2048 Game Developed with JavaScript Following a TDD Implementation

Test i Qualitat del Software

Pol Pagès Luque - **1494769**
Sergi Vila Varas - **1531267**

Introduction

The purpose of this report is to explain and detail all the tests programmed for the 2048 game[1].

2048 is a game where you combine numbered tiles in order to gain a higher numbered tile. In this game you start with two tiles, the lowest possible number available is two. Then you will play by combining the tiles with the same number to have a tile with the sum of the number on the two tiles.

The game of 2048 does not include complicated controls. The controls you'll be using are just upward, downward, and sideways. The rules are also simple. You just need to move the tiles, and every time you move, another tile pops up in a random place anywhere in the board. When two tiles with the same number on them collide with one another as you move them, they will merge into one tile with the sum of the numbers written on them initially.

The project can be found in this link:

➔ **https://github.com/polpages1999/2048-tdd.git**

The game is coded under Javascript and we used Jest[2] as the testing framework for the project.

The game can be played by just opening a *Live Preview* of the 'index.html' file. In the repository *README.md* you can find a detailed guide on how to download and set up the project locally.

[1] https://www.ripublication.com/aama17/aamav12n1_01.pdf
[2] https://jestjs.io/

Rules and Usability

Basic Rules

★ **Grid Layout**: The game is played on a 4x4 grid, comprising 16 cells. Each cell can either be empty or contain a tile with a numerical value.
★ **Starting the Game**: At the beginning, two tiles, each with a value of either 2 or 4, are placed in random positions on the grid.
★ **Player Moves**: Players can slide the tiles in one of four directions: up, down, left, or right. Every valid move shifts all the tiles in the chosen direction.
★ **Merging Tiles**: When two tiles of the same number collide while moving, they merge into a single tile. The value of this new tile is the sum of the two originals. This merging plays a crucial role in progressing in the game.
★ **Scoring**: Every time two tiles merge, the player's score increases by the value of the new tile.
★ **Adding New Tiles**: After each move, a new tile (either 2 or 4) appears in a random empty spot on the grid.
★ **Game Over**: The game ends when the grid is full, and no adjacent tiles can be merged.
★ **Winning the Game**: A player wins by creating a tile with the number 2048.

Usability Features

★ **Intuitive Controls**: Players interact with the game by selecting a direction to slide the tiles. This simplicity makes it accessible and easy to understand.
★ **Progress Tracking**: The game tracks and displays the player's current score, allowing them to gauge their performance.
★ **End-Game Detection**: The game automatically detects a win or loss condition, providing immediate feedback to the player on their game status.
★ **Adaptive Difficulty**: The difficulty of the game naturally increases as the board fills up, offering a challenging but rewarding experience.
★ **Test-Driven Development**: The game's implementation, guided by TDD, ensures robustness and reliability, contributing to a seamless user experience.

Tests

Equivalent Partitions

This involves breaking down the inputs of the software into partitions that can be tested as equal. This involves partitions for the game board states, such as empty board, board with some tiles, and full board.

1. Empty Board Initialization

- **Objective**: To verify that a new game board initializes correctly with all cells empty.
- **Test**: Check if every cell in a new Board instance is 0.
- **Expected Result**: The test should confirm that all cells are empty.
- **Code:**

```
test('should start with an empty board', () => {
    const gameBoard = new Board();
    const isEmpty = gameBoard.board.every(row => row.every(cell => cell === 0));
    expect(isEmpty).toBeTruthy();
});
```

2. Initial Tile Placement

- **Objective**: To ensure that exactly two tiles appear on the board at game start.
- **Test**: Count the number of non-zero cells after calling addInitialTiles().
- **Expected Result**: There should be exactly two non-zero cells.
- **Code**:

```
test('should have two tiles on the board after initialization', () => {
    const gameBoard = new Board();
    gameBoard.addInitialTiles();
    const tileCount = gameBoard.board.flat().filter(cell => cell !== 0).length;
    expect(tileCount).toBe(2);
});
```

3. Move

- **Objective**: To confirm that tiles move and merge correctly when moved.
- **Test**: Set a predefined board state and perform a move, then check the new state of the board.
- **Expected Result**: Tiles should shift in the correct direction, combining equal adjacent tiles.
- **Code**:

```
test('should move tiles down when the down move is made', () => {
    const gameBoard = new Board();
    gameBoard.board = [
        [0, 0, 2, 4],
        [0, 0, 2, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]
    ];
    gameBoard.moveDown();
    expect(gameBoard.board[3]).toEqual([0, 0, 4, 4]);
})
```

4. Tile Combination

- **Objective**: To verify that tile combination logic is correctly implemented.
- **Test**: Set up a board where tiles are positioned to combine upon movement and then perform the move.
- **Expected Result**: Adjacent equal tiles should merge into a single tile with their combined value.
- **Code**:

```
    test('should combine tiles correctly when moved', () => {
        const gameBoard = new Board();
        gameBoard.board = [
            [2, 2, 4, 0],
            [4, 4, 0, 0],
            [0, 0, 0, 0],
            [0, 0, 0, 0]
        ];
        gameBoard.moveLeft();
        expect(gameBoard.board[0]).toEqual([4, 4, 0, 0]);
        expect(gameBoard.board[1]).toEqual([8, 0, 0, 0]);
    });
```

5. End Game Check

- **Objective**: To test if the game correctly identifies the 'no more moves' condition.
- **Test**: Fill the board with a pattern that allows no further moves and call hasLost().
- **Expected Result**: The method should return true, indicating the game is over.
- **Code**:

```
    test('should determine if no moves are left', () => {
        const gameBoard = new Board();
        gameBoard.board = [
            [2, 4, 2, 4],
            [4, 2, 4, 2],
            [2, 4, 2, 4],
            [4, 2, 4, 2]
        ];
        const noMovesLeft = gameBoard.hasLost();
        expect(noMovesLeft).toBeTruthy();
    });
});
```

Limit and Frontier Values

Limit and frontier value testing focuses on the edge cases and boundary conditions of software functionality. Frontier values include the highest possible tile before reaching 2048, or the move that causes a win or lose state.

1. Win Recognition at 2048

- **Objective**: To test if the game correctly identifies a win when a 2048 tile is formed.
- **Test**: Manually set a board state that can merge two tiles to form 2048 and perform the merge.
- **Expected Result**: The game should recognize the win when the 2048 tile is created.
- **Code**:

```
test('should recognize a win when 2048 tile is reached', () => {
    const gameBoard = new Board();
    gameBoard.board = [
        [1024, 1024, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]
    ];
    gameBoard.moveLeft();
    expect(gameBoard.hasWon()).toBeTruthy();
});
```

2. No Horizontal Moves

- **Objective**: To verify the game correctly identifies when no horizontal moves are left.
- **Test**: Arrange the board with alternating tiles such that no horizontal merges are possible and check horizontal move availability.
- **Expected Result**: The game should report that no horizontal moves are available.
- **Code**:

```
test('should identify no horizontal moves left', () => {
    const gameBoard = new Board();
    gameBoard.board = [
        [2, 4, 2, 4],
        [4, 2, 4, 2],
        [2, 4, 2, 4],
        [4, 2, 4, 2]
    ];
    expect(gameBoard.canMoveHorizontal()).toBeFalsy();
});
```

3. No Vertical Moves

- **Objective**: To confirm the game correctly identifies when no vertical moves are left.
- **Test**: Set up the board with alternating tiles to prevent vertical merges and evaluate vertical move availability.

- **Expected Result**: The game should indicate that no vertical moves are available.
- **Code**:

```
test('should identify no vertical moves left', () => {
    const gameBoard = new Board();
    gameBoard.board = [
        [2, 4, 2, 4],
        [4, 2, 4, 2],
        [2, 4, 2, 4],
        [4, 2, 4, 2]
    ];
    expect(gameBoard.canMoveVertical()).toBeFalsy();
});
```

4. Full Board with Possible Moves

- **Objective**: To ensure that the game does not end prematurely when the board is full but moves are still available.
- **Test**: Fill the board in a manner that leaves no empty spaces but allows for at least one merge, and then check if the game is considered over.
- **Expected Result**: The game should not end, acknowledging the availability of moves.
- **Code**:

```
test('should not end game if the board is full but moves are available', () => {
    const gameBoard = new Board();
    gameBoard.board = [
        [2, 4, 2, 4],
        [4, 2, 4, 2],
        [2, 4, 2, 4],
        [4, 2, 4, 4]
    ];
    expect(gameBoard.hasLost()).toBeFalsy();
});
```

5. Highest Tile Identification Post-Game

- **Objective**: To test whether the game can correctly identify the highest-value tile on the board after the game is over.
- **Test**: Create a board with various tiles and no possible moves, and retrieve the highest tile value.
- **Expected Result**: The game should accurately report the highest tile value on the board.
- **Code**:

```
test('should report the highest tile after game over', () => {
    const gameBoard = new Board();
    gameBoard.board = [
        [1024, 64, 32, 16],
        [512, 256, 128, 64],
        [256, 128, 64, 32],
        [128, 64, 32, 16]
    ];
    expect(gameBoard.getHighestTile()).toBe(1024);
});
```

Pairwise Testing

Since 2048 is a deterministic game where the outcome is solely dependent on the game's state and the player's move, we will consider the game state (the board configuration) and the player's move (up, down, left, right) for pairwise testing.

Pairwise testing is usually done when the number of inputs is small, as it's meant to be a more efficient way to get good coverage with fewer tests. Given that the 2048 game has a large number of possible board states, it's not feasible to do pairwise testing for all possible states and moves. Therefore, we focused on representative states that are likely to uncover bugs. For simplicity, we will consider a reduced set of board states, on the other hand you can find on the code, tests covering all 4 directions:

1. Single Combination

- **Objective**: To test the game's response to a single combinable pair of tiles when moving.
- **Test**: Set up a board where only two tiles can combine by moving and then execute the move.
- **Expected Result**: The two tiles should merge into one tile.
- **Code**:

```
    test('Single combination move available - move up', () => {
        const gameBoard = new Board();
        gameBoard.board = [
            [0, 2, 0, 0],
            [0, 2, 0, 0],
            [0, 0, 0, 0],
            [0, 0, 0, 0]
        ];
        gameBoard.moveUp();
        expect(gameBoard.board[0]).toEqual([0, 4, 0, 0]);
    });
```

2. Multiple Combinations

- **Objective**: To evaluate the game's handling of multiple combinable pairs in a single move.
- **Test**: Configure multiple combinable pairs in columns and move.
- **Expected Result**: All possible combinations should occur.
- **Code**:

```
    test('Multiple combination moves available - move up', () => {
        const gameBoard = new Board();
        gameBoard.board = [
            [2, 2, 4, 4],
            [0, 2, 4, 0],
            [0, 0, 0, 4],
            [2, 0, 0, 0]
        ];
        gameBoard.moveUp();
        expect(gameBoard.board[0]).toEqual([4, 4, 8, 8]);
    });
```

3. No Combination but Moves Available

- **Objective**: To check if the game allows movement without combinations.
- **Test**: Create a scenario where tiles can move but not combine.
- **Expected Result**: The tiles should move without combining.
- **Code**:

```
test('No combination but moves available - move down', () => {
    const gameBoard = new Board();
    gameBoard.board = [
        [2, 4, 2, 4],
        [4, 2, 4, 2],
        [0, 0, 0, 0],
        [0, 0, 0, 0]
    ];
    gameBoard.moveDown();
    expect(gameBoard.board[2]).toEqual([2, 4, 2, 4]);
    expect(gameBoard.board[3]).toEqual([4, 2, 4, 2]);
});
```

4. No Moves Possible (Game Over)

- **Objective**: To verify the game correctly identifies a no-move scenario when attempting to move.
- **Test**: Set up a full board with no possible moves and attempt a move.
- **Expected Result**: The game should recognize that no moves are possible, indicating game over.
- **Code**:

```
test('No moves possible (game over) - attempt move up', () => {
    const gameBoard = new Board();
    gameBoard.board = [
        [2, 4, 2, 4],
        [4, 2, 4, 2],
        [2, 4, 2, 4],
        [4, 2, 4, 2]
    ];
    gameBoard.moveUp();
    expect(gameBoard.hasLost()).toBeTruthy();
});
```

Mock Objects

2048 involves a user interface and possibly randomness, we used mocks to simulate these parts during testing.

To implement the logic for the tests that use mock objects, we focused on the specific areas where mocking is beneficial: random tile generation, user input, and potentially the rendering process. Since Jest provides powerful mocking capabilities, we can use it to mock functions or modules as needed.

Based on the render method we implemented in the Board class, we created tests that simulate the next four possible states of the board. These states will typically involve adding tiles and making moves. Here are five tests that cover different scenarios:

1. Tests 30-33: Mocking Random Tile Generation

- **Objective**: To test the game's behavior when specific tiles are added to the board.
- **Method**: Replace the addRandomTile method with a mock function that adds a predetermined tile (either 2 or 4) to a specified position.
- **Expected Result**: The specified tile should be added to the correct position, and the mock function should be called as expected.
- **Code**:

```
test('adds a random tile of 2 at an empty position', () => {
    const gameBoard = new Board();
    gameBoard.addRandomTile = jest.fn(() => {
        gameBoard.board[0][0] = 2;
    });
    gameBoard.addRandomTile();
    expect(gameBoard.board[0][0]).toBe(2);
    expect(gameBoard.addRandomTile).toHaveBeenCalled();
});
test('adds a random tile of 2 at an empty position', () => {
    const gameBoard = new Board();
    gameBoard.addRandomTile = jest.fn(() => {
        gameBoard.board[2][3] = 2;
    });
    gameBoard.addRandomTile();
    expect(gameBoard.board[2][3]).toBe(2);
    expect(gameBoard.addRandomTile).toHaveBeenCalled();
});
test('adds a random tile of 4 at an empty position', () => {
    const gameBoard = new Board();
    gameBoard.addRandomTile = jest.fn(() => {
        gameBoard.board[0][0] = 4;
    });
    gameBoard.addRandomTile();
    expect(gameBoard.board[0][0]).toBe(4);
    expect(gameBoard.addRandomTile).toHaveBeenCalled();
});
test('adds a random tile of 4 at an empty position', () => {
    const gameBoard = new Board();
    gameBoard.addRandomTile = jest.fn(() => {
        gameBoard.board[3][2] = 4;
    });
    gameBoard.addRandomTile();
    expect(gameBoard.board[3][2]).toBe(4);
    expect(gameBoard.addRandomTile).toHaveBeenCalled();
});
```

2. Tests 34-37: Mocking User Input for Tile Movement

- **Objective**: To evaluate the game's response to different movement directions without actual user input.
- **Method**: Set up a specific board state, mock user input for different directions (up, down, left, right), and invoke the move method.
- **Expected Result**: Tiles should move in the specified direction, combine correctly if applicable, and a new random tile should be added to the board.
- **Code**:

```
test('processes user input to move tiles left', () => {
    const gameBoard = new Board();
    gameBoard.board = [
        [2, 2, 4, 4],
        [0, 0, 0, 0],
        [4, 0, 4, 2],
        [0, 0, 0, 0]
    ];
    const userInput = jest.fn(() => 'left');
    gameBoard.move(userInput());
    expect(gameBoard.board[0].slice(0, 2)).toEqual([4, 8]);
    expect(gameBoard.board[2].slice(0, 2)).toEqual([8, 2]);
    // Here we check if a random tile was added after the move.
    expect(gameBoard.board.flat().filter(value => value !== 0).length).toBe(5);
});
test('processes user input to move tiles right', () => {
    const gameBoard = new Board();
    gameBoard.board = [
        [2, 2, 4, 4],
        [0, 0, 0, 0],
        [4, 0, 4, 2],
        [0, 0, 0, 0]
    ];
    const userInput = jest.fn(() => 'right');
    gameBoard.move(userInput());
```

```
        expect(gameBoard.board[0].slice(-2)).toEqual([4, 8]);
        expect(gameBoard.board[2].slice(-2)).toEqual([8, 2]);
        // Here we check if a random tile was added after the move.
        expect(gameBoard.board.flat().filter(value => value !== 0).length).toBe(5);
    });
    test('processes user input to move tiles up', () => {
        const gameBoard = new Board();
        gameBoard.board = [
            [2, 2, 4, 4],
            [0, 0, 0, 0],
            [4, 0, 4, 2],
            [0, 0, 0, 0]
        ];
        const userInput = jest.fn(() => 'up');
        gameBoard.move(userInput());
        expect(gameBoard.board[0]).toEqual([2, 2, 8, 4]);
        expect(gameBoard.board[1].slice(-1)).toEqual([2]);
        expect(gameBoard.board[1].slice(0, 1)).toEqual([4]);
        // Here we check if a random tile was added after the move.
        expect(gameBoard.board.slice(-3).flat().filter(value => value !==
0).length).toBe(3);
    });
    test('processes user input to move tiles down', () => {
        const gameBoard = new Board();
        gameBoard.board = [
            [2, 2, 4, 4],
            [0, 0, 0, 0],
            [4, 0, 4, 2],
            [0, 0, 0, 0]
        ];
        const userInput = jest.fn(() => 'down');
        gameBoard.move(userInput());
        expect(gameBoard.board[3]).toEqual([4, 2, 8, 2]);
        expect(gameBoard.board[2].slice(0, 1)).toEqual([2]);
        expect(gameBoard.board[2].slice(-1)).toEqual([4]);
        // Here we check if a random tile was added after the move.
        expect(gameBoard.board.slice(0, 3).flat().filter(value => value !==
0).length).toBe(3);
    });
```

3. Tests 38-42: Mocking Game Rendering

- **Objective**: To test the visual representation of the board state under different scenarios.
- **Method**: After performing various actions like adding initial tiles or moving tiles, invoke the render method to simulate the game's visual output.
- **Expected Result**: The rendered board state should correctly reflect the result of the performed actions.
- **Code**:

```
test('Board state after adding initial tiles', () => {
    let gameBoard = new Board();
    gameBoard.addInitialTiles();
    const renderedOutput = gameBoard.render();

    // Split the output into rows and then into cells
    const allCells = renderedOutput.split('\n').flatMap(row => row.split(' '));

    // Filter out the '0's and count the non-zero cells
    const nonZeroCells = allCells.filter(cell => cell !== '0');
    const tileCount = nonZeroCells.length;

    // Check if two tiles have been added
    expect(tileCount).toBe(2);
});
test('Board state after moving left', () => {
    let gameBoard = new Board();
    gameBoard.board = [
        [2, 2, 4, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]
    ];
    gameBoard.moveLeft();
    expect(gameBoard.render()).toBe('4 4 0 0\n0 0 0 0\n0 0 0 0\n0 0 0 0');
});
test('Board state after moving right', () => {
    let gameBoard = new Board();
    gameBoard.board = [
        [2, 2, 4, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]
    ];
    gameBoard.moveRight();
    expect(gameBoard.render()).toBe('0 0 4 4\n0 0 0 0\n0 0 0 0\n0 0 0 0');
});
test('Board state after moving up', () => {
        let gameBoard = new Board();
        gameBoard.board = [
            [2, 0, 0, 0],
            [2, 0, 0, 0],
            [4, 0, 0, 0],
            [0, 0, 0, 0]
        ];
        gameBoard.moveUp();
        expect(gameBoard.render()).toBe('4 0 0 0\n4 0 0 0\n0 0 0 0\n0 0 0 0');
    });
    test('Board state after moving down', () => {
        let gameBoard = new Board();
        gameBoard.board = [
            [2, 0, 0, 0],
            [2, 0, 0, 0],
            [4, 0, 0, 0],
            [0, 0, 0, 0]
        ];
        gameBoard.moveDown();
        expect(gameBoard.render()).toBe('0 0 0 0\n0 0 0 0\n4 0 0 0\n4 0 0 0');
    });
});
```

Loop Testing

This involves ensuring that loops in the code work correctly under various conditions. For our 'Board' class, we identified a few key areas where loop testing will be beneficial:

1. Testing the *addRandomTile* method (Simple Loop): This method contains a loop that continues until a random tile is successfully added. We tested for scenarios where the first attempt succeeds, where multiple attempts are needed, and the edge case where the board is full.

```
    test('addRandomTile adds a tile to an empty board', () => {
        const gameBoard = new Board();
        gameBoard.addRandomTile();
        const tiles = gameBoard.board.flat().filter(tile => tile !== 0);
        expect(tiles.length).toBe(1); // Expect exactly one tile to be added
    });

    test('addRandomTile does not add a tile to a full board', () => {
        const gameBoard = new Board();
        gameBoard.board = gameBoard.board.map(row => row.map(() => 2)); // Fill the
board
        gameBoard.addRandomTile();
        const tiles = gameBoard.board.flat().filter(tile => tile === 0);
        expect(tiles.length).toBe(0); // Expect no empty tiles
    });
```

2. Testing the *moveLeft* or *moveRight* methods (Nested Loops): These methods contain nested loops and handle multiple scenarios including shifting tiles, merging them, and filling in zeros. We tested cases like moving when no merge is possible, when a merge happens, and when the row is already in its final state.

```
    test('moveLeft combines tiles correctly', () => {
        const gameBoard = new Board();
        gameBoard.board = [
            [2, 2, 4, 4],
            [0, 0, 0, 0],
            [0, 0, 0, 0],
            [0, 0, 0, 0]
        ];
        gameBoard.moveLeft();
        expect(gameBoard.board[0]).toEqual([4, 8, 0, 0]);
    });

    test('moveLeft on a row with no merges', () => {
        const gameBoard = new Board();
        gameBoard.board = [
            [2, 4, 8, 16],
            [0, 0, 0, 0],
            [0, 0, 0, 0],
            [0, 0, 0, 0]
        ];
        gameBoard.moveLeft();
        expect(gameBoard.board[0]).toEqual([2, 4, 8, 16]); // Row remains unchanged
    });
```

3. Testing the *canMoveHorizontal* and *canMoveVertical* methods (Simple Loops): These methods involve simple loops to check if any moves are possible in the respective directions. Test includes scenarios with no moves possible, moves possible in one direction but not the other, and moves possible in both directions.

```
    test('canMoveHorizontal with no possible moves', () => {
        const gameBoard = new Board();
        gameBoard.board = [
            [2, 4, 8, 16],
            [16, 8, 4, 2],
            [2, 4, 8, 16],
            [16, 8, 4, 2]
        ];
        expect(gameBoard.canMoveHorizontal()).toBeFalsy();
    });
```

```
test('canMoveVertical with a possible move', () => {
    const gameBoard = new Board();
    gameBoard.board = [
        [2, 4, 8, 16],
        [2, 8, 4, 2],
        [0, 4, 8, 16],
        [0, 0, 0, 0]
    ];
    expect(gameBoard.canMoveVertical()).toBeTruthy();
});
```

Statement Coverage

In the provided tests, statement coverage ensures that every statement in the Board class is executed. For example, tests that initialize the game, add tiles, and move tiles, ensure that the respective constructor, addRandomTile, and move methods are executed. Later on we provide a more in-depth analysis of the testing coverage.

Decision Coverage

Tests that check for win, loss, and valid/invalid moves (e.g., tests 49, 50, 52, and 53) cover decision points in the game logic, ensuring both outcomes of decisions (true and false) are evaluated. Later on we provide a more in-depth analysis of the testing coverage.

Condition Coverage

In the game code, condition coverage ensures that conditions in methods like *canMoveHorizontal*, *hasWon*, and *hasLost* are tested for all possible outcomes. Test cases 47 and 48 are examples where condition coverage is applicable. Later on we provide a more in-depth analysis of the testing coverage.

Path Coverage

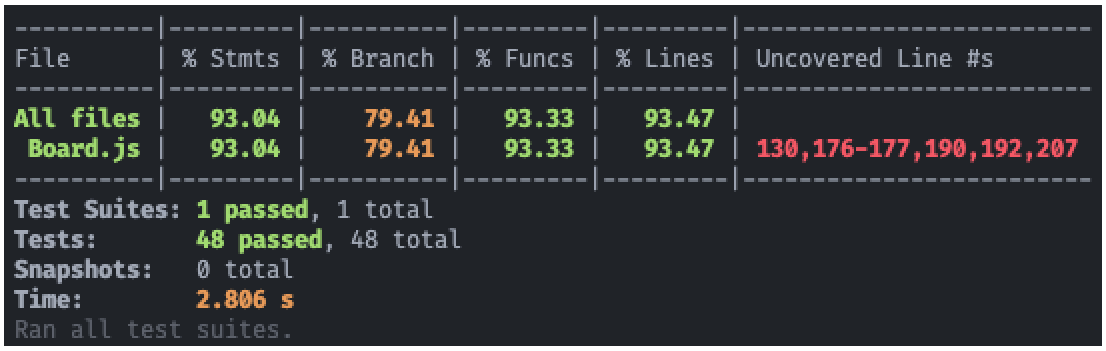
This involves ensuring every possible route through a given part of the code is tested. For 2048, this means testing every possible way a game can progress. Path coverage is more extensive than statement or decision coverage and involves testing every possible route through methods like move and checkGameOver. For instance, test 53 ensures that the path where an invalid move is made is tested, while test 54 checks the path where a win is announced. Later on we provide a more in-depth analysis of the testing coverage.

Commented Code

Meaningful comments were added throughout the code along with the implementation of the code and tests. We consider that the code is properly commented.

Test Coverage Reports

Test Coverage 1: Before ensuring any coverage.



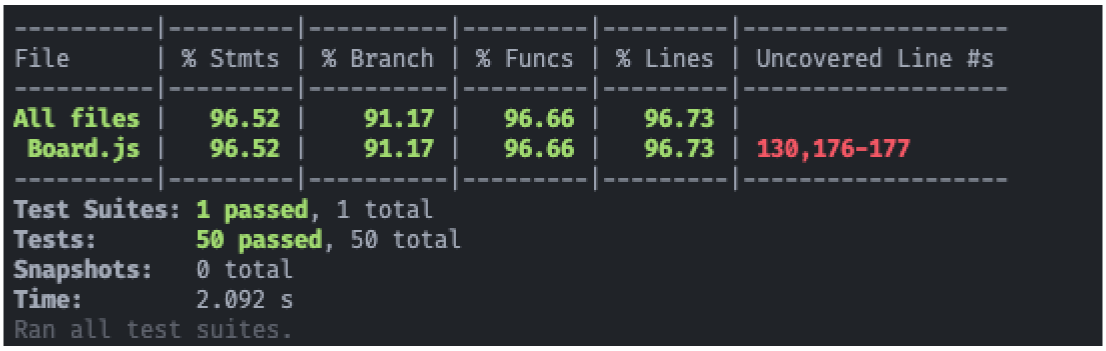
```
----------|---------|----------|---------|---------|-------------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------------
All files |   93.04 |    79.41 |   93.33 |   93.47 |
 Board.js |   93.04 |    79.41 |   93.33 |   93.47 | 130,176-177,190,192,207
----------|---------|----------|---------|---------|-------------------------
Test Suites: 1 passed, 1 total
Tests:       48 passed, 48 total
Snapshots:   0 total
Time:        2.806 s
Ran all test suites.
```

Added tests:

```
// 49. Ensure that checkGameOver recognizes a win when the board is full and no
moves are possible.
   test('checkGameOver recognizes a win', () => {
       const gameBoard = new Board();
       gameBoard.board[0][0] = 2048;
       gameBoard.checkGameOver();
       // Expect that gameBoard.hasWon() returns true
       expect(gameBoard.hasWon()).toBeTruthy();
   });

   // 50. Ensure that checkGameOver recognizes a loss when the board is full and no
moves are possible.
   test('checkGameOver recognizes a loss', () => {
       const gameBoard = new Board();
       gameBoard.board = [
           [2, 4, 8, 16],
           [4, 8, 16, 32],
           [8, 16, 32, 64],
           [16, 32, 64, 128]
       ];
       gameBoard.checkGameOver();
       // Expect that gameBoard.hasLost() returns true
       expect(gameBoard.hasLost()).toBeTruthy();
   });

   // 51. Ensure that addRandomTile does not add a tile when the board is full and
no moves are possible.
   test('addRandomTile does not add a tile when the board is full', () => {
       const gameBoard = new Board();
```

```
        gameBoard.board = gameBoard.board.map(row => row.map(() => 2)); // Fill the
board
        const result = gameBoard.addRandomTile();
        // Expect that no tile was added
        expect(result).toBeFalsy();
    });
```

Test Coverage 2: Fixed coverage on function checkGameOver() and addRandomTile().

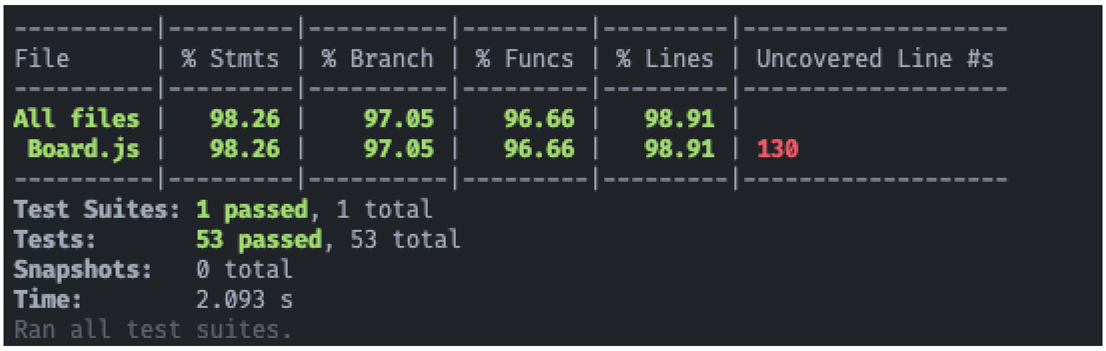


```
----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |   96.52 |    91.17 |   96.66 |   96.73 |
 Board.js |   96.52 |    91.17 |   96.66 |   96.73 | 130,176-177
----------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       50 passed, 50 total
Snapshots:   0 total
Time:        2.092 s
Ran all test suites.
```

Added tests:

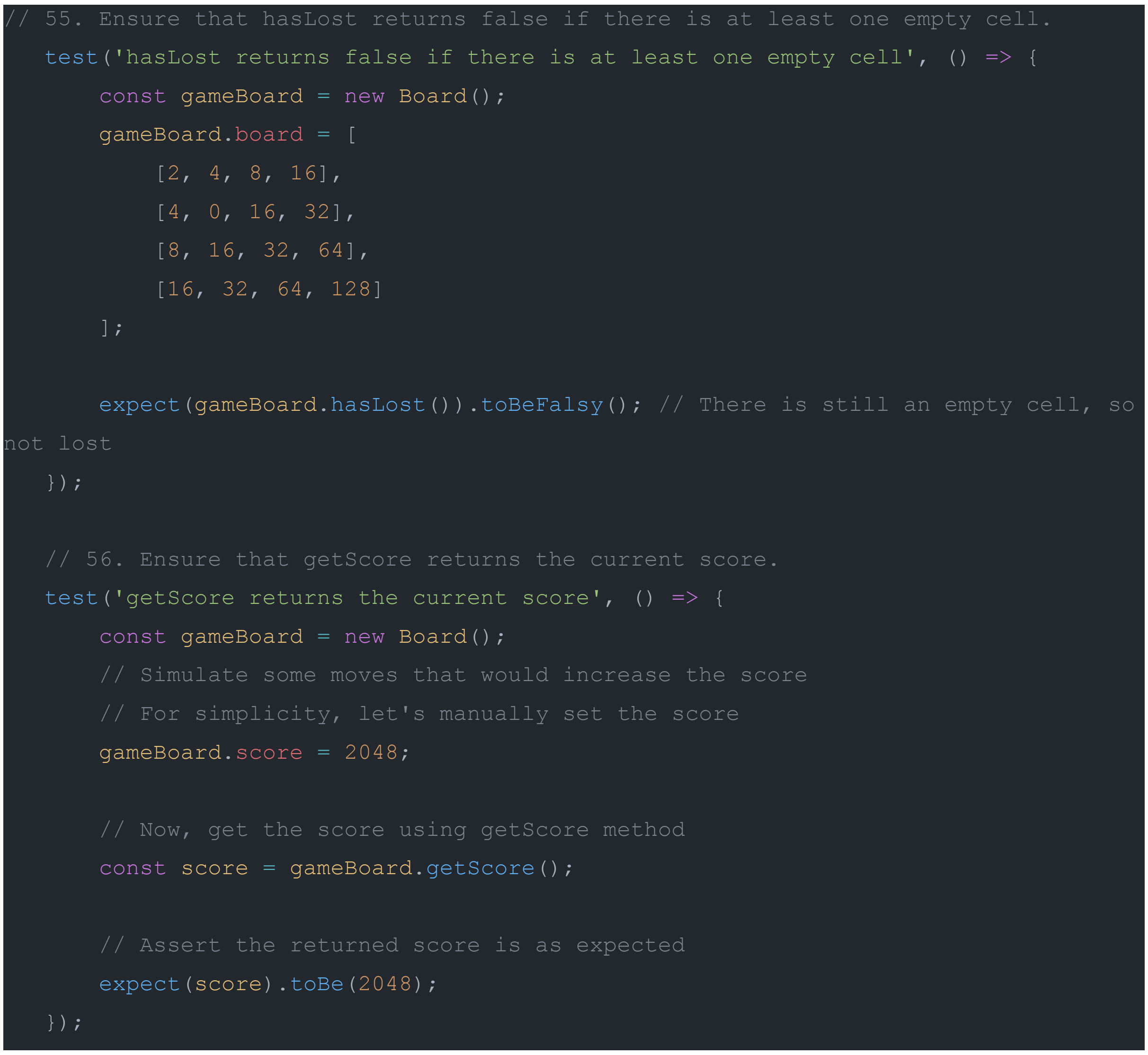
```
// 53. Ensure that move() does nothing when the direction is invalid.
    test('move with invalid direction does nothing', () => {
        const gameBoard = new Board();
        const initialBoardState = gameBoard.board.map(row => [...row]); // Clone the
initial state
        gameBoard.move('invalidDirection');
        // Expect that the board did not change
        expect(gameBoard.board).toEqual(initialBoardState);
    });

    // 54. Ensure that checkGameOver announces win correctly when a winning tile is
found.
    test('checkGameOver announces win correctly', () => {
        const gameBoard = new Board();
        gameBoard.board[0][0] = 2048; // Set a winning tile
        document.body.innerHTML = `<div id="board"></div>`; // Mock the board
element
        const consoleSpy = jest.spyOn(console, 'log'); // Spy on console.log to
verify output

        gameBoard.checkGameOver();
        expect(consoleSpy).toHaveBeenCalledWith('Congratulations, You Won!');
        consoleSpy.mockRestore(); // Restore original console.log
    });
```

Test Coverage 3: Fixed coverage for the move() function.



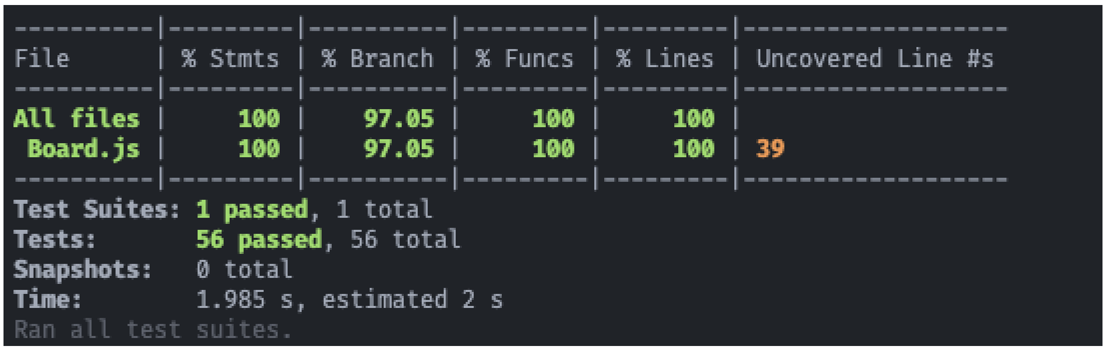
```
----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |   98.26 |    97.05 |   96.66 |   98.91 |
 Board.js |   98.26 |    97.05 |   96.66 |   98.91 | 130
----------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       53 passed, 53 total
Snapshots:   0 total
Time:        2.093 s
Ran all test suites.
```

Added tests:



```
// 55. Ensure that hasLost returns false if there is at least one empty cell.
   test('hasLost returns false if there is at least one empty cell', () => {
       const gameBoard = new Board();
       gameBoard.board = [
           [2, 4, 8, 16],
           [4, 0, 16, 32],
           [8, 16, 32, 64],
           [16, 32, 64, 128]
       ];

       expect(gameBoard.hasLost()).toBeFalsy(); // There is still an empty cell, so
not lost
   });

   // 56. Ensure that getScore returns the current score.
   test('getScore returns the current score', () => {
       const gameBoard = new Board();
       // Simulate some moves that would increase the score
       // For simplicity, let's manually set the score
       gameBoard.score = 2048;

       // Now, get the score using getScore method
       const score = gameBoard.getScore();

       // Assert the returned score is as expected
       expect(score).toBe(2048);
   });
```

Test Coverage 4: Fixed coverage for function hasLost() and getScore().

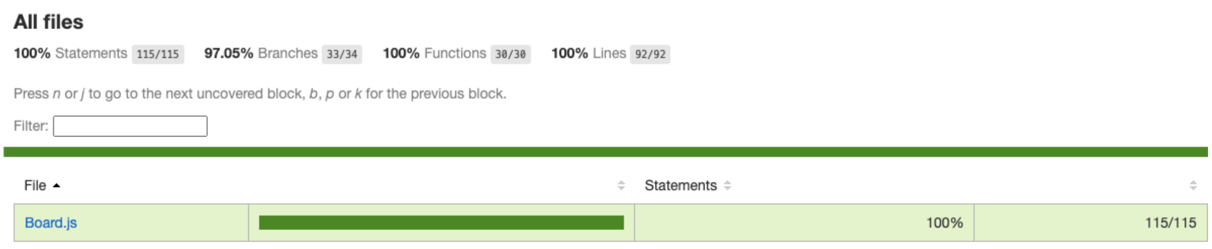


```
----------|---------|----------|---------|---------|-------------------
File      | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------|---------|----------|---------|---------|-------------------
All files |     100 |    97.05 |     100 |     100 |
 Board.js |     100 |    97.05 |     100 |     100 | 39
----------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       56 passed, 56 total
Snapshots:   0 total
Time:        1.985 s, estimated 2 s
Ran all test suites.
```

Observations

In our case, one of the execution branches appears as not covered. This is due to the function 'addRandomTile()'. This function returns a new tile on the board with a random value between 2 or 4. The branch not covered refers to one of this values not being covered.
We have not contemplated this branch coverage in our testing because it is just a matter of randomness, sometimes it is covered and sometimes it is not, depending on what the function returns. There are other tests that assure that this function works correctly, so we did not consider it as not covered. The code line causing this issue is the following:

```
this.board[row][col] = Math.random() < 0.9 ? 2 : 4;
```

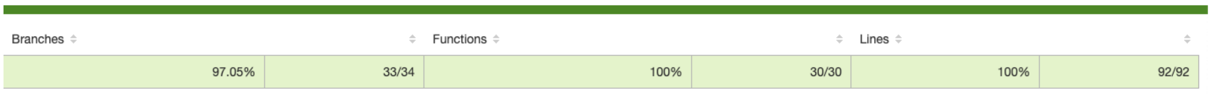


All files
100% Statements 115/115 **97.05%** Branches 33/34 **100%** Functions 30/30 **100%** Lines 92/92

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File		Statements	
Board.js		100%	115/115



Branches		Functions		Lines	
97.05%	33/34	100%	30/30	100%	92/92

Conclusions

Overview

This report has detailed a comprehensive testing strategy for the 2048 game, encompassing various testing methodologies to ensure the reliability, functionality, and robustness of the game. The testing approach included equivalent partitions, limit and frontier tests, pairwise testing, mock object tests, loop testing, and thorough test coverage analysis.

Key Findings

- **Robustness and Reliability**: Through a combination of testing methods, the game demonstrates robustness in handling various game scenarios, from initialization to gameplay mechanics like tile movement and combination.
- **Comprehensive Coverage**: The test suite achieves extensive coverage, including statement, decision, condition, and path coverage. This ensures that the majority of the code and scenarios are evaluated, significantly reducing the likelihood of undetected bugs.
- **Functional Integrity**: The game's core functionalities, such as tile generation, movement, merging, and win/loss detection, are rigorously tested. This ensures that the game behaves as expected under various conditions.
- **Effective Use of Mock Objects**: Mock object testing proved effective in isolating and testing specific functionalities like tile addition and user input handling, ensuring these components function correctly without the unpredictability of actual gameplay.
- **Loop and Conditional Logic Testing**: Loop testing and condition coverage were particularly important in validating the game's logic in repetitive processes and decision-making scenarios, ensuring consistent game behavior.

Implications

- **User Experience**: The thorough testing ensures a smooth and bug-free user experience, enhancing player engagement and satisfaction.
- **Maintainability**: The comprehensive test suite serves as a solid foundation for future development and maintenance of the game, allowing for easier identification and rectification of issues as the game evolves.
- **Confidence in Deployment**: The extensive testing provides confidence in the game's stability and performance, crucial for its successful deployment and adoption.

The game’s testing regimen has demonstrated that a well-planned and executed testing strategy is critical for the development of high-quality software. The methodologies applied here ensure that the game is not only functional and reliable but also provides a foundation for future enhancements and a positive user experience.