos.
   * **Prueba**:
     + Se utiliza una planta con salud inicial de 200 y se valida el daño acumulado tras dos ataques.
     + Se espera un intervalo de 0.5 segundos entre ataques.
   * **Resultado esperado**:
     + La salud de la planta se reduce correctamente tras dos ataques consecutivos.

* ZombieDisparadorTest: La clase ZombieDisparadorTest contiene pruebas unitarias para la clase abstracta ZombieDisparador, la cual representa un zombi que dispara proyectiles de manera periódica. Para realizar las pruebas, se utiliza una clase concreta de prueba llamada ZombieDisparadorTestImpl, que implementa los métodos abstractos y lleva un contador para rastrear las veces que el zombi dispara.

1. **testStartShooting()**
   * **Objetivo**: Verificar que el zombi comienza a disparar proyectiles de manera periódica cuando se invoca el método **startShooting()**.
   * **Prueba**:
     + Se crea una instancia de **ZombieDisparadorTestImpl** con un contador para los disparos.
     + Se inicia el disparo y se espera el tiempo suficiente para que ocurran al menos dos disparos (2.1 segundos).
     + Se detiene el disparo tras la prueba.
   * **Resultado esperado**:
     + El contador de disparos debe ser al menos 2, indicando que el método **shoot()** se ejecutó correctamente.
2. **testStopShooting()**
   * **Objetivo**: Verificar que el zombi deja de disparar cuando se invoca el método **stopShooting()**.
   * **Prueba**:
     + Se crea una instancia de **ZombieDisparadorTestImpl**.
     + Se inicia el disparo y se espera un tiempo para que ocurra al menos un disparo (1.1 segundos).
     + Se detiene el disparo y se guarda el valor del contador de disparos.
     + Se espera más tiempo para comprobar que el contador no aumenta después de detener los disparos.
   * **Resultado esperado**:
     + El contador de disparos no debe incrementarse después de detener el disparo.
3. **testDecreaseHealthStopsShooting()**
   * **Objetivo**: Verificar que el zombi deja de disparar automáticamente cuando su salud llega a 0.
   * **Prueba**:
     + Se crea una instancia de **ZombieDisparadorTestImpl**.
     + Se inicia el disparo.
     + Se reduce la salud del zombi a 0 utilizando el método **decreaseHealth()**.
     + Se espera un tiempo suficiente para comprobar si el método **shoot()** se sigue ejecutando.
   * **Resultado esperado**:
     + El contador de disparos no debe incrementarse cuando la salud del zombi es 0.

**Clase de Prueba ZombieDisparadorTestImpl**

Esta clase es una implementación concreta de **ZombieDisparador** utilizada exclusivamente para las pruebas. Implementa los métodos abstractos **shoot()**, **move()** y **attack()**, pero solo **shoot()** realiza una acción real, incrementando un contador de disparos para rastrear cuántas veces se ejecuta.

**Propósito**:

* + Proporcionar un entorno controlado y verificable para las pruebas del comportamiento del disparo periódico en **ZombieDisparador**.
* ZombiePasivoTest: La clase ZombiePasivoTest contiene pruebas unitarias para la clase abstracta ZombiePasivo, que representa un zombi que no ataca ni realiza acciones agresivas, pero puede ejecutar acciones pasivas específicas. Estas pruebas verifican la salud y el comportamiento básico del zombi pasivo, asegurando que se ajusta a las restricciones de su naturaleza.

1. **testZombiePasivoNoAtaca()**
   * **Objetivo**: Verificar que un zombi pasivo no tenga capacidad de ataque y que su salud disminuya correctamente al recibir daño.
   * **Prueba**:
     + Se crea una instancia de **ZombiePasivo** con 100 de salud y 0 de daño.
     + Se verifica que el daño del zombi sea 0.
     + Se reduce su salud en 50 y se verifica que el valor de la salud se actualice correctamente.
   * **Resultado esperado**:
     + El daño del zombi debe permanecer en 0.
     + La salud debe disminuir en 50, quedando en 50.
2. **testZombiePasivoMuere()**
   * **Objetivo**: Verificar que el zombi pasivo sea destruido al recibir daño suficiente para reducir su salud a 0.
   * **Prueba**:
     + Se crea una instancia de **ZombiePasivo** con 100 de salud.
     + Se reduce la salud del zombi en 100.
     + Se verifica que la salud sea 0 y que el zombi sea considerado no vivo (**isAlive()** debe retornar false).
   * **Resultado esperado**:
     + La salud del zombi debe ser 0.
     + El método **isAlive()** debe retornar false, indicando que el zombi ha sido destruido.