Marc Belmonte Alcázar
Xavier Prats Fernández
Dilluns 10:30
2048 (https://github.com/NIU1633672/2048TQS)

Funcionalitat: Inicialitzar el tauler
Localització: <Arxiu, classe i mètode desenvolupat>

- Arxiu: src/model/board.py
- Classe: Board
- Mètode desenvolupat: _init_

Test:

- Arxiu: tests/test_board.py
- Mètode de test associat a la funcionalitat:

- Caixa negra / caixa blanca
- Tècniques utilitzades:
  - Statement coverage:

Name	Stmts	Miss	Cover
src\controller\game_controller.py	37	20	46%
src\model\board.py	146	1	99%
src\model\cell.py	18	1	94%
src\model\game.py	36	0	100%
src\view\game_view.py	15	9	40%
TOTAL	252	31	88%

  - Valors límit i frontera (Límits: 3, 5. Frontera: 4) / Particions equivalents (valor vàlid = 4, no vàlid != 4)

```python
def test_board_size_values():
    """
    Verifica el comportamiento del tamaño del tablero en valores frontera y límites.
    """

    # Valor frontera o único valor permitido (4)

    board_max = Board(4)  # Frontera superior válida
    assert len(board_max.grid) == 4

    # Valores límite inferiores (no válidos)
    with pytest.raises(ValueError):
        Board(3)  # Menor que el mínimo válido

    # Valores límite superiores (no válidos)
    with pytest.raises(ValueError):
        Board(5)  # Mayor que el máximo válido
```

Funcionalitat: Reset

Localització: <Arxiu, classe i mètode desenvolupat>

- Arxiu: src/model/board.py
- Classe: Board
- Mètode desenvolupat: reset

Test:

- Arxiu: tests/test_board.py
- Mètode de test associat a la funcionalitat:

- Caixa negra / caixa blanca
- Tècniques utilitzades:
  - Statement coverage:

Name	Stmts	Miss	Cover
src\controller\game_controller.py	37	20	46%
src\model\board.py	146	1	99%
src\model\cell.py	18	1	94%
src\model\game.py	36	0	100%
src\view\game_view.py	15	9	40%
TOTAL	252	31	88%

  - Valors límit i frontera / Particions equivalents (tauler ple, amb algunes posicions ocupades, buit)

```python
# Particions equivalents i valors límit reset

def test_reset_empty_board():
    board = Board(4)  # Tablero vacío
    board.reset()
    assert board.is_empty()   # Debe seguir vacío

def test_reset_partial_filled_board():
    board = Board(4)
    board.grid[0][0].set_value(2)  # Una celda ocupada
    board.reset()
    assert board.is_empty()  # Debe estar vacío después de reset

def test_reset_fully_filled_board():
    board = Board(4)
    for row in board.grid:
        for cell in row:
            cell.set_value(2)  # Llenamos todas las celdas
    board.reset()
    assert board.is_empty()  # Debe estar vacío después de reset
```

Funcionalitat: Afegir cel·la aleatoria

Localització: <Arxiu, classe i mètode desenvolupat>

- Arxiu: src/model/board.py
- Classe: Board
- Mètode desenvolupat: add_random_tile

Test:

- Arxiu: tests/test_board.py
- Mètode de test associat a la funcionalitat:

- Caixa negra / caixa blanca
- Tècniques utilitzades:
  - Statement coverage:

Name	Stmts	Miss	Cover
src\controller\game_controller.py	37	20	46%
src\model\board.py	146	1	99%
src\model\cell.py	18	1	94%
src\model\game.py	36	0	100%
src\view\game_view.py	15	9	40%
TOTAL	252	31	88%

  - Particions equivalents (valor vàlid / no vàlid)

```python
# Partició equivalent add_random_tile (vàlid / no vàlid)

def test_add_random_tile():
    controller = GameController(4)
    initial_board = controller.game.board  # Cambiar aquí
    controller.add_random_tile()
    # Verifica que una celda vacía ahora tiene un valor válido
    has_valid_value = any(cell.get_value() in (2, 4) for row in initial_board.grid for cell in row if cell.get_value() is not None)
    assert has_valid_value

def test_add_random_tile_invalid_value():
    board = Board(4)  # Crea un tablero de 4x4
    board.reset()  # Asegúrate de que el tablero esté vacío al inicio

    # Asegúrate de que no hay celdas llenas antes de agregar una
    assert all(cell.is_empty() for row in board.grid for cell in row), "El tablero debe estar vacío"

    # Llama a add_random_tile y verifica que se añade un valor válido (2 o 4)
    board.add_random_tile()

    # Comprueba que no hay valores inválidos (3) en el tablero
    for row in board.grid:
        for cell in row:
            assert cell.get_value() != 3, "No debe haber un valor inválido en el tablero"
```

  - Valor límit superior i inferior (1,3,5)

```python
def test_add_random_tile_invalid_values():
    board = Board(4)  # Crea un tablero de 4x4
    board.reset()  # Asegúrate de que el tablero esté vacío al inicio

    # Asegúrate de que no hay celdas llenas antes de agregar una
    assert all(cell.is_empty() for row in board.grid for cell in row), "El tablero debe estar vacío"

    # Llama a add_random_tile y verifica que se añade un valor válido (2 o 4)
    board.add_random_tile()

    # Comprueba que no hay valores inválidos (1, 3 y 5) en el tablero
    for row in board.grid:
        for cell in row:
            assert cell.get_value() != 1, "No debe haber un valor inválido 1 en el tablero"
            assert cell.get_value() != 3, "No debe haber un valor inválido 3 en el tablero"
            assert cell.get_value() != 5, "No debe haber un valor inválido 5 en el tablero"
```

  - Valor frontera (2 o 4)

```python
def test_add_random_tile():
    controller = GameController(4)
    initial_board = controller.game.board  # Cambiar aquí
    controller.add_random_tile()
    # Verifica que una celda vacía ahora tiene un valor válido
    has_valid_value = any(cell.get_value() in (2, 4) for row in initial_board.grid for cell in row if cell.get_value() is not None)
    assert has_valid_value
```

Funcionalitat: Moure a l'esquerra

Localització: <Arxiu, classe i mètode desenvolupat>

- Arxiu: src/model/board.py
- Classe: Board
- Mètode desenvolupat: move_left

Test:

- Arxiu: tests/test_board.py
- Mètode de test associat a la funcionalitat:

- Caixa negra / caixa blanca
- Tècniques utilitzades:
    - Statement coverage:

Name	Stmts	Miss	Cover
src\controller\game_controller.py	37	20	46%
src\model\board.py	146	1	99%
src\model\cell.py	18	1	94%
src\model\game.py	36	0	100%
src\view\game_view.py	15	9	40%
TOTAL	252	31	88%

    - Valor límit / frontera:

```python
def test_move_left():
    controller = GameController(4)
    controller.game.board.grid[0][0].set_value(2)
    controller.game.board.grid[0][1].set_value(2)

    # Asegúrate de que el movimiento a la izquierda funciona como se espera
    controller.play_turn('left')
    # Añade aserciones para verificar el estado del tablero después del movimiento
    assert controller.game.board.grid[0][0].get_value() == 4

def test_move_left():
    controller = GameController(4)
    controller.game.board.grid[0][0].set_value(2048)
    controller.game.board.grid[0][1].set_value(2048)

    with pytest.raises(ValueError):
        controller.play_turn('left')
```

```python
def test_move_left_with_1():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(1)  # Debería fallar al establecer el valor

def test_move_left_with_3():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(3)  # Debería fallar al establecer el valor

def test_move_left_with_2047():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(2047)  # Debería fallar al establecer el valor

def test_move_left_with_2049():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(2049)  # Debería fallar al establecer el valor
```

- Particions equivalents: combinació possible / no possible. També són proves de Decision coverage

```python
def move_left(self):
    """
    Mueve todas las fichas hacia la izquierda, combinando las que sean iguales.
    Devuelve True si hubo algún cambio, False en caso contrario.
    """
    self.last_move_score = 0  # Inicializamos el puntaje del último movimiento
    moved = False
    for row in self.grid:
        # Extraemos los valores no vacíos
        values = [cell.value for cell in row if not cell.is_empty()]

        # Combinamos valores iguales
        new_values = []
        skip = False
        for i in range(len(values)):
            if skip:
                skip = False
                continue
            if i < len(values) - 1 and values[i] == values[i + 1]:
                new_values.append(values[i] * 2)
                combined_value = values[i] * 2
                self.last_move_score += combined_value
                skip = True  # Saltar la siguiente celda
            else:
                new_values.append(values[i])

        # Rellenamos con ceros hasta completar el tamaño original de la fila
        new_values.extend([0] * (self.size - len(new_values)))

        # Verificamos si hubo cambios
        if new_values != [cell.value for cell in row]:
            moved = True

        # Actualizamos la fila del tablero
        for i in range(self.size):
            row[i].set_value(new_values[i])

    if moved: #Poscondicion: ha habido movimiento de una ficha
        assert any(cell.value != 0 for row in self.grid for cell in row)
    return moved
```

```python
def test_move_left_combination():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(2)
    board.move_left()
    assert board.grid[0][0].get_value() == 4  # Se combinan las celdas

def test_move_left_no_combination():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(4)
    board.move_left()
    assert board.grid[0][0].get_value() == 2  # No se combinan

def test_move_left_empty_cells():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(0)  # Celda vacía
    board.move_left()
    assert board.grid[0][0].get_value() == 2  # Debe permanecer igual
    assert board.grid[0][1].get_value() == 0  # Debe permanecer vacío

def test_move_left_multiple_combinations():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(2)
    board.grid[0][2].set_value(4)
    board.grid[0][3].set_value(4)  # Se pueden combinar
    assert board.move_left() == True  # Se espera que haya movimiento
    assert board.grid[0][0].get_value() == 4  # Combinación 2 + 2
    assert board.grid[0][1].get_value() == 8  # Combinación 4 + 4
    assert board.grid[0][2].get_value() == 0  # Relleno con 0

def test_move_left_no_change():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(4)  # No se pueden combinar
    assert board.move_left() == False  # No hubo movimiento
    assert board.grid[0][0].get_value() == 2  # Sin cambios
    assert board.grid[0][1].get_value() == 4  # Sin cambios
```

`if i < len(values) - 1 and values[i] == values[i + 1]:`

- Ruta 1: Si son iguals, es combinen
- Ruta 2: Si no son iguals, s'afegeixen a new_values

`if new_values != [cell.value for cell in row]:`

- Ruta 1: Si hi ha canvis, s'estableix moved = true
- Ruta 2: Si no hi ha canvis, moved = false

Amb això cobrim:

- combinació de cel·les si o no
- canvis en l'estat del tauler (canvis o sense canvis)
- cel·les buides / plenes

Funcionalitat: Moure a la dreta

Localització: <Arxiu, classe i mètode desenvolupat>

- Arxiu: src/model/board.py
- Classe: Board
- Mètode desenvolupat: move_right

Test:

- Arxiu: tests/test_board.py
- Mètode de test associat a la funcionalitat: test_move_right

- Caixa negra / caixa blanca
- Tècniques utilitzades:
    - Statement coverage:

Name	Stmts	Miss	Cover
src\controller\game_controller.py	37	20	46%
src\model\board.py	146	1	99%
src\model\cell.py	18	1	94%
src\model\game.py	36	0	100%
src\view\game_view.py	15	9	40%
TOTAL	252	31	88%

    - Particions equivalents (cel·les es poden combinar, no es poden combinar, cel·les buides, cel·les amb un valor i una cel·la buida que es poden moure).

```python
def test_move_right_combination():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(2)  # Deben combinarse
    assert board.move_right() == True  # Se espera que haya movimiento
    assert board.grid[0][3].get_value() == 4  # La combinación debe estar a la derecha

def test_move_right_no_combination():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(4)  # No se pueden combinar
    assert board.grid[0][0].get_value() == 2  # Sin cambios
    assert board.grid[0][1].get_value() == 4  # Sin cambios

def test_move_right_with_empty_cells():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(0)  # Celda vacía
    assert board.move_right() == True  # Debe mover 2 a la derecha
    assert board.grid[0][3].get_value() == 2  # Debe moverse a la derecha


def test_move_right_multiple_combinations():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(2)
    board.grid[0][2].set_value(4)
    board.grid[0][3].set_value(4)  # Deben combinarse
    assert board.move_right() == True  # Se espera que haya movimiento
    assert board.grid[0][2].get_value() == 4
    assert board.grid[0][3].get_value() == 8
```

```python
def test_move_right_no_change():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(4)
    board.grid[0][2].set_value(2)
    board.grid[0][3].set_value(4)  # Sin cambios
    assert board.move_right() == False  # No hubo movimiento
    assert board.grid[0][0].get_value() == 2  # Sin cambios
    assert board.grid[0][1].get_value() == 4  # Sin cambios
```

Valors límit i frontera:

```python
def test_move_right():
    controller = GameController(4)
    controller.game.board.grid[0][2].set_value(2)
    controller.game.board.grid[0][3].set_value(2)

    # Asegúrate de que el movimiento a la derecha funciona como se espera
    controller.play_turn('right')
    # Añade aserciones para verificar el estado del tablero después del movimiento
    assert controller.game.board.grid[0][3].get_value() == 4

def test_move_right_with_2048():
    controller = GameController(4)
    controller.game.board.grid[0][2].set_value(2048)
    controller.game.board.grid[0][3].set_value(2048)

    with pytest.raises(ValueError):
        controller.play_turn('right')

def test_move_right_with_1():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(1)  # Debería fallar al establecer el valor
```

```python
def test_move_right_with_3():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(3)  # Debería fallar al establecer el valor

def test_move_right_with_2047():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(2047)  # Debería fallar al establecer el valor

def test_move_right_with_2049():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(2049)  # Debería fallar al establecer el valor
```

- Decision coverage:

```python
def move_right(self):
    """
    Mueve todas las fichas hacia la derecha, combinando las que sean iguales.
    Devuelve True si hubo algún cambio, False en caso contrario.
    """
    self.last_move_score = 0
    moved = False
    for row in self.grid:
        # Extraemos los valores no vacíos
        values = [cell.value for cell in row if not cell.is_empty()]

        # Combinamos valores iguales
        new_values = []
        skip = False
        for i in range(len(values) - 1, -1, -1):  # Iteramos de derecha a izquierda
            if skip:
                skip = False
                continue
            if i > 0 and values[i] == values[i - 1]:  # Comparar con la celda a la izquierda
                new_values.append(values[i] * 2)
                combined_value = values[i] * 2
                self.last_move_score += combined_value
                skip = True  # Saltar la siguiente celda
            else:
                new_values.append(values[i])

        # Rellenamos con ceros hasta completar el tamaño original de la fila
        new_values.extend([0] * (self.size - len(new_values)))

        # Actualizamos la fila del tablero en el orden correcto (de derecha a izquierda)
        new_values.reverse()  # Invertir para colocar los valores de nuevo en la fila

        # Verificamos si hubo cambios
        if new_values != [cell.value for cell in row]:
            moved = True

        for i in range(self.size):
            row[i].set_value(new_values[i])

    if moved: #Poscondicion: ha habido movimiento de una ficha
        assert any(cell.value != 0 for row in self.grid for cell in row)
    return moved
```

```python
def test_move_right_combination():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(2)  # Deben combinarse
    assert board.move_right() == True  # Se espera que haya movimiento
    assert board.grid[0][3].get_value() == 4  # La combinación debe estar a la derecha

def test_move_right_no_combination():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(4)  # No se pueden combinar
    assert board.grid[0][0].get_value() == 2  # Sin cambios
    assert board.grid[0][1].get_value() == 4  # Sin cambios

def test_move_right_with_empty_cells():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(0)  # Celda vacía
    assert board.move_right() == True  # Debe mover 2 a la derecha
    assert board.grid[0][3].get_value() == 2  # Debe moverse a la derecha


def test_move_right_multiple_combinations():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(2)
    board.grid[0][2].set_value(4)
    board.grid[0][3].set_value(4)  # Deben combinarse
    assert board.move_right() == True  # Se espera que haya movimiento
    assert board.grid[0][2].get_value() == 4
    assert board.grid[0][3].get_value() == 8

def test_move_right_no_change():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(4)
    board.grid[0][2].set_value(2)
    board.grid[0][3].set_value(4)  # Sin cambios
    assert board.move_right() == False  # No hubo movimiento
    assert board.grid[0][0].get_value() == 2  # Sin cambios
    assert board.grid[0][1].get_value() == 4  # Sin cambios
```

- test_move_right_combination: Cubre el caso donde se combinan valores (verdadero para la condición de combinación).
- test_move_right_no_combination: Cubre el caso donde no hay combinación (falso para la condición de combinación).

- test_move_right_with_empty_cells(): Cubre el movimiento de una celda a una celda vacía
- test_move_right_multiple_combinations: Cubre múltiples combinaciones, asegurando que se manejen correctamente múltiples decisiones en una sola fila.
- test_move_right_no_change: Cubre el caso donde no hay ningún cambio, lo que es crucial para la decisión de ver si se movió algo.

Funcionalitat: Moure cap amunt

Localització: <Arxiu, classe i mètode desenvolupat>

- Arxiu: src/model/board.py
- Classe: Board
- Mètode desenvolupat: move_up

Test:

- Arxiu: tests/test_board.py
- Mètode de test associat a la funcionalitat: test_move_up

- Caixa negra / caixa blanca
- Tècniques utilitzades:
    - Statement coverage:

Name	Stmts	Miss	Cover
src\controller\game_controller.py	37	20	46%
src\model\board.py	146	1	99%
src\model\cell.py	18	1	94%
src\model\game.py	36	0	100%
src\view\game_view.py	15	9	40%
TOTAL	252	31	88%

    - Particions equivalents (combinació possible o no)

```python
def test_move_up_combination():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[1][0].set_value(2)
    assert board.move_up()  # Debería devolver True
    assert board.grid[0][0].get_value() == 4  # Las celdas deben combinarse

def test_move_up_no_combination():
    board = Board()
    board.grid[0][0].set_value(2)
    board.grid[1][0].set_value(4)
    assert not board.move_up()  # Debería devolver False, sin cambios
    assert board.grid[0][0].get_value() == 2  # Sin cambios
```

- Valors límit i frontera

```python
def test_move_up():
    controller = GameController(4)
    controller.game.board.grid[2][0].set_value(2)
    controller.game.board.grid[3][0].set_value(2)

    # movimiento hacia arriba funciona como se espera
    controller.play_turn('up')
    # Añade aserciones para verificar el estado del tablero después del movimiento
    assert controller.game.board.grid[0][0].get_value() == 4

def test_move_up_with_2048():
    controller = GameController(4)
    controller.game.board.grid[2][0].set_value(2048)
    controller.game.board.grid[3][0].set_value(2048)

    with pytest.raises(ValueError):
        controller.play_turn('up')

def test_move_up_with_1():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(1)  # Debería fallar al establecer el valor

def test_move_up_with_3():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(3)  # Debería fallar al establecer el valor
```

```python
def test_move_up_with_2047():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(2047)  # Debería fallar al establecer el valor

def test_move_up_with_2049():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(2049)  # Debería fallar al establecer el valor
```

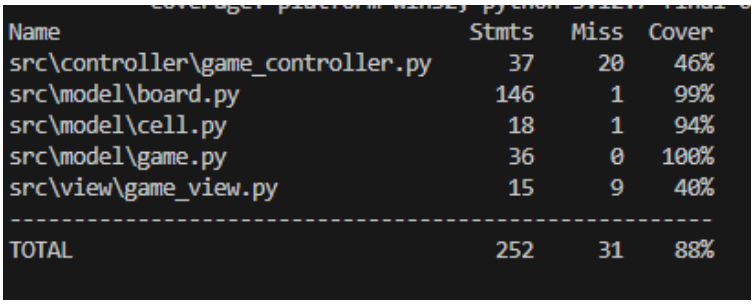
Funcionalitat: Moure cap avall

Localització: <Arxiu, classe i mètode desenvolupat>

- Arxiu: src/model/board.py
- Classe: Board
- Mètode desenvolupat: move_down

Test:

- Arxiu: tests/test_board.py
- Mètode de test associat a la funcionalitat: Caixa negra / caixa blanca
- Tècniques utilitzades:
    - Statement coverage:



Name	Stmts	Miss	Cover
src\controller\game_controller.py	37	20	46%
src\model\board.py	146	1	99%
src\model\cell.py	18	1	94%
src\model\game.py	36	0	100%
src\view\game_view.py	15	9	40%
TOTAL	252	31	88%

    - Valors límit i frontera:

```python
def test_move_down():
    controller = GameController(4)
    controller.game.board.grid[2][0].set_value(2)
    controller.game.board.grid[3][0].set_value(2)

    # Asegúrate de que el movimiento hacia abajo funciona como se espera
    controller.play_turn('down')
    # Añade aserciones para verificar el estado del tablero después del movimiento
    assert controller.game.board.grid[3][0].get_value() == 4

def test_move_down_with_2048():
    controller = GameController(4)
    controller.game.board.grid[2][0].set_value(2048)
    controller.game.board.grid[3][0].set_value(2048)

    with pytest.raises(ValueError):
        controller.play_turn('down')

def test_move_down_with_1():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(1)  # Debería fallar al establecer el valor
```

```python
def test_move_down_with_3():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(3)  # Debería fallar al establecer el valor

def test_move_down_with_2047():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(2047)  # Debería fallar al establecer el valor

def test_move_down_with_2049():
    controller = GameController(4)
    with pytest.raises(ValueError):
        controller.game.board.grid[0][0].set_value(2049)  # Debería fallar al establecer el valor
```

- Particions equivalents (moviment possible / no possible):

```python
def test_move_down_equivalence_partitions():
    controller = GameController(4)

    # Caso válido: movimientos que deben combinarse
    controller.game.board.grid[2][0].set_value(2)
    controller.game.board.grid[3][0].set_value(2)

    # Realizar movimiento hacia abajo
    move_successful = controller.play_turn('down')

    # Comprobar que el movimiento fue exitoso
    assert move_successful
    assert controller.game.board.grid[3][0].get_value() == 4  # Combinación debe dar 4
    assert controller.game.board.grid[2][0].get_value() == 0  # La celda debe estar vacía

    # Caso no válido: intentar mover un 2 y un 4 (no se deben combinar)
    controller.game.board.grid[2][0].set_value(2)  # Establecer 2 en la celda
    controller.game.board.grid[3][0].set_value(4)  # Establecer 4 en la celda inferior

    # Intentar mover hacia abajo, se espera que el movimiento no sea exitoso
    move_successful = controller.play_turn('down')

    # Verificar que no hubo movimiento
    assert not move_successful
    assert controller.game.board.grid[3][0].get_value() == 4  # La celda inferior debe seguir siendo 4
    assert controller.game.board.grid[2][0].get_value() == 2  # La celda superior debe seguir siendo 2
```

Funcionalitat: Tauler ple

Localització: <Arxiu, classe i mètode desenvolupat>

- Arxiu: src/model/board.py
- Classe: Board
- Mètode desenvolupat: is_full

Test:

- Arxiu: tests/test_board.py
- Mètode de test associat a la funcionalitat:

- Caixa negra / caixa blanca
- Tècniques utilitzades:
  - Statement coverage:

Name	Stmts	Miss	Cover
src\controller\game_controller.py	37	20	46%
src\model\board.py	146	1	99%
src\model\cell.py	18	1	94%
src\model\game.py	36	0	100%
src\view\game_view.py	15	9	40%
TOTAL	252	31	88%

  - Decision coverage i condition coverage:

    És decision coverage perquè verifiquem que l'if sigui true i false així com la condició sigui true o false

```python
def is_full(self):
    """Verifica si todas las celdas del tablero están llenas."""

    # Precondición: self.grid debe ser una matriz válida de celdas
    assert isinstance(self.grid, list) and all(isinstance(row, list) for row in self.grid)

    for row in self.grid:
        for cell in row:
            if cell.is_empty():
                return False
    return True
```

```python
def test_is_full_with_empty_cells():
    board = Board(size=4)
    board.grid[0][0].set_value(2)  # Una celda con valor, las demás vacías
    assert not board.is_full()  # El tablero no está lleno

def test_is_full_with_all_cells_filled():
    board = Board(size=4)
    for row in board.grid:
        for cell in row:
            cell.set_value(2)  # Llenamos todas las celdas
    assert board.is_full()  # El tablero está lleno
```

Funcionalitat: Hi ha moviments

Localització: <Arxiu, classe i mètode desenvolupat>

- Arxiu: src/model/board.py
- Classe: Board
- Mètode desenvolupat: has_moves

Test:

- Arxiu: tests/test_board.py
- Mètode de test associat a la funcionalitat: has_moves

- Caixa negra / caixa blanca
- Tècniques utilitzades:
  - Statement coverage:

```
Name                                Stmts   Miss  Cover
-------------------------------------------------------
src\controller\game_controller.py      37     20    46%
src\model\board.py                    146      1    99%
src\model\cell.py                      18      1    94%
src\model\game.py                      36      0   100%
src\view\game_view.py                  15      9    40%
-------------------------------------------------------
TOTAL                                 252     31    88%
```

  - Particions equivalents i valors limit / frontera (hi ha moviments o no hi ha moviments)

```python
def test_has_no_moves_possible_full_no_combinations():
    board = Board(size=4)
    values = [
        [2, 4, 8, 16],
        [32, 64, 128, 256],
        [512, 1024, 2048, 2],
        [2, 4, 8, 16],
    ]
    for i in range(4):
        for j in range(4):
            board.grid[i][j].set_value(values[i][j])
    assert not board.has_moves()  # El tablero lleno y sin combinaciones posibles

def test_has_moves_possible_with_combinations():
    board = Board(size=4)
    values = [
        [2, 2, 4, 8],
        [16, 32, 64, 128],
        [256, 512, 1024, 2048],
        [2, 4, 8, 16],
    ]
    for i in range(4):
        for j in range(4):
            board.grid[i][j].set_value(values[i][j])
    assert board.has_moves()  # Hay combinaciones posibles (dos 2 en la primera fila)
```

- Loop testing

```python
def has_moves(self):
    """
    Verifica si hay movimientos posibles en el tablero:
    - Celdas vacías.
    - Combinaciones posibles entre celdas adyacentes.
    """

    # Precondición: self.grid debe ser una matriz válida de celdas
    assert isinstance(self.grid, list) and all(isinstance(row, list) for row in self.grid)


    # Verifica si hay celdas vacías
    if not self.is_full():
        # Postcondición: Si hay celdas vacías, se puede mover
        assert any(cell.is_empty() for row in self.grid for cell in row)
        return True

    # Verifica si hay combinaciones posibles
    # Poscondiciones: se encuentra combinacion posible (true)
    for i in range(self.size):
        for j in range(self.size):
            current_value = self.grid[i][j].value

            # Verifica combinaciones horizontales
            if j + 1 < self.size and self.grid[i][j + 1].value == current_value:
                return True

            # Verifica combinaciones verticales
            if i + 1 < self.size and self.grid[i + 1][j].value == current_value:
                return True


    # Postcondición: Si no hay celdas vacías ni combinaciones posibles (si devuelve false no puede haber celdas vacías)
    assert not any(cell.is_empty() for row in self.grid for cell in row)
    return False
```

0 iteraciones:

```python
def test_has_no_moves_possible_full_no_combinations():
    board = Board(size=4)
    values = [
        [2, 4, 8, 16],
        [32, 64, 128, 256],
        [512, 1024, 2048, 2],
        [2, 4, 8, 16],
    ]
    for i in range(4):
        for j in range(4):
            board.grid[i][j].set_value(values[i][j])
    assert not board.has_moves()  # El tablero lleno y sin combinaciones posibles
```

```python
def test_has_moves_one_iteration():
    board = Board(size=4)
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(2)  # Un movimiento posible
    game = Game(size=4, board=board)
    assert board.has_moves()  # Iteraciones: 1

def test_has_moves_multiple_iterations():
    board = Board(size=4)
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(2)  # Un movimiento posible
    board.grid[1][0].set_value(4)
    board.grid[1][1].set_value(4)  # Otro movimiento posible
    game = Game(size=4, board=board)
    assert board.has_moves()  # Iteraciones: 3 filas x 3 comparaciones
```

Funcionalitat: Init

Localització:

- Arxiu: src/model/cell.py
- Classe: Cell
- Mètode desenvolupat: Init

Test:

- Arxiu: tests/test_cell.py
- Mètode de test associat a la funcionalitat: Caixa negra / caixa blanca
- Tècniques utilitzades: Statement coverage, Particions equivalents, Valors limit i frontera

Statement coverage:

```
Name                                Stmts   Miss  Cover
-------------------------------------------------------
src\controller\game_controller.py      37     20    46%
src\model\board.py                    146      1    99%
src\model\cell.py                      18      1    94%
src\model\game.py                      36      0   100%
src\view\game_view.py                  15      9    40%
-------------------------------------------------------
TOTAL                                 252     31    88%
```

- Valors límit i frontera

```python
def test_cell_boundary_values():
    # Valor inicial (límite inferior)
    cell = Cell()
    assert cell.value == 0  # Límite inferior esperado
    assert cell.is_empty() == True  # Una celda recién creada debe estar vacía

    # Modificar a un valor válido (límite interior)
    cell.value = 2  # Un valor típico del juego 2048
    assert cell.is_empty() == False  # Ya no está vacía

    # Modificar de vuelta a 0 (límite inferior)
    cell.value = 0
    assert cell.is_empty() == True  # Vuelve a estar vacía

    # Caso borde con un valor límite negativo (aunque en este caso, fuera de los valores válidos del juego)
    cell.value = -1
    assert cell.is_empty() == False  # Una celda con valor negativo no debería considerarse vacía
```

- Particions equivalents:

```python
def test_cell_equivalence_partitions():
    # Partición: celda vacía (valor 0)
    cell_empty = Cell()
    assert cell_empty.is_empty() == True  # Una celda recién creada debe estar vacía

    # Partición: celda con valores válidos del juego (2, 4, 8, ...)
    cell_valid = Cell()
    cell_valid.value = 2
    assert cell_valid.is_empty() == False  # Una celda con valor 2 no está vacía

    cell_valid.value = 4
    assert cell_valid.is_empty() == False  # Una celda con valor 4 no está vacía

    # Partición: celda con valores inválidos (negativos o no usados en el juego)
    cell_invalid = Cell()
    cell_invalid.value = -1
    assert cell_invalid.is_empty() == False  # Una celda con valor negativo no debe considerarse vacía

    cell_invalid.value = 3  # Un valor no estándar en el juego
    assert cell_invalid.is_empty() == False  # Tampoco debería considerarse vacía
```

Funcionalitat: is_empty

Localització:

- Arxiu: src/model/board.py
- Classe: Cell
- Mètode desenvolupat: is_empty

Test:

- Arxiu: tests/test_cell.py
- Mètode de test associat a la funcionalitat: Caixa negra / caixa blanca
- Tècniques utilitzades: Statement coverage, Particions equivalents, Valors limit i frontera

Statement coverage:

```
Name                                Stmts   Miss  Cover
-------------------------------------------------------
src\controller\game_controller.py      37     20    46%
src\model\board.py                    146      1    99%
src\model\cell.py                      18      1    94%
src\model\game.py                      36      0   100%
src\view\game_view.py                  15      9    40%
-------------------------------------------------------
TOTAL                                 252     31    88%
```

Valors limit i frontera:

```python
def test_is_empty_boundary_values():
    # Límite inferior: valor 0 (vacío)
    cell = Cell()
    cell.value = 0
    assert cell.is_empty() == True  # Una celda con valor 0 debe estar vacía

    # Límite cercano al inferior: valor 1 (no vacío)
    cell.value = 1
    assert cell.is_empty() == False  # Una celda con valor 1 no está vacía

    # Límite negativo: valor -1 (fuera del dominio normal del juego)
    cell.value = -1
    assert cell.is_empty() == False  # Una celda con valor negativo no está vacía

    # Límite superior (aunque no suele aplicarse en el juego): valor grande positivo
    cell.value = 2048
    assert cell.is_empty() == False  # Una celda con un valor válido del juego no está vacía
```

Particions equivalents:

```python
def test_is_empty_equivalence_partitions():
    # Partición: celdas vacías (valor 0)
    cell = Cell()
    cell.value = 0
    assert cell.is_empty() == True  # Celda vacía

    # Partición: celdas con valores válidos del juego (2, 4, 8, ...)
    cell.value = 2
    assert cell.is_empty() == False  # Celda con valor 2 no está vacía
    cell.value = 4
    assert cell.is_empty() == False  # Celda con valor 4 no está vacía

    # Partición: celdas con valores fuera del rango válido (negativos)
    cell.value = -1
    assert cell.is_empty() == False  # Celda con valor -1 no está vacía

    # Partición: celdas con valores no utilizados en el juego (no múltiplos de 2)
    cell.value = 3
    assert cell.is_empty() == False  # Celda con valor 3 no está vacía
```

Funcionalitat: set_value

Localització:

- Arxiu: src/model/board.py
- Classe: Cell
- Mètode desenvolupat: set_value

Test:

- Arxiu: tests/test_cell.py
- Mètode de test associat a la funcionalitat: Caixa negra
- Tècniques utilitzades: Statement coverage, Particions equivalents, Valors limit i frontera

Statement coverage:

```
Name                                Stmts   Miss  Cover
-------------------------------------------------------
src\controller\game_controller.py      37     20    46%
src\model\board.py                    146      1    99%
src\model\cell.py                      18      1    94%
src\model\game.py                      36      0   100%
src\view\game_view.py                  15      9    40%
-------------------------------------------------------
TOTAL                                 252     31    88%
```

Particions equivalents:

```python
# Tests de particiones equivalentes (Valores válidos / no válidos)

def test_set_value():
    """
    Verifica que los valores válidos se asignan correctamente a la celda.
    """
    cell = Cell()
    valid_values = [2, 4, 8]  # Valores válidos definidos por el juego
    for value in valid_values:
        cell.set_value(value)
        assert cell.value == value  # El valor de la celda debe coincidir con el asignado
        assert not cell.is_empty()  # La celda no debe estar vacía después de asignar un val

def test_set_invalid_value():
    """
    Verifica que los valores no válidos lancen una excepción al intentar asignarlos.
    """
    cell = Cell()
    invalid_values = [-1, 3, 5, 6, 7, 9, 10, 12]  # Valores no válidos
    for value in invalid_values:
        with pytest.raises(ValueError):
            cell.set_value(value)  # Debe lanzar una excepción si el valor no es válido
```

Valors límit i frontera:

```python
# Test de valores límite y frontera

def test_set_value_with_limits_and_boundaries():
    """
    Verifica el comportamiento del método set_value en valores frontera y límite.
    """
    cell = Cell()

    # Valores frontera (válidos)
    cell.set_value(2)  # Frontera inferior
    assert cell.value == 2

    cell.set_value(2048)  # Frontera superior
    assert cell.value == 2048

    cell.set_value(16) # Valor interior
    assert cell.value == 16

    invalid_values = [1, 3, 2047, 2049]  # Valores no válidos
    for value in invalid_values:
        with pytest.raises(ValueError):
            cell.set_value(value)  # Debe lanzar una excepción si el valor no es válido
```

Funcionalitat: reset

Localització:

- Arxiu: src/model/board.py
- Classe: Cell
- Mètode desenvolupat: reset

Test:

- Arxiu: tests/test_cell.py
- Mètode de test associat a la funcionalitat: Caixa negra
- Tècniques utilitzades: Statement coverage, Particions equivalents, Valors limit i frontera

Statement coverage:

```
Name                                Stmts   Miss  Cover
-------------------------------------------------------
src\controller\game_controller.py      37     20    46%
src\model\board.py                    146      1    99%
src\model\cell.py                      18      1    94%
src\model\game.py                      36      0   100%
src\view\game_view.py                  15      9    40%
-------------------------------------------------------
TOTAL                                 252     31    88%
```

Valors limit i frontera:

```python
def test_reset_boundary_values():
    # Caso límite inferior: valor inicial 0
    cell = Cell()
    cell.value = 0
    cell.reset()
    assert cell.value == 0  # La celda debe estar vacía después del reset

    # Caso límite cercano al inferior: valor inicial 1
    cell.value = 1
    cell.reset()
    assert cell.value == 0  # La celda debe estar vacía después del reset

    # Caso límite alto: valor inicial 2048 (valor máximo en el juego)
    cell.value = 2048
    cell.reset()
    assert cell.value == 0  # La celda debe estar vacía después del reset

    # Caso límite negativo: valor inicial -1 (fuera del dominio normal)
    cell.value = -1
    cell.reset()
    assert cell.value == 0  # La celda debe estar vacía después del reset
```

Particions equivalents:

```python
def test_reset_equivalence_partitions():
    # Partición: celdas vacías (valor 0)
    cell = Cell()
    cell.value = 0
    cell.reset()
    assert cell.is_empty() == True  # Debe mantenerse vacía

    # Partición: celdas con valores válidos del juego (2, 4, 8, ...)
    cell.value = 2
    cell.reset()
    assert cell.is_empty() == True  # Debe estar vacía después del reset
    cell.value = 4
    cell.reset()
    assert cell.is_empty() == True  # Debe estar vacía después del reset

    # Partición: celdas con valores fuera del rango válido (negativos o no múltiplos de 2)
    cell.value = -1
    cell.reset()
    assert cell.is_empty() == True  # Debe estar vacía después del reset
    cell.value = 3
    cell.reset()
    assert cell.is_empty() == True  # Debe estar vacía después del reset
```

Funcionalitat: init

Localització:

- Arxiu: src/model/game.py
- Classe: Game
- Mètode desenvolupat: init

Test:

- Arxiu: tests/test_game.py
- Mètode de test associat a la funcionalitat: Caixa negra
- Tècniques utilitzades: Statement coverage, Particions equivalents, Valors limit i frontera

Statement coverage:

Name	Stmts	Miss	Cover
src\controller\game_controller.py	37	20	46%
src\model\board.py	146	1	99%
src\model\cell.py	18	1	94%
src\model\game.py	36	0	100%
src\view\game_view.py	15	9	40%
TOTAL	252	31	88%

Valor limit i frontera:

```python
def test_init_boundary_values():
    # Caso límite: tamaño correcto (4)
    game = Game(4)
    assert game.board.size == 4  # Debe crear un tablero de tamaño 4
    assert isinstance(game.board, Board)  # Debe ser una instancia de Board
    assert game.score == 0  # Puntuación inicial debe ser 0

    # Caso límite incorrecto: tamaño incorrecto
    try:
        Game(3)  # Debe lanzar una excepción
        assert False, "Expected an AssertionError for size 3"
    except AssertionError:
        pass  # Se espera que falle, así que esto es correcto

    try:
        Game(5)  # Debe lanzar una excepción
        assert False, "Expected an AssertionError for size 5"
    except AssertionError:
        pass  # Se espera que falle, así que esto es correcto
```

Particions equivalents:

```python
def test_init_equivalence_partitions():
    # Partición: Tamaño correcto (4)
    game = Game(4)
    assert game.board.size == 4  # Debe crear un tablero de tamaño 4
    assert isinstance(game.board, Board)  # Debe ser una instancia de Board
    assert game.score == 0  # Puntuación inicial debe ser 0

    # Partición: Otro tablero (board no es None)
    custom_board = Board(4)
    game_with_custom_board = Game(4, custom_board)
    assert game_with_custom_board.board == custom_board  # Debe usar el otro tablero
    assert game_with_custom_board.score == 0  # Puntuación inicial debe seguir siendo

    # Partición: Tamaño incorrecto (3 o 5, que deben causar una excepción)
    for invalid_size in [3, 5]:
        try:
            Game(invalid_size)  # Debe lanzar una excepción
            assert False, f"Expected an AssertionError for size {invalid_size}"
        except AssertionError:
            pass  # Se espera que falle, así que esto es correcto
```

Funcionalitat: play_turn

Localització:

- Arxiu: src/model/game.py
- Classe: Game
- Mètode desenvolupat: play_turn

Test:

- Arxiu: tests/test_game.py
- Mètode de test associat a la funcionalitat: Caixa negra
- Tècniques utilitzades: Statement coverage, Particions equivalents, Valors limit i frontera

Statement coverage:

```
Name                                Stmts   Miss  Cover
-------------------------------------------------------
src\controller\game_controller.py      37     20    46%
src\model\board.py                    146      1    99%
src\model\cell.py                      18      1    94%
src\model\game.py                      36      0   100%
src\view\game_view.py                  15      9    40%
-------------------------------------------------------
TOTAL                                 252     31    88%
```

Particions equivalents

```python
# Particions equivalents, valors valids / no valids

def test_play_turn_valid_directions():
    """
    Prova que les direccions vàlides executen moviments correctament.
    """
    game = Game(4)
    for direction in ["left", "right", "up", "down"]:
        assert game.play_turn(direction) in [True, False]  # Retorna True si hi ha canvi

def test_play_turn_invalid_directions():
    """
    Prova que les direccions no vàlides llencen una excepció.
    """
    game = Game(4)
    invalid_directions = ["diagonal", "", 123, None]
    for direction in invalid_directions:
        with pytest.raises(ValueError):
            game.play_turn(direction)
```

Valors limit i frontera:

```python
def test_play_turn_boundary_values():
    """
    Verifica el comportamiento de play_turn en valores límite y frontera.
    """
    # Caso límite: tablero completamente lleno
    board_full = Board(size=4)
    values_full = [
        [2, 4, 2, 4],
        [4, 2, 4, 2],
        [2, 4, 2, 4],
        [4, 2, 4, 2],
    ]
    for i in range(4):
        for j in range(4):
            board_full.grid[i][j].set_value(values_full[i][j])
    game_full = Game(size=4, board=board_full)

    # Intentar mover en una dirección (debería retornar False porque no hay movimientos posibles)
    assert not game_full.play_turn("left")
    assert not game_full.play_turn("right")
    assert not game_full.play_turn("up")
    assert not game_full.play_turn("down")

    # Caso límite: un solo movimiento posible
    board_single_move = Board(size=4)
    values_single_move = [
        [2, 2, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
    ]
    for i in range(4):
        for j in range(4):
            board_single_move.grid[i][j].set_value(values_single_move[i][j])
    game_single_move = Game(size=4, board=board_single_move)
```

```python
game_single_move = Game(size=4, board=board_single_move)

# Intentar mover en una dirección (debería retornar True porque hay un movimiento válido)
assert game_single_move.play_turn("left")
assert game_single_move.score == 4  # La puntuación debe haber aumentado por el movimiento

# Verificar el estado del tablero después del movimiento
assert game_single_move.board.grid[0][0].value == 4  # Los dos 2 deben haberse combinado en 4
assert game_single_move.board.grid[0][1].is_empty()  # La segunda celda debe estar vacía

# Caso límite: tablero inicial vacío
board_empty = Board(size=4)
game_empty = Game(size=4, board=board_empty)

# Intentar mover en una dirección (debería retornar False porque no hay movimientos posibles)
assert not game_empty.play_turn("left")
assert not game_empty.play_turn("right")
assert not game_empty.play_turn("up")
assert not game_empty.play_turn("down")
```

Pairwise testing

```python
@pytest.mark.parametrize("initial_grid, expected_grid, direction", [
    ([[0, 2, 2, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]],
     [[4, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]], "left"),

    ([[0, 2, 2, 0], [0, 0, 0, 0], [0, 0, 0, 0], [2, 0, 0, 0]],
     [[4, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [2, 0, 0, 0]], "left"),

    ([[0, 0, 2, 2], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]],
     [[0, 0, 0, 4], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]], "right"),

    ([[2, 0, 0, 0], [2, 0, 0, 0], [4, 0, 0, 0], [0, 0, 0, 0]],
     [[4, 0, 0, 0], [4, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]], "up"),

    ([[0, 0, 0, 0], [0, 2, 0, 0], [0, 2, 0, 0], [0, 0, 0, 0]],
     [[0, 4, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]], "up"),

    ([[2, 0, 0, 0], [2, 0, 0, 0], [0, 0, 0, 0], [4, 0, 0, 0]],
     [[0, 0, 0, 0], [0, 0, 0, 0], [4, 0, 0, 0], [4, 0, 0, 0]], "down"),

     ([[0, 0, 0, 0], [2, 0, 0, 0], [2, 0, 0, 0], [4, 0, 0, 0]],
     [[0, 0, 0, 0], [0, 0, 0, 0], [4, 0, 0, 0], [4, 0, 0, 0]], "down"),
])
```

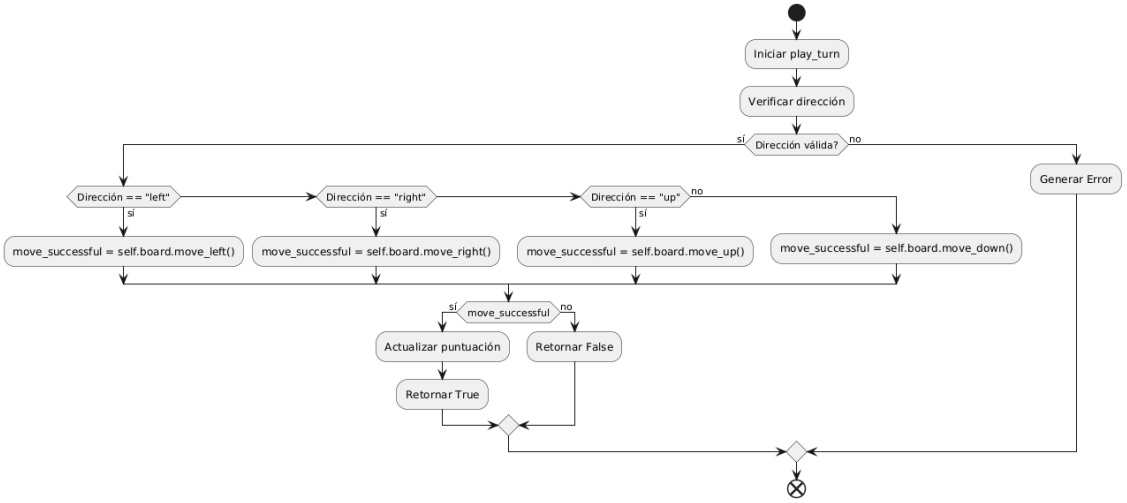
```python
@patch.object(Board, 'add_random_tile')  # Mock para evitar generación aleatoria
def test_play_turn_pairwise(mock_add_random_tile, initial_grid, expected_grid, direction):
    board = Board()
    for i in range(board.size):
        for j in range(board.size):
            board.grid[i][j].set_value(initial_grid[i][j])
    game = Game(size=4, board=board)

    # Llamar a la función de juego
    game.play_turn(direction)

    # Comprobar el estado del tablero después del movimiento
    assert [[cell.value for cell in row] for row in game.board.grid] == expected_grid
```

Loop testing:

```python
def play_turn(self, direction):
    """
    Realiza un turno completo en el juego: movimiento, puntuación y nueva ficha.
    """

    # Realizamos el movimiento
    if direction == "left":
        move_successful = self.board.move_left()
    elif direction == "right":
        move_successful = self.board.move_right()
    elif direction == "up":
        move_successful = self.board.move_up()
    elif direction == "down":
        move_successful = self.board.move_down()
    else:
        raise ValueError("Dirección inválida") #Precondicion: si direccion no es correcta no se hace movimiento

    assert isinstance(move_successful, bool) # poscondicion: se hace el movimiento

    if not move_successful:
        return False

    # Actualizamos la puntuación
    previous_score = self.score
    self.score += self.board.last_move_score

    assert self.score >= previous_score # Poscondicion: se aumenta puntuacion



    return True
```

```python
def test_play_turn_no_iterations():
    board = Board(size=4)  # Tablero 4x4 vacío
    game = Game(size=4, board=board)
    assert not game.play_turn("left")  # Sin iteraciones

def test_play_turn_one_iteration():
    board = Board(size=4)
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(2)  # Solo una fila con movimiento válido
    game = Game(size=4, board=board)
    assert game.play_turn("left")  # Iteraciones: 1 fila

def test_play_turn_multiple_iterations():
    board = Board(size=4)
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(4)  # Varias iteraciones posibles
    game = Game(size=4, board=board)
    assert not game.play_turn("up")  # Iteraciones: varias filas y columnas
```

Path coverage:




```python
def test_play_turn_invalid_direction():
    game = Game(size=4)
    with pytest.raises(ValueError):
        game.play_turn("diagonal")  # Camino 1: dirección no válida


def test_play_turn_no_valid_move():
    board = Board(size=4)
    game = Game(size=4, board=board)
    values = [
        [2, 4, 2, 4],
        [4, 2, 4, 2],
        [2, 4, 2, 4],
        [4, 2, 4, 2],
    ]
    for i in range(4):
        for j in range(4):
            board.grid[i][j].set_value(values[i][j])  # Tablero sin movimientos
    assert not game.play_turn("left")  # Camino 2: movimiento inválido
    assert not game.play_turn("right")  # Camino 2: movimiento inválido
    assert not game.play_turn("up")  # Camino 2: movimiento inválido
    assert not game.play_turn("down")  # Camino 2: movimiento inválido
```

```python
def test_play_turn_valid_move_left():
    board = Board(size=4)
    game = Game(size=4, board=board)
    board.grid[0][0].set_value(2)
    board.grid[0][1].set_value(2)
    assert game.play_turn("left")  # Camino 3: movimiento válido
    non_empty_cells = sum(
        not cell.is_empty() for row in game.board.grid for cell in row
    )
    assert non_empty_cells == 1  # Se añade una nueva ficha


def test_play_turn_valid_move_right():
    board = Board(size=4)
    game = Game(size=4, board=board)
    board.grid[0][2].set_value(2)
    board.grid[0][3].set_value(2)
    assert game.play_turn("right")  # Camino 4: movimiento válido
    non_empty_cells = sum(
        not cell.is_empty() for row in game.board.grid for cell in row
    )
    assert non_empty_cells == 1  # Se añade una nueva ficha
```

```python
def test_play_turn_valid_move_up():
    board = Board(size=4)
    game = Game(size=4, board=board)
    board.grid[2][0].set_value(2)
    board.grid[3][0].set_value(2)
    assert game.play_turn("up")  # Camino 5: movimiento válido
    non_empty_cells = sum(
        not cell.is_empty() for row in game.board.grid for cell in row
    )
    assert non_empty_cells == 1  # Se añade una nueva ficha


def test_play_turn_valid_move_down():
    board = Board(size=4)
    game = Game(size=4, board=board)
    board.grid[0][0].set_value(2)
    board.grid[1][0].set_value(2)
    assert game.play_turn("down")  # Camino 6: movimiento válido
    non_empty_cells = sum(
        not cell.is_empty() for row in game.board.grid for cell in row
    )
    assert non_empty_cells == 1  # Se añade una nueva ficha
```

Funcionalitat: is_game_over

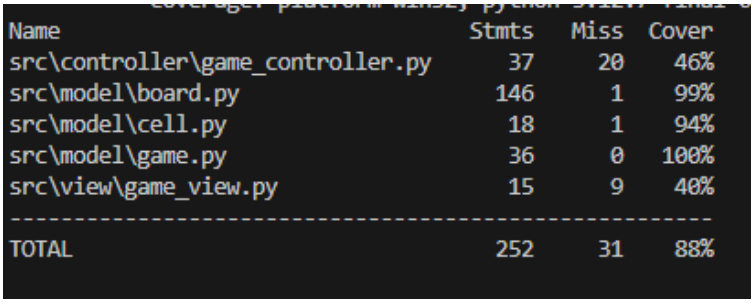
Localització:

- Arxiu: src/model/game.py
- Classe: Game
- Mètode desenvolupat: is_game_over

Test:

- Arxiu: tests/test_game.py
- Mètode de test associat a la funcionalitat: Caixa negra, caixa blanca
- Tècniques utilitzades: Statement coverage, Particions equivalents, Valors limit i frontera

Statement coverage



Name	Stmts	Miss	Cover
src\controller\game_controller.py	37	20	46%
src\model\board.py	146	1	99%
src\model\cell.py	18	1	94%
src\model\game.py	36	0	100%
src\view\game_view.py	15	9	40%
TOTAL	252	31	88%

Valors límit i frontera

```python
# Valor limit i frontera: Tauler completament ple sense moviments possibles
def test_is_game_over_with_full_board_no_moves():
    board = Board(size=4)
    game = Game(size=4, board=board)
    values = [
        [2, 4, 2, 4],
        [4, 2, 4, 2],
        [2, 4, 2, 4],
        [4, 2, 4, 2],
    ]
    for i in range(4):
        for j in range(4):
            board.grid[i][j].set_value(values[i][j])
    assert game.is_game_over()  # Tablero lleno y sin movimientos posibles
```

Particio equivalent:

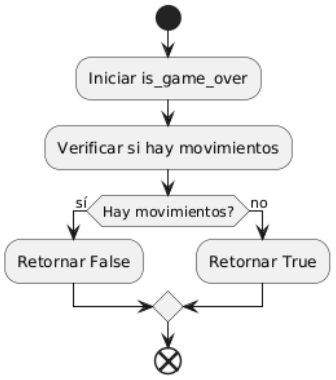
```python
def test_is_game_over_equivalence_partitions():
    """
    Verifica el comportamiento de is_game_over en particiones equivalentes.
    """
    # Partición 1: El juego NO ha terminado (hay movimientos disponibles)
    board_with_moves = Board(size=4)
    values_with_moves = [
        [2, 2, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
    ]
    for i in range(4):
        for j in range(4):
            board_with_moves.grid[i][j].set_value(values_with_moves[i][j])

    game_with_moves = Game(size=4, board=board_with_moves)

    # Asegurarse de que el juego NO ha terminado
    assert not game_with_moves.is_game_over()

    # Partición 2: El juego ha terminado (no hay movimientos disponibles)
    board_no_moves = Board(size=4)
    values_no_moves = [
        [2, 4, 2, 4],
        [4, 2, 4, 2],
        [2, 4, 2, 4],
        [4, 2, 4, 2],
    ]
    for i in range(4):
        for j in range(4):
            board_no_moves.grid[i][j].set_value(values_no_moves[i][j])

    game_no_moves = Game(size=4, board=board_no_moves)

    # Asegurarse de que el juego ha terminado
    assert game_no_moves.is_game_over()
```

Loop testing:

```python
def is_game_over(self):

    # Si hay movimientos disponibles, el juego no ha terminado
    if self.board.has_moves():
        return False

    # Poscondicion: devuelve false si hay movimientos

    # Si no hay movimientos disponibles, el juego ha terminado
    return True
```

```python
def test_is_game_over_no_iterations():
    board = Board(size=4)  # Tablero 4x4 lleno
    for i in range(4):
        for j in range(4):
            board.grid[i][j].set_value(2)  # Ninguna celda vacía
    game = Game(size=4, board=board)
    assert not game.is_game_over()  # Iteraciones: 16 (sin ceros)

def test_is_game_over_one_iteration():
    board = Board(size=4)
    for i in range(4):
        for j in range(4):
            board.grid[i][j].set_value(2)  # Rellenar todo menos una celda
    board.grid[0][0].set_value(0)  # Una celda vacía
    game = Game(size=4, board=board)
    assert not game.is_game_over()  # Iteraciones: 1

def test_is_game_over_multiple_iterations():
    board = Board(size=4)
    for i in range(4):
        for j in range(4):
            board.grid[i][j].set_value(2)  # Rellenar todo
    game = Game(size=4, board=board)
    assert not game.is_game_over()  # Iteraciones: 16 (sin ceros)
```

Condition coverage and path coverage:




```python
# Condition coverage and path coverage

def test_is_game_over_moves_available():
    board = Board(size=4)
    board.grid[0][0].set_value(2)  # Hay movimientos disponibles
    game = Game(size=4, board=board)
    assert not game.is_game_over()  # has_moves devuelve True -> Juego no terminado

def test_is_game_over_no_moves():
    board = Board(size=4)
    values = [
        [2, 4, 8, 16],
        [32, 64, 128, 256],
        [512, 1024, 2048, 2],
        [4, 8, 16, 32]
    ]
    for i in range(4):
        for j in range(4):
            board.grid[i][j].set_value(values[i][j])  # Sin movimientos disponibles
    game = Game(size=4, board=board)
    assert game.is_game_over()  # has_moves devuelve False -> Juego terminado
```

Funcionalitat: is_victory

Localització:

- Arxiu: src/model/game.py
- Classe: Game
- Mètode desenvolupat: is_victory

Test:

- Arxiu: tests/test_game.py
- Mètode de test associat a la funcionalitat: Caixa negra, caixa blanca
- Tècniques utilitzades: Statement coverage, Particions equivalents, Valors limit i frontera, Loop testing

Valors límit i frontera i particions equivalents

```python
def test_is_victory_with_victory():
    board = Board(size=4)
    board.grid[0][0].set_value(2048)  # Una celda con 2048
    game = Game(size=4, board=board)
    assert game.is_victory()  # cell.value == 2048 sale True -> Victoria

def test_is_victory_no_victory():
    board = Board(size=4)
    board.grid[0][0].set_value(2)  # Ninguna celda con 2048
    game = Game(size=4, board=board)
    assert not game.is_victory()  # cell.value == 2048 sale False -> No hay victoria
```

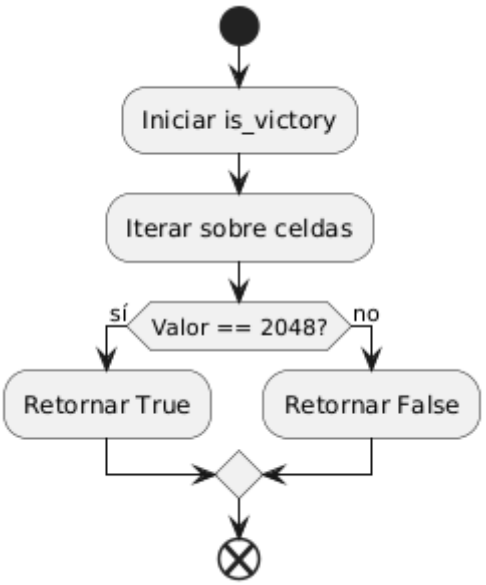
```python
def test_is_victory_equivalence_partitions():
    """
    Verifica el comportamiento de is_victory en particiones equivalentes.
    """
    # Partición 1: El juego ha ganado (hay al menos una celda con valor 2048)
    board_with_victory = Board(size=4)
    values_with_victory = [
        [2, 4, 2, 4],
        [4, 2048, 4, 2],
        [2, 4, 2, 4],
        [4, 2, 4, 2],
    ]
    for i in range(4):
        for j in range(4):
            board_with_victory.grid[i][j].set_value(values_with_victory[i][j])

    game_with_victory = Game(size=4, board=board_with_victory)

    # Asegurarse de que el juego ha ganado
    assert game_with_victory.is_victory()

    # Partición 2: El juego NO ha ganado (no hay celdas con valor 2048)
    board_no_victory = Board(size=4)
    values_no_victory = [
        [2, 4, 2, 4],
        [4, 8, 4, 2],
        [2, 4, 2, 4],
        [4, 2, 4, 2],
    ]
    for i in range(4):
        for j in range(4):
            board_no_victory.grid[i][j].set_value(values_no_victory[i][j])

    game_no_victory = Game(size=4, board=board_no_victory)

    # Asegurarse de que el juego NO ha ganado
    assert not game_no_victory.is_victory()
```

Loop testing:

```python
def is_victory(self):
    # Comprobar si hay alguna celda con valor 2048 (victoria)
    for row in self.board.grid:
        for cell in row:
            if cell.value == 2048:
                return True  # El juego se ha ganado

    return False
```

```python
def test_is_victory_no_iterations():
    board = Board(size=4)  # Tablero 4x4 vacío
    game = Game(size=4, board=board)
    assert not game.is_victory()  # Iteraciones: 0 (sin 2048)

def test_is_victory_one_iteration():
    board = Board(size=4)
    board.grid[0][0].set_value(2048)  # Victoria en la primera celda
    game = Game(size=4, board=board)
    assert game.is_victory()  # Iteraciones: 1

def test_is_victory_multiple_iterations():
    board = Board(size=4)
    board.grid[1][1].set_value(2048)  # Victoria en otra celda
    game = Game(size=4, board=board)
    assert game.is_victory()  # Iteraciones: multiples, pero se detiene en la primera coincidencia

def test_is_victory_no_victory():
    board = Board(size=4)
    game = Game(size=4, board=board)  # Ninguna celda tiene 2048
    assert not game.is_victory()  # Iteraciones: 16 (4x4)
```

Condition coverage and path coverage:




```python
def test_is_victory_with_victory():
    board = Board(size=4)
    board.grid[0][0].set_value(2048)  # Una celda con 2048
    game = Game(size=4, board=board)
    assert game.is_victory()  # cell.value == 2048 sale True -> Victoria

def test_is_victory_no_victory():
    board = Board(size=4)
    board.grid[0][0].set_value(2)  # Ninguna celda con 2048
    game = Game(size=4, board=board)
    assert not game.is_victory()  # cell.value == 2048 sale False -> No hay victoria
```