Escuela Colombiana de Ingeniería Julio Garavito



**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

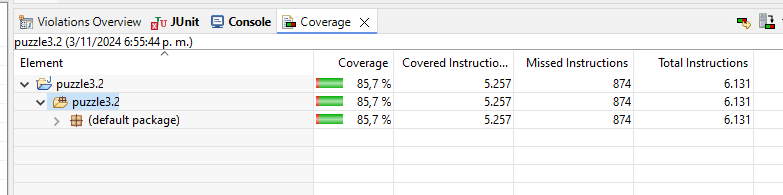
María Irma Diaz Rozo

El análisis dinámico de cobertura de código se realizó en Eclipse 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 85.7%, 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:

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

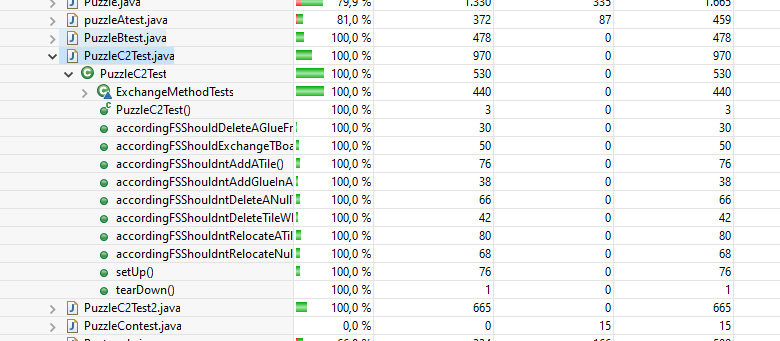
## **Cobertura de Código por Clase:**

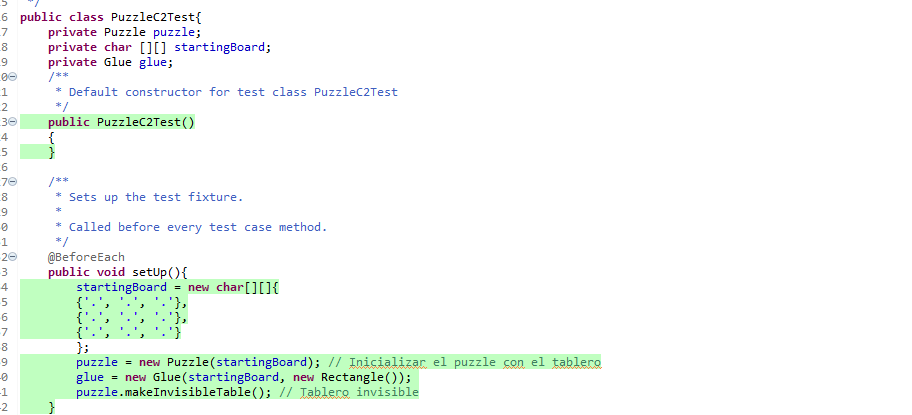
A continuación, se muestra el porcentaje de cobertura y las instrucciones cubiertas y faltantes para cada clase:

1. \*\*Canvas.java\*\*: Cobertura del 79.5%, con 287 instrucciones cubiertas de 361.
2. \*\*Glue.java\*\*: Cobertura del 78.8%, con 510 instrucciones cubiertas de 647.
3. \*\*Puzzle.java\*\*: Cobertura del 79.9%, con 1,330 instrucciones cubiertas de 1,665.
4. \*\*PuzzleAtest.java\*\*: Cobertura completa del 100%, con todas las 372 instrucciones cubiertas.
5. \*\*PuzzleBtest.java\*\*: Cobertura completa del 100%, con todas las 478 instrucciones cubiertas.
6. \*\*PuzzleC2Test.java\*\*: Cobertura completa del 100%, con todas las 665 instrucciones cubiertas.
7. \*\*PuzzleC2test2.java\*\*: Cobertura completa del 100%, con todas las 15 instrucciones cubiertas.
8. \*\*PuzzleContest.java\*\*: Cobertura completa del 100%, con todas las 334 instrucciones cubiertas.
9. \*\*Rectangle.java\*\*: Cobertura del 66.8%, con 334 instrucciones cubiertas de 500.
10. \*\*Tilt.java\*\*: Cobertura del 83.8%, con 311 instrucciones cubiertas de 371.

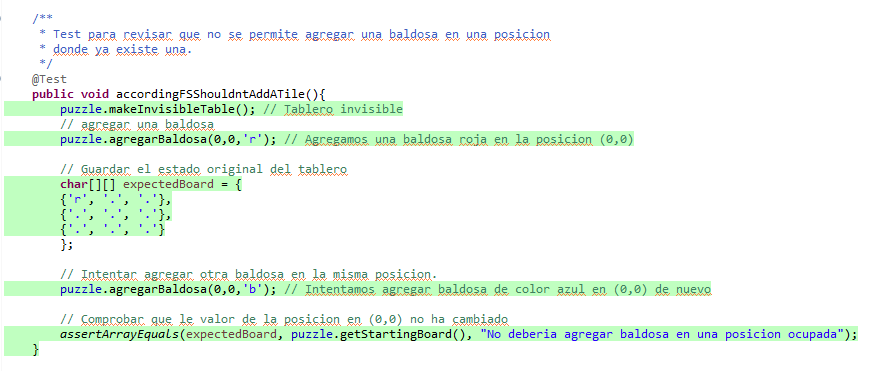
Una vez mostrado cada porcentaje simplemente nos enfocaremos en este documento en las clases de pruebas que se realizaron a lo largo de este proyecto, y se dejaran de ultimas las pruebas de aceptación hechas por cada uno:

1. **PuzzleC2Test:** La clase PuzzleC2Test contiene un conjunto de pruebas unitarias diseñadas para verificar el comportamiento correcto de varias funcionalidades de la clase Puzzle. Todas las pruebas se ejecutan satisfactoriamente, indicadas por el color verde en el reporte de cobertura de código, lo que significa que cada una de las funciones probadas cumple con las especificaciones definidas.

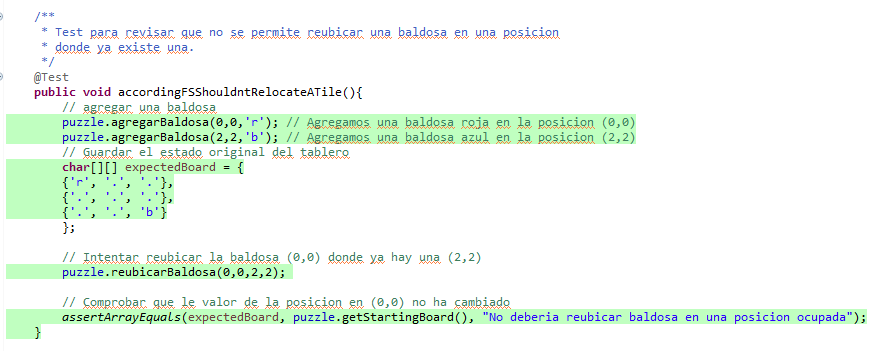
Primero como podemos ver tenemos un código que va antes de todos los test, este sirve para tomar como referencia el tablero y todas las pruebas tomaran este método para probar diferentes métodos implementados a lo largo de puzzle y cómo podemos ver todo está en verde, lo que significa que está bien implementado.

**accordingFSShouldntAddATile**

* **Descripción:** Esta prueba verifica que no se permita agregar una baldosa en una posición donde ya existe una baldosa.
* **Procedimiento:** Se agrega una baldosa roja en la posición (0,0) y luego se intenta agregar otra baldosa azul en la misma posición.
* **Resultado Esperado:** El color de la baldosa en (0,0) permanece como rojo, indicando que la segunda inserción no fue permitida. La prueba pasa, asegurando que el sistema evita la duplicación en una posición ocupada.

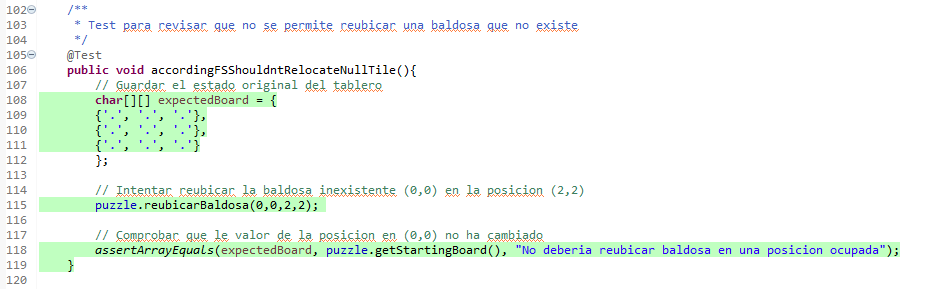
**accordingFSShouldntRelocateATile**

* **Descripción:** Verifica que el sistema no permita reubicar una baldosa en una posición que ya está ocupada por otra baldosa.
* **Procedimiento:** Se coloca una baldosa roja en (0,0) y una azul en (2,2). Luego, se intenta reubicar la baldosa roja en (2,2).
* **Resultado Esperado:** El estado de (0,0) y (2,2) no cambia, manteniendo sus valores iniciales. Esto asegura que el sistema evita la reubicación en posiciones ocupadas.

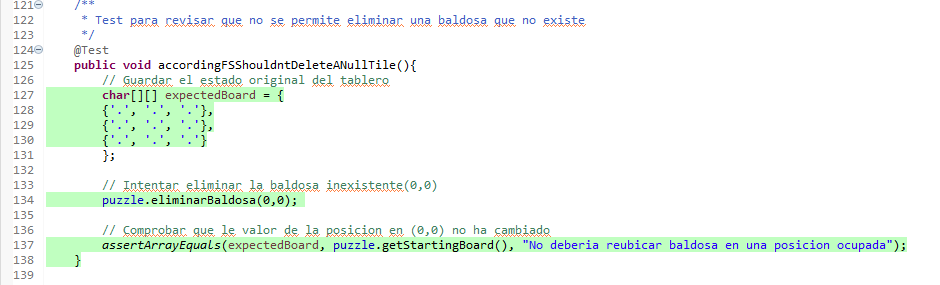


**accordingFSSShouldntRelocateNullTile:**

* **Descripción:** Esta prueba asegura que no se permita reubicar una baldosa inexistente.
* **Procedimiento:** Se intenta mover una baldosa en la posición (0,0) (vacía) a (2,2).
* **Resultado Esperado:** Ningún cambio ocurre en el tablero, y la posición (0,0) sigue vacía, validando que el sistema previene reubicaciones de baldosas inexistentes.

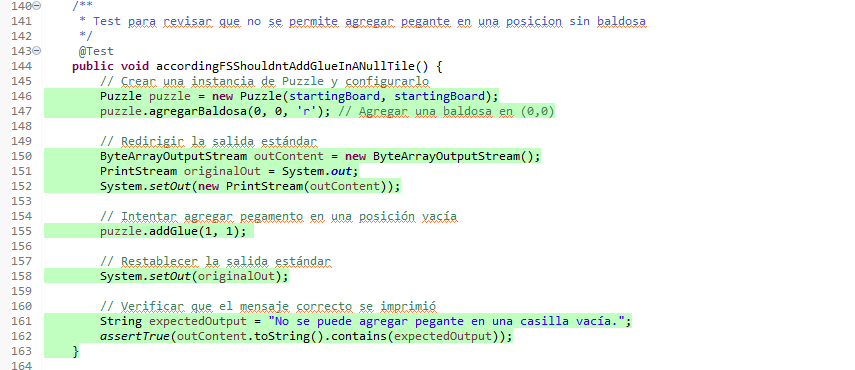
**accordingFSShouldntDeleteANullTile**

* **Descripción:** Asegura que no se pueda eliminar una baldosa en una posición vacía.
* **Procedimiento:** Se intenta eliminar una baldosa en la posición (0,0), que está vacía.
* **Resultado Esperado:** No ocurre ningún cambio en el tablero, confirmando que el sistema no permite la eliminación de baldosas inexistentes.

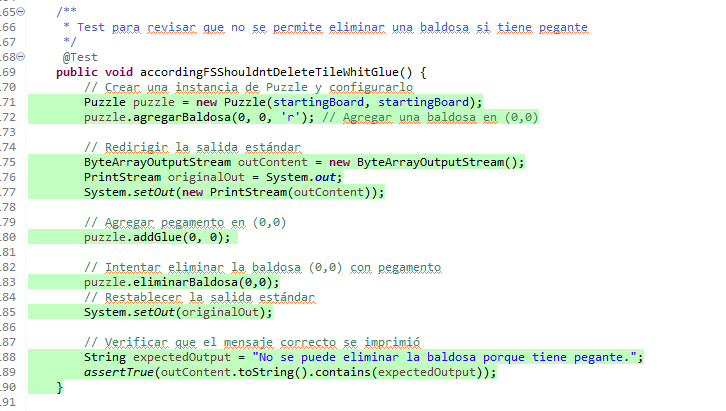


**accordingFSShouldntAddGlueInANullTile**

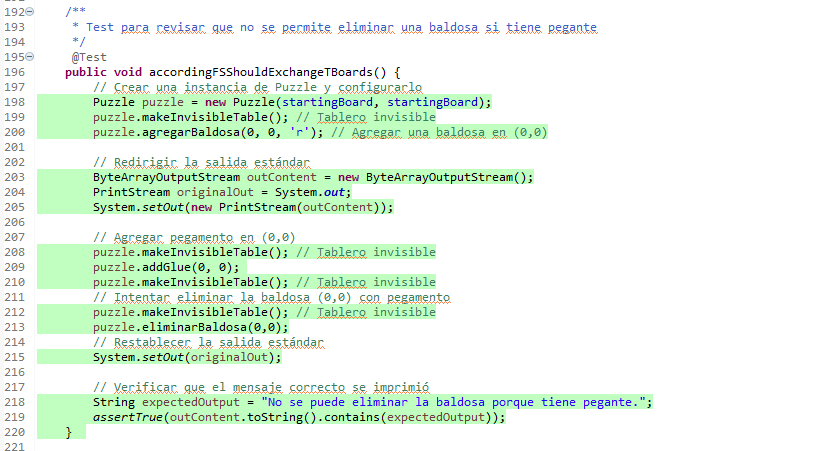
* **Descripción:** Verifica que no se pueda agregar pegamento en una posición sin baldosa.
* **Procedimiento:** Intenta agregar pegamento en (1,1), que está vacía.
* **Resultado Esperado:** El sistema imprime un mensaje de error indicando que no se puede agregar pegamento en una casilla vacía. La prueba pasa, validando el comportamiento esperado.

**accordingFSShouldntDeleteTileWithGlue**

* **Descripción:** Comprueba que no se pueda eliminar una baldosa que tenga pegamento aplicado.
* **Procedimiento:** Se agrega una baldosa en (0,0) y se le aplica pegamento. Luego se intenta eliminar la baldosa.
* **Resultado Esperado:** El sistema muestra un mensaje de error indicando que no se puede eliminar la baldosa debido al pegamento. La prueba confirma que las baldosas con pegamento no son eliminables.

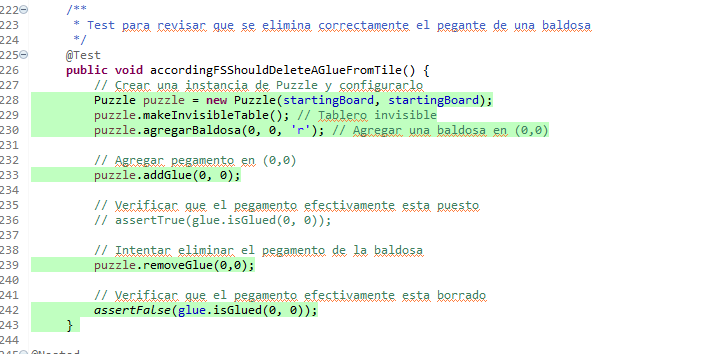
**accordingFSShouldExchangeTBoards**

* **Descripción:** Prueba la funcionalidad de intercambio de tableros en el sistema.
* **Procedimiento:** Se configura el tablero de inicio y final, luego se realiza el intercambio.
* **Resultado Esperado:** Los tableros se intercambian correctamente, y el estado final del tablero coincide con el esperado.

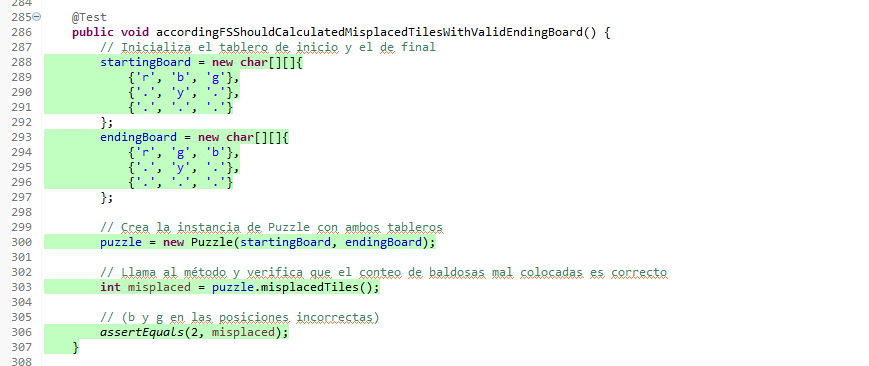


**accordingFSShouldDeleteAGlueFromTile**

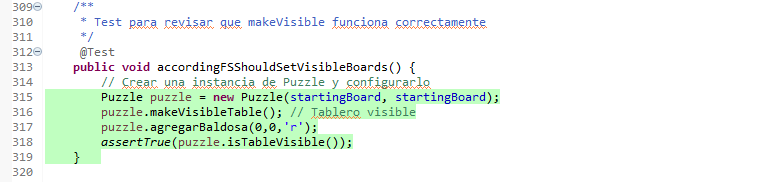
* **Descripción:** Verifica que se pueda eliminar el pegamento de una baldosa.
* **Procedimiento:** Se aplica pegamento a una baldosa y luego se elimina.
* **Resultado Esperado:** El pegamento se elimina correctamente, asegurando que el sistema permite la eliminación del pegamento según lo esperado

**accordingFSShouldCalculatedMisplacedTilesWithValidEndingBoard**

* **Descripción:** Calcula el número de baldosas que están en una posición incorrecta comparando los tableros inicial y final.
* **Procedimiento:** Configura un tablero de inicio y otro de final y cuenta las posiciones incorrectas.
* **Resultado Esperado:** La cuenta de baldosas mal colocadas es precisa. La prueba confirma el correcto funcionamiento del cálculo de posiciones incorrectas.

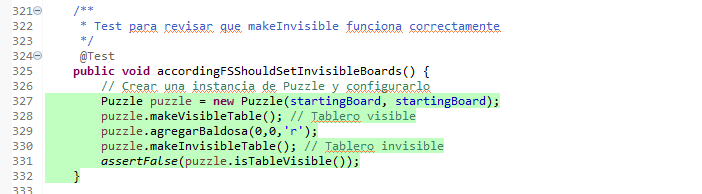
**accordingFSShouldSetVisibleBoards**

* **Descripción:** Verifica que el método makeVisibleTable funcione correctamente al hacer visible el tablero.
* **Procedimiento:** Se hace visible el tablero y se coloca una baldosa en (0,0).
* **Resultado Esperado:** La visibilidad del tablero se establece correctamente en verdadero, confirmando la funcionalidad.

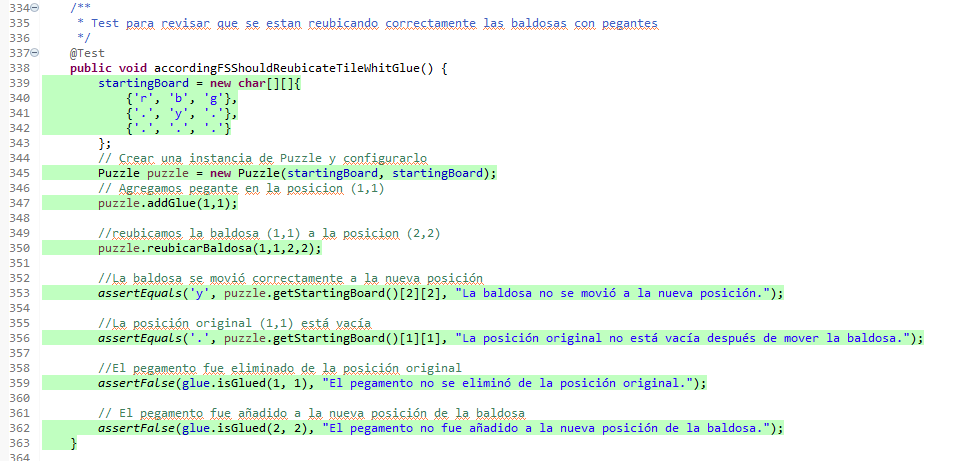


**accordingFSShouldSetInvisibleBoards**

* **Descripción:** Prueba que el método makeInvisibleTable funciona al hacer invisible el tablero.
* **Procedimiento:** El tablero se establece en modo visible y luego en invisible.
* **Resultado Esperado:** La visibilidad del tablero se establece correctamente en falso, asegurando que el método funciona según lo esperado.

**accordingFSShouldReubicateTileWithGlue**

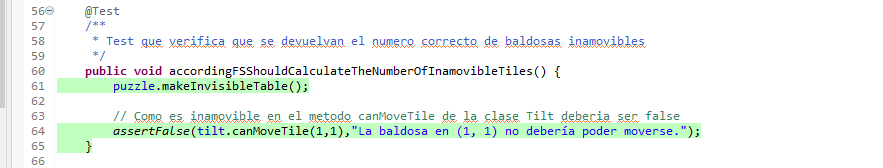
* **Descripción:** Verifica que una baldosa con pegamento se pueda reubicar y que el pegamento sea eliminado de la posición original.
* **Procedimiento:** Se coloca una baldosa en (1,1), se le aplica pegamento y luego se reubica a (2,2).
* **Resultado Esperado:** La baldosa se mueve a (2,2), y el pegamento se transfiere a la nueva posición mientras se elimina de la original.



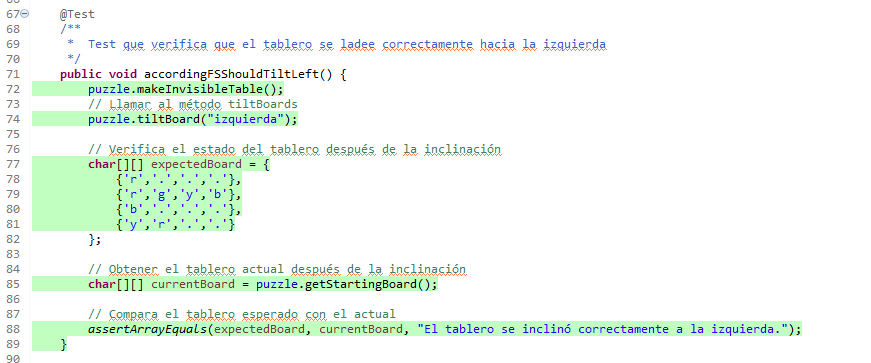
1. **PuzzleC2Test2**: La clase PuzzleC2Test2 contiene un conjunto de pruebas unitarias que verifican funcionalidades adicionales de la clase Puzzle, incluyendo la inclinación del tablero en diferentes direcciones y la función de parpadeo de baldosas. Todas las pruebas se ejecutan exitosamente, lo cual indica que las funcionalidades cumplen con los requisitos especificados.

**accordingFSShouldCalculateTheNumberOfInamovibleTiles**

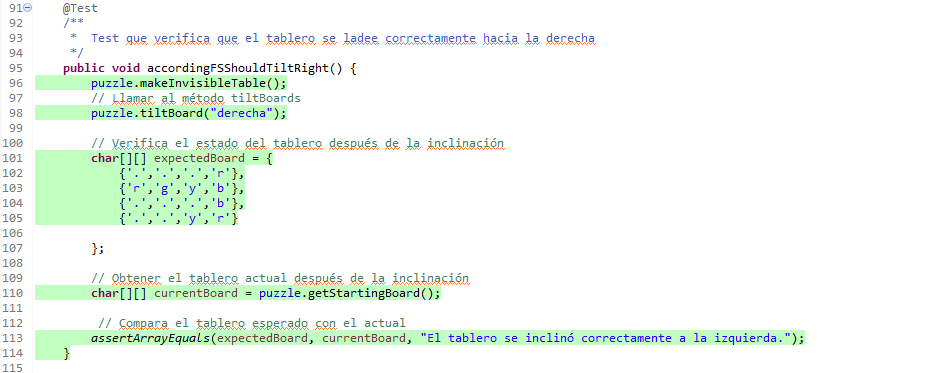
* **Descripción:** Esta prueba verifica que el método canMoveTile de la clase Tilt identifique correctamente una baldosa como inamovible.
* **Procedimiento:** Se configura una baldosa en la posición (1,1), y se verifica que el método canMoveTile devuelva false, indicando que la baldosa no puede moverse.
* **Resultado Esperado:** La baldosa en (1,1) es inamovible, y el sistema devuelve false como esperado.

**accordingFSShouldTiltLeft**

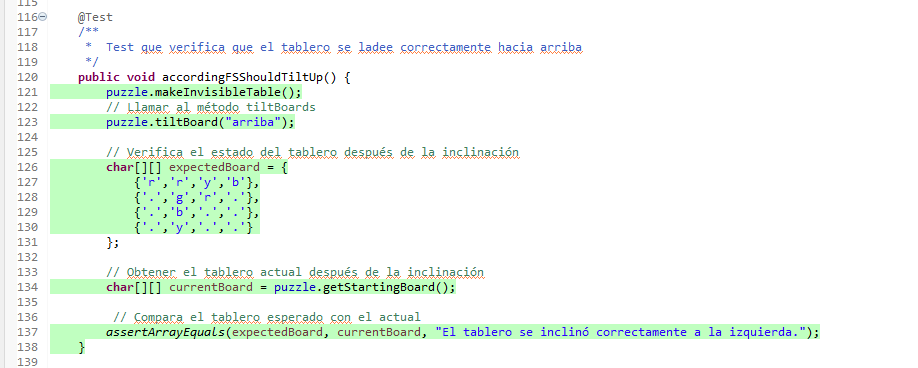
* **Descripción:** Prueba la funcionalidad de inclinar el tablero hacia la izquierda.
* **Procedimiento:** Se llama al método tiltBoard con la dirección "izquierda" y se verifica que el tablero cambie de acuerdo con la inclinación.
* **Resultado Esperado:** El estado del tablero después de la inclinación coincide con el tablero esperado, validando que las baldosas se movieron correctamente hacia la izquierda.

**accordingFSShouldTiltRight**

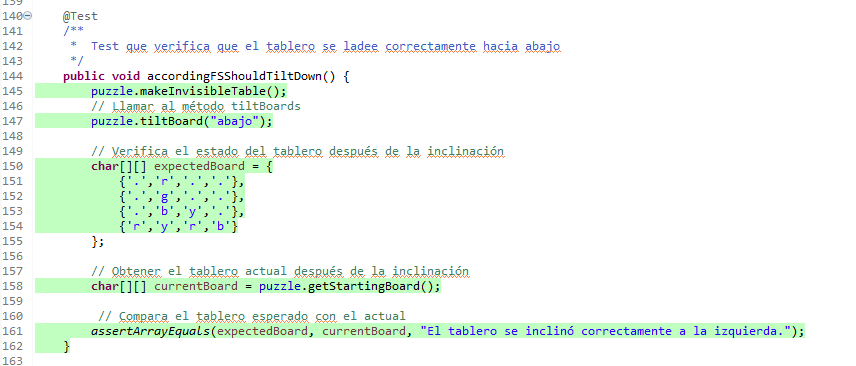
* **Descripción:** Verifica que el tablero se incline correctamente hacia la derecha.
* **Procedimiento:** Se llama al método tiltBoard con la dirección "derecha" y se compara el estado del tablero después de la inclinación con el tablero esperado.
* **Resultado Esperado:** Las baldosas se mueven hacia la derecha según lo esperado, y el estado final del tablero coincide con el tablero esperado.

**accordingFSShouldTiltUp**

* **Descripción:** Comprueba que el tablero se incline correctamente hacia arriba.
* **Procedimiento:** Se llama al método tiltBoard con la dirección "arriba" y se verifica el estado del tablero después de la inclinación.
* **Resultado Esperado:** El estado del tablero coincide con el esperado, indicando que las baldosas se movieron correctamente hacia arriba.

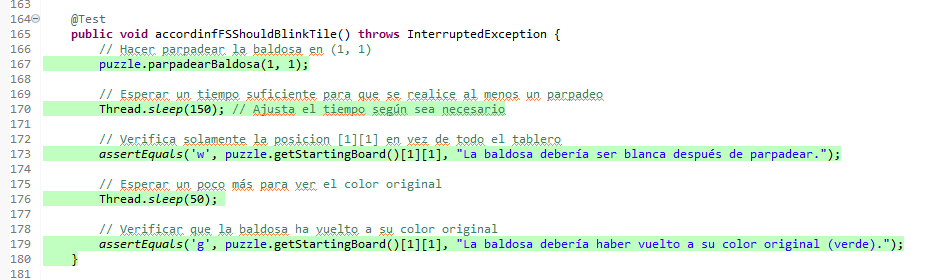
**accordingFSShouldTiltDown**

* **Descripción:** Verifica la inclinación del tablero hacia abajo.
* **Procedimiento:** Se llama al método tiltBoard con la dirección "abajo" y se compara el estado del tablero tras la inclinación.
* **Resultado Esperado:** El estado final del tablero coincide con el tablero esperado, confirmando que la inclinación hacia abajo se realizó correctamente.



**accordingFSShouldBlinkTile**

* **Descripción:** Esta prueba verifica que la función parpadearBaldosa cambie temporalmente el color de una baldosa y luego lo restaure.
* **Procedimiento:** Se hace parpadear la baldosa en la posición (1,1), y se verifica que cambie a color blanco temporalmente y luego vuelva a su color original.
* **Resultado Esperado:** La baldosa parpadea en blanco y luego regresa a su color original, validando que la función de parpadeo funciona correctamente.



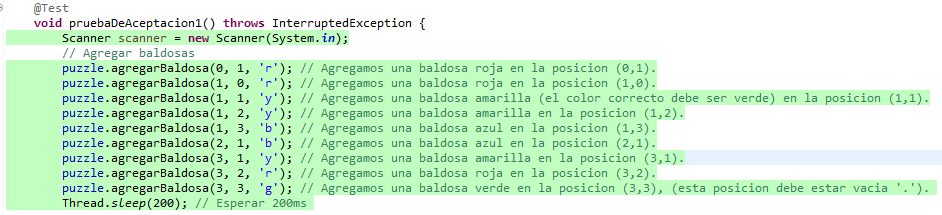
1. **PuzzleATest**

* **Descripción:** Prueba de aceptación en la que probamos distintos métodos de clases como Puzzle, Retangle, Tilt y PuzzleContest.
* **Procedimiento:**

**Paso 1:**

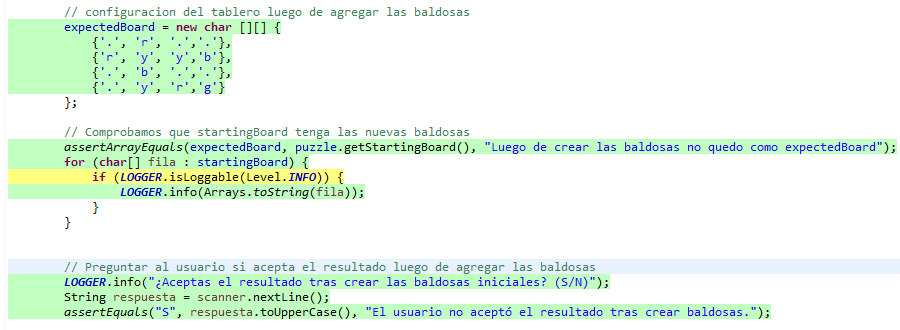
* **Accion:** Inicializamos los dos tableros (startingBoard y endingBoard) el primero completamente vacío y el segundo con la configuración final o esperada, luego procedemos con el metodo agregarBaldosa a llenar startingBoard.
* **Resultado esperado:** Se espera que mediante el método agregarBaldosa de la clase Puzzle se coloquen correctamente todas las baldosas en el tablero inicial.





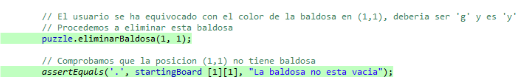
**Paso 2:**

* **Accion:** Creamos una variable expectedBoard para la primera prueba, en la cual verificaremos que, en efecto, las baldosas se hayan colocado en su posición correspondiente.
* **Resultado esperado:** Usando assertArrayEquals verificamos que todo vaya como se espera. Luego le pediremos al usuario que confirme el resultado, poniendo un mensaje en consola con LOGGER para evitar cualquier violación de estilo.

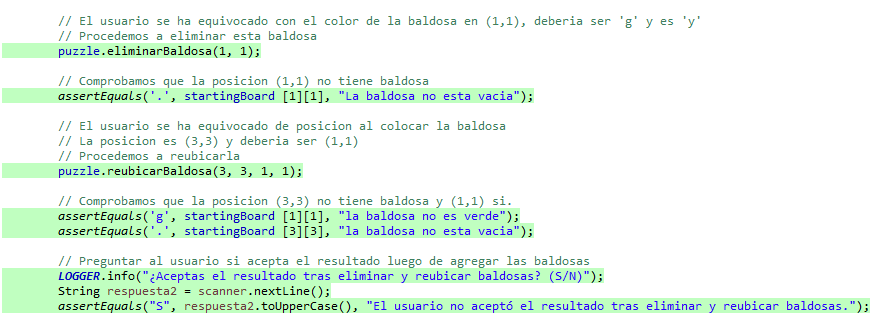


**Paso 3:**

* **Accion:** El usuario se ha equivocado al colocar el color de la baldosa en (1,1), ya que debería ser ‘g’ y no ‘y’. Debemos usar el método eliminarBaldosa(1, 1) para quitarla de esa posición, ya que no existe ningún método que permita cambiar el color de una baldosa.
* **Resultado esperado:** Con ayuda de un assertEquals, verificamos que la baldosa se haya eliminado correctamente y su posición sea ‘.’.



**Paso 4:**

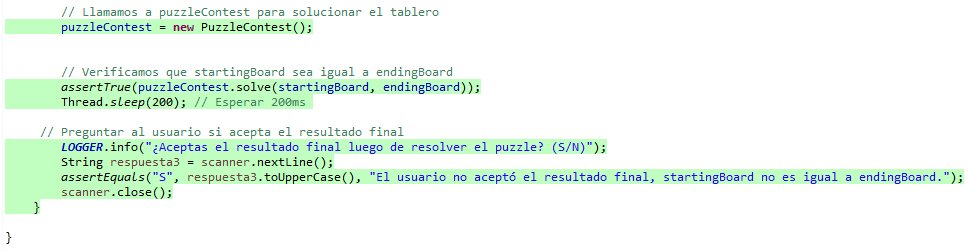
* **Accion:** Nuevamente, el usuario se ha equivocado y no colocó correctamente la baldosa en (3,3), ya que debería estar en (1,1). Afortunadamente, contamos con el método reubicarBaldosa, que nos permite cambiar la posición de las baldosas, por lo que procedemos a hacer el cambio. Nuevamente, le preguntamos al usuario si está de acuerdo con los resultados.
* **Resultado esperado:** Confirmamos mediante un assertEquals que la baldosa se halla movido correctamente verificando que el color de la posición (1,1) es verde y en (3,3) quedo vacío. 

**Paso 5:**

* **Accion:** Por último, el usuario desea verificar si el tablero tiene solución o si es imposible llegar de la configuración inicial al estado final. Para ello, llamamos a PuzzleContest, específicamente a su método solve, para que comience a resolverlo.

Al finalizar, le preguntamos por última vez al usuario si está de acuerdo con el resultado final.

* **Resultado esperado:** Para verificar, encapsulamos la acción en un assertTrue, ya que, en caso de que falle, lanzará un error. Esto es debido a que la función solve está diseñada para en caso de tener solución retorne un true y en caso contrario un false a la vez que muestra en pantalla el tablero y las distintas posiciones que visite para resolver el puzzle.

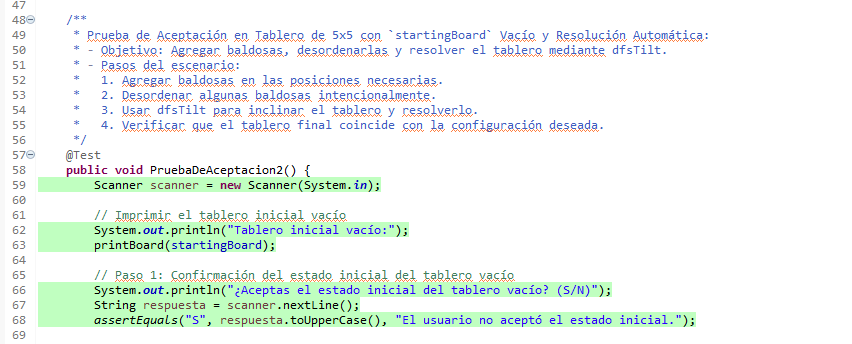


* **Conclusión:** Se espera que, al finalizar la prueba, el puzzle se resuelva habiendo realizado todos los cambios necesarios al tablero para alcanzar un estado donde sea posible solucionar el problema. En este caso específico vemos que en efecto es así por tanto la prueba de aceptación pasa exitosamente.

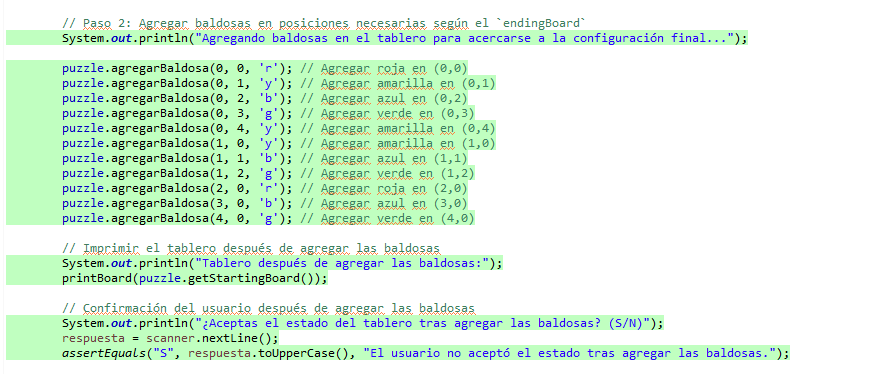
1. **PuzzleBTest(copia):** Esta prueba de aceptación se enfoca en evaluar la funcionalidad de dfsTilt, una técnica de búsqueda en profundidad que aplica inclinaciones inteligentes para resolver el tablero. La prueba inicia con un tablero de 5x5 vacío (startingBoard) y utiliza un endingBoard predefinido para establecer el estado final deseado. Los pasos del escenario incluyen agregar baldosas, desordenarlas intencionalmente y luego usar dfsTilt para resolver el tablero automáticamente.

**Procedimiento y Resultados Esperados:**

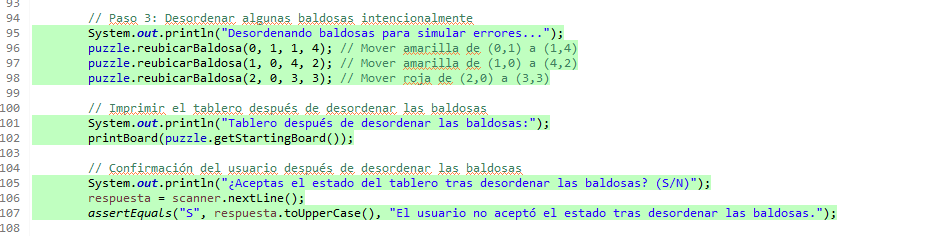
* **Paso 1:** Confirmación del Estado Inicial
* **Acción:** Se imprime el tablero inicial vacío y se solicita al usuario que confirme su estado.
* **Resultado Esperado:** El usuario acepta el estado inicial vacío del tablero. Esto asegura que la configuración inicial es correcta.



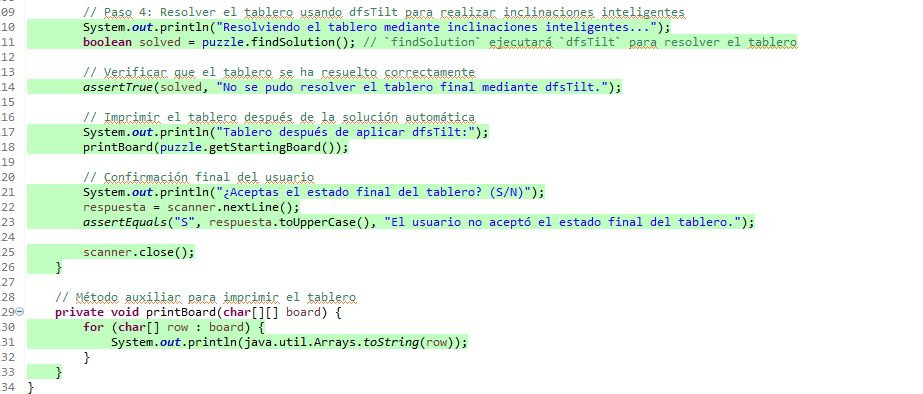
* **Paso 2:** Agregar Baldosas en las Posiciones Necesarias
* **Acción:** Se agregan baldosas en posiciones específicas para acercar el tablero a la configuración final deseada (endingBoard). Se agregan baldosas de diferentes colores (rojas, verdes, azules, amarillas) en las posiciones definidas en el endingBoard.
* **Resultado Esperado:** El tablero refleja correctamente las posiciones de las baldosas añadidas, y el usuario acepta el estado actual del tablero. Esto confirma que las baldosas se colocaron correctamente y que el tablero está en el estado previsto antes de realizar cualquier modificación.



* **Paso 3:** Desordenar Baldosas para Simular Errores
* **Acción:** Se desordenan algunas baldosas intencionalmente, moviéndolas a posiciones diferentes para simular un estado de desorden. Por ejemplo, se mueve una baldosa amarilla de (0,1) a (1,4), otra de (1,0) a (4,2), y una baldosa roja de (2,0) a (3,3).
* **Resultado Esperado:** El estado del tablero refleja los cambios realizados en las posiciones de las baldosas desordenadas, y el usuario acepta el estado del tablero tras el desorden. Este paso verifica que el sistema permite la reubicación de baldosas de acuerdo con las acciones específicas del usuario.



* **Paso 4:** Resolver el Tablero Usando dfsTilt
* **Acción:** Se llama al método findSolution, que utiliza dfsTilt para aplicar inclinaciones inteligentes en el tablero y llevarlo a la configuración final.
* **Resultado Esperado:** El método findSolution retorna true, indicando que el tablero se resolvió correctamente y que su configuración final coincide con la deseada (endingBoard). Este resultado asegura que la lógica de dfsTilt funciona adecuadamente, permitiendo resolver el tablero desde un estado desordenado.



**Confirmación Final:** Al final del proceso, se muestra el tablero final al usuario, quien debe confirmar que el tablero está en el estado correcto. Esto proporciona una verificación adicional de que la prueba fue exitosa y que el sistema completó todas las acciones según lo esperado.

**Conclusión:** La prueba de aceptación pasa exitosamente, lo que valida que la función dfsTilt es capaz de resolver automáticamente un tablero de 5x5 a partir de un estado vacío inicial y una configuración final predefinida.