Minesweeper Test Cases Documentation

Overview

This document describes the unit test suite developed for the Minesweeper game implemented in `minesweeper.py` for the EECS 581 Project 1. The test suite, written using Python's `unittest` framework, verifies the correctness of the game's core components: the `Cell`, `Board`, `Game`, and `Audio` classes, as well as the CLI input helper function `get_val`. The tests are designed to run in a headless environment by mocking Pygame dependencies, ensuring no display or audio device is required, and avoiding dependencies on external sound files.

The test suite focuses on:

- Verifying core game logic (mine placement, cell revealing, win/loss conditions).
- Testing AI solver behavior across difficulty levels (easy, medium, hard).
- Ensuring robust input handling and audio functionality.
- Covering edge cases, such as invalid inputs and first-click safety.

The tests are implemented in `test_minesweeper.py` and can be executed with:

python -m unittest test_minesweeper.py

Test Suite Structure

The test suite is organized into test cases grouped by the class or function they target:

1. **Cell Class Tests**: Verify cell state initialization and rendering behavior.
2. **Board Class Tests**: Test grid initialization, mine placement, reveal logic, and AI strategies.
3. **Game Class Tests**: Validate game initialization, state management, event handling, and win conditions.
4. **Audio Class Tests**: Ensure sound loading and playback work correctly, with fallback for audio failures.
5. **CLI Helper Tests**: Confirm robust input validation for console prompts.

Key Features of the Test Suite

- **Pygame Mocking**: A `MockPygame` class simulates Pygame's display, font, mixer, and event modules to avoid requiring a graphical or audio environment.

- **Reproducible Results**: Uses `random.seed(42)` to ensure consistent mine placement and AI moves.
- **Comprehensive Coverage**: Tests cover core functionality, edge cases, and error handling.
- **Headless Execution**: No external dependencies (e.g., sound files) are required.

Test Case Details

Below is a detailed breakdown of each test case, including its purpose, what it verifies, and why it's important.

1. Cell Class Tests

The `Cell` class represents a single grid square with state (mine, revealed, flagged) and rendering logic.

- **`test_cell_initial_state`**
    - **Purpose**: Verifies that a `Cell` object is initialized correctly.
    - **What It Tests**: Checks the row, column, mine status, revealed status, flagged status, adjacent mine count, pixel coordinates, and border state of a newly created cell at (0,0).
    - **Expected Outcome**: All properties are set as expected (e.g., `is_mine=False`, `x=40`, `y=140` due to `LABEL_AREA_SIZE` and `HEADER_HEIGHT`).
    - **Why It's Important**: Ensures cells start in a consistent state, critical for grid setup and gameplay logic.
- **`test_cell_draw_covered`**
    - **Purpose**: Tests rendering a covered cell.
    - **What It Tests**: Calls `draw` on a covered cell and verifies that no text is rendered (since it's not revealed or flagged).
    - **Expected Outcome**: No calls to `screen.blit` for text rendering.
    - **Why It's Important**: Confirms that covered cells are rendered correctly without exposing hidden information.
- **`test_cell_draw_revealed_mine`**
    - **Purpose**: Tests rendering a revealed mine cell.
    - **What It Tests**: Sets `is_mine=True` and `is_revealed=True`, then calls `draw` and checks that no text is rendered (mines don't show numbers).
    - **Expected Outcome**: No calls to `font.render`, and border is not set.
    - **Why It's Important**: Ensures mines are visually distinct when revealed (e.g., on game loss).

- **`test_cell_draw_revealed_number`**
    - **Purpose**: Tests rendering a revealed cell with an adjacent mine count.
    - **What It Tests**: Sets `is_revealed=True` and `adjacent_mines=3`, then verifies that `font.render` is called with the correct number and color (`COLOR_NUMBERS[3]`).
    - **Expected Outcome**: `font.render("3", True, (206, 145, 120))` is called.
    - **Why It's Important**: Confirms that adjacent mine counts are displayed with the correct color-coded styling.
- **`test_cell_draw_flagged`**
    - **Purpose**: Tests rendering a flagged cell.
    - **What It Tests**: Sets `is_flagged=True` and calls `draw`, ensuring no text is rendered (flags don't show numbers).
    - **Expected Outcome**: No calls to `screen.blit` for text.
    - **Why It's Important**: Verifies that flagged cells are rendered correctly without exposing underlying state.

2. Board Class Tests

The `Board` class manages the 10x10 grid, mine placement, reveal logic, and AI strategies.

- **`test_board_initialization`**
    - **Purpose**: Verifies correct initialization of the `Board` class.
    - **What It Tests**: Creates a board with 10 mines and "easy" difficulty, checking the number of mines, AI strategy, last cell (None), and grid structure.
    - **Expected Outcome**: `num_mines=10`, `uncover_cell` is the decorated easy strategy, `last_cell=None`, and grid is a 10x10 array of `Cell` objects.
    - **Why It's Important**: Ensures the board is set up correctly for gameplay and AI integration.
- **`test_place_mines_first_click_safety`**
    - **Purpose**: Tests mine placement with first-click safety.
    - **What It Tests**: Places 10 mines with a first click at (5,5) and verifies that the 3x3 safe zone around (5,5) contains no mines and the total mine count is 10.
    - **Expected Outcome**: Exactly 10 mines, none in the safe zone [(4,4), (4,5), ..., (6,6)].
    - **Why It's Important**: Validates the standard Minesweeper UX where the first click is guaranteed safe.

- `test_count_adjacent_mines`
    - **Purpose**: Verifies counting of adjacent mines.
    - **What It Tests**: Sets mines at (5,5) and (5,6), then checks the adjacent mine count for (4,5).
    - **Expected Outcome**: Returns 2 (mines at (5,5) and (5,6)).
    - **Why It's Important**: Ensures accurate mine counts for gameplay and AI decisions.
- `test_reveal_cell_single`
    - **Purpose**: Tests revealing a single cell with non-zero adjacent mines.
    - **What It Tests**: Sets `adjacent_mines=2` for (5,5), reveals it, and checks that only (5,5) is revealed.
    - **Expected Outcome**: (5,5) is revealed; adjacent cells (e.g., (5,4)) remain hidden.
    - **Why It's Important**: Confirms that non-zero cells don't trigger flood reveals.
- `test_reveal_cell_flood`
    - **Purpose**: Tests flood reveal for a zero-adjacent cell.
    - **What It Tests**: Reveals (5,5) with no adjacent mines, checking that the 3x3 region around it is revealed.
    - **Expected Outcome**: All cells in [(4,4), (4,5), ..., (6,6)] are revealed.
    - **Why It's Important**: Verifies recursive flood reveal, a core Minesweeper mechanic.
- `test_reveal_all_mines`
    - **Purpose**: Tests revealing all mines on game loss.
    - **What It Tests**: Sets a mine at (5,5), calls `reveal_all_mines`, and checks that only the mine is revealed.
    - **Expected Outcome**: (5,5) is revealed; non-mine cells (e.g., (0,0)) remain hidden.
    - **Why It's Important**: Ensures mines are shown correctly when the game ends in a loss.
- `test_neighbors`
    - **Purpose**: Verifies the neighbor generator for a cell.
    - **What It Tests**: Gets neighbors of (5,5) and checks their coordinates.
    - **Expected Outcome**: Returns 8 cells: [(4,4), (4,5), (4,6), (5,4), (5,6), (6,4), (6,5), (6,6)].
    - **Why It's Important**: Neighbor iteration is critical for mine counting and AI strategies.
- `test_uncover_cell_easy`
    - **Purpose**: Tests the easy AI strategy.

    - **What It Tests**: Calls `uncover_cell` with `is_first=True`, checking that a random unrevealed, unflagged cell is chosen and highlighted.
    - **Expected Outcome**: Selected cell has `border=True`, is not revealed, and is tracked as `last_cell`.
    - **Why It's Important**: Validates the baseline AI behavior for random cell selection.
- **`test_uncover_cell_medium_safe`**
    - **Purpose**: Tests the medium AI strategy for revealing safe cells.
    - **What It Tests**: Sets up a revealed cell (5,5) with `adjacent_mines=1` and a flagged neighbor (5,6), then checks that a safe cell is revealed.
    - **Expected Outcome**: A non-flagged, safe cell is revealed with `border=True`.
    - **Why It's Important**: Ensures the medium AI uses simple constraints to identify safe cells.
- **`test_uncover_cell_medium_flag`**
    - **Purpose**: Tests the medium AI strategy for flagging mines.
    - **What It Tests**: Sets up a revealed cell (5,5) with `adjacent_mines=1` and a mine at (5,6), then checks that (5,6) is flagged.
    - **Expected Outcome**: (5,6) is flagged with `border=True` and not revealed.
    - **Why It's Important**: Verifies that the medium AI correctly flags mines when constraints are met.
- **`test_uncover_cell_hard`**
    - **Purpose**: Tests the hard AI strategy.
    - **What It Tests**: Calls `uncover_cell` and checks that a non-mine, unrevealed cell is chosen and revealed.
    - **Expected Outcome**: Selected cell is non-mine, has `border=True`, and is revealed.
    - **Why It's Important**: Confirms the simplified hard AI avoids mines (placeholder for a full solver).

3. Game Class Tests

The `Game` class orchestrates the game loop, event handling, and rendering.

- **`test_game_initialization`**
    - **Purpose**: Verifies correct game initialization.
    - **What It Tests**: Creates a game with 10 mines, "easy" difficulty, and interactive mode, checking properties and Pygame setup.

    - **Expected Outcome**: Correct `num_mines`, `difficulty`, `is_interactive`, and calls to `pygame.display.set_mode` and `set_caption`.
    - **Why It's Important**: Ensures the game starts in a valid state with proper UI setup.
- **`test_reset_game`**

    - **Purpose**: Tests game reset functionality.
    - **What It Tests**: Sets `game_over=True` and `win=True`, resets the game, and checks the new state and `last_game_status`.
    - **Expected Outcome**: `last_game_status="Victory!"`, `game_over=False`, `win=False`, `first_click=True`, `flags_placed=0`.
    - **Why It's Important**: Confirms that resetting restores the initial state and tracks prior results.
- **`test_check_win_condition`**

    - **Purpose**: Tests the win condition check.
    - **What It Tests**: Reveals all non-mine cells (with one mine at (0,0)) and checks `game_over` and `win`.
    - **Expected Outcome**: `game_over=True`, `win=True`.
    - **Why It's Important**: Verifies the win condition (all non-mine cells revealed) works correctly.
- **`test_handle_events_restart`**

    - **Purpose**: Tests the restart button event.
    - **What It Tests**: Simulates a mouse click on the restart button and checks that the game resets and plays the "restart" sound.
    - **Expected Outcome**: `first_click=True`, `flags_placed=0`, and `Audio.play("restart")` called.
    - **Why It's Important**: Ensures the restart button resets the game state and triggers the correct sound.
- **`test_handle_events_left_click`**

    - **Purpose**: Tests left-click event handling.
    - **What It Tests**: Simulates a left click on cell (1,1), checking that it's revealed and the "click" sound plays.
    - **Expected Outcome**: Cell (1,1) is revealed, `first_click=False`, and `Audio.play("click")` called.
    - **Why It's Important**: Validates core interaction for revealing cells.
- **`test_handle_events_right_click`**

    - **Purpose**: Tests right-click event handling.

    - **What It Tests**: Simulates a right click on cell (1,1), checking that it's flagged and the "flag" sound plays.
    - **Expected Outcome**: Cell (1,1) is flagged, `flags_placed=1`, and `Audio.play("flag")` called.
    - **Why It's Important**: Confirms flagging functionality and sound integration.

4. Audio Class Tests

The `Audio` class manages sound effects for game events.

- **`test_audio_init_success`**
    - **Purpose**: Tests successful audio initialization.
    - **What It Tests**: Initializes `Audio` and checks that Pygame mixer is set up and sounds are loaded.
    - **Expected Outcome**: `enabled=True`, 6 sounds loaded, and `pygame.mixer.pre_init(44100, -16, 2, 256)` called.
    - **Why It's Important**: Ensures audio initializes correctly when a device is available.
- **`test_audio_init_failure`**
    - **Purpose**: Tests audio initialization failure handling.
    - **What It Tests**: Simulates a mixer initialization failure and checks that audio is disabled gracefully.
    - **Expected Outcome**: `enabled=False`, all sounds are `None`.
    - **Why It's Important**: Verifies the game remains playable without audio.
- **`test_audio_play_enabled`**
    - **Purpose**: Tests sound playback when audio is enabled.
    - **What It Tests**: Plays the "click" sound and checks that the mocked sound's `play` method is called.
    - **Expected Outcome**: `sound.play()` called once.
    - **Why It's Important**: Confirms that sound effects trigger correctly.
- **`test_audio_play_disabled`**
    - **Purpose**: Tests sound playback when audio is disabled.
    - **What It Tests**: Disables audio and attempts to play a sound, ensuring no errors occur.
    - **Expected Outcome**: No errors, no `play` calls.
    - **Why It's Important**: Ensures robustness when audio is unavailable.

5. CLI Helper Tests

The `get_val` function handles console input validation.

- **`test_get_val_valid_input`**
    - **Purpose**: Tests valid input handling.
    - **What It Tests**: Simulates input “15” with a 10–20 range validator, checking the returned value.
    - **Expected Outcome**: Returns 15.
    - **Why It’s Important**: Confirms that valid inputs are processed correctly.
- **`test_get_val_invalid_then_valid`**
    - **Purpose**: Tests handling of invalid followed by valid input.
    - **What It Tests**: Simulates inputs “5” (invalid) then “15” (valid) with a 10–20 range validator.
    - **Expected Outcome**: Returns 15 after rejecting “5”.
    - **Why It’s Important**: Ensures robust error handling for out-of-range inputs.
- **`test_get_val_value_error`**
    - **Purpose**: Tests handling of non-numeric input.
    - **What It Tests**: Simulates inputs “invalid” then “15” with an `int` cast.
    - **Expected Outcome**: Returns 15 after rejecting “invalid”.
    - **Why It’s Important**: Verifies that invalid inputs (e.g., non-numeric) are handled gracefully.

Running the Tests

To run the test suite:

1. Save `test_minesweeper.py` in the same directory as `minesweeper.py`.
2. Ensure Python’s `unittest` module is available (included in the standard library).

Run the command:
python -m unittest test_minesweeper.py

3.
4. The tests use a mocked Pygame environment, so no display, audio device, or sound files are required.

Notes

- **Reproducibility**: The tests set `random.seed(42)` for consistent mine placement and AI moves.

- **Dependencies**: The test suite requires only `unittest` and the `unittest.mock` module; no external Pygame installation is needed due to mocking.
- **Extensibility**: To add tests for new features (e.g., a timer or advanced AI), extend the test class with new methods targeting the added functionality.
- **Limitations**: The tests focus on unit-level functionality. Integration tests (e.g., full game simulation) or performance tests could be added for broader coverage.

Conclusion

The test suite provides thorough coverage of the Minesweeper game's core logic, UI interactions, AI behavior, and error handling. It ensures the game functions as intended while being robust against edge cases and audio failures. If additional features are added to `minesweeper.py`, corresponding test cases can be developed to maintain reliability.

For further assistance, contact the maintainers (Aniketh and Yaeesh) or refer to the original project authors (Asa Maker and Zach Sevart).

```python
import unittest
from unittest.mock import patch, Mock
import random
from minesweeper import Cell, Board, Game, Audio, get_val

# Mock Pygame to avoid requiring a display or audio device
class MockPygame:
    class Rect:
        def __init__(self, x, y, w, h, border_radius=0):
            self.x, self.y, self.w, self.h = x, y, w, h
            self.border_radius = border_radius
        def collidepoint(self, pos):
            x, y = pos
            return self.x <= x < self.x + self.w and self.y <= y < self.y + self.h
        @property
        def center(self):
            return (self.x + self.w // 2, self.y + self.h // 2)

    class Font:
        def __init__(self, *args):
            pass
        def render(self, text, antialias, color):
            return Mock(text=text, color=color)

    class Surface:
        def __init__(self, size):
```

```
            pass
        def fill(self, color):
            pass
        def blit(self, surface, pos):
            pass

    def __init__(self):
        self.Rect = self.Rect
        self.USEREVENT = 32768
        self.QUIT = 256
        self.MOUSEBUTTONDOWN = 1025

pygame = MockPygame()
pygame.display = Mock(set_mode=Mock(return_value=pygame.Surface((0, 0))),
                      set_caption=Mock())
pygame.font = Mock(Font=pygame.Font, init=Mock())
pygame.mixer = Mock(pre_init=Mock(), init=Mock(), Sound=Mock())
pygame.event = Mock(get=Mock(return_value=[]))
pygame.time = Mock(set_timer=Mock())

# Patch the global pygame module in the minesweeper module
@patch('minesweeper.pygame', pygame)
class TestMinesweeper(unittest.TestCase):
    def setUp(self):
        # Reset random seed for reproducible mine placement
        random.seed(42)
        # Reset mocks
        pygame.display.reset_mock()
        pygame.font.reset_mock()
        pygame.mixer.reset_mock()
        pygame.event.reset_mock()
        pygame.time.reset_mock()

    # --- Cell Class Tests ---
    def test_cell_initial_state(self):
        cell = Cell(0, 0)
        self.assertEqual(cell.row, 0)
        self.assertEqual(cell.col, 0)
        self.assertFalse(cell.is_mine)
        self.assertFalse(cell.is_revealed)
        self.assertFalse(cell.is_flagged)
        self.assertEqual(cell.adjacent_mines, 0)
        self.assertEqual(cell.x, 40)  # LABEL_AREA_SIZE
        self.assertEqual(cell.y, 140)  # HEADER_HEIGHT + LABEL_AREA_SIZE
```

```
        self.assertFalse(cell.border)

    def test_cell_draw_covered(self):
        cell = Cell(0, 0)
        screen = Mock()
        font = Mock()
        cell.draw(screen, font)
        screen.fill.assert_not_called()  # No fill for individual cell
        screen.blit.assert_not_called()  # No text for covered cell

    def test_cell_draw_revealed_mine(self):
        cell = Cell(0, 0)
        cell.is_mine = True
        cell.is_revealed = True
        screen = Mock()
        font = Mock()
        cell.draw(screen, font)
        screen.blit.assert_not_called()  # No text for mine
        self.assertTrue(cell.border is False)

    def test_cell_draw_revealed_number(self):
        cell = Cell(0, 0)
        cell.is_revealed = True
        cell.adjacent_mines = 3
        screen = Mock()
        font = Mock()
        cell.draw(screen, font)
        font.render.assert_called_with("3", True, (206, 145, 120))  # COLOR_NUMBERS[3]

    def test_cell_draw_flagged(self):
        cell = Cell(0, 0)
        cell.is_flagged = True
        screen = Mock()
        font = Mock()
        cell.draw(screen, font)
        screen.blit.assert_not_called()  # No text for flagged cell

    # --- Board Class Tests ---
    def test_board_initialization(self):
        board = Board(num_mines=10, difficulty="easy")
        self.assertEqual(board.num_mines, 10)
        self.assertEqual(board.uncover_cell.__name__, "wrapper")  # Decorated easy strategy
        self.assertIsNone(board.last_cell)
        self.assertEqual(len(board.grid), 10)
```

```
        self.assertEqual(len(board.grid[0]), 10)
        self.assertTrue(all(isinstance(cell, Cell) for row in board.grid for cell in row))

    def test_place_mines_first_click_safety(self):
        board = Board(num_mines=10, difficulty="easy")
        board.place_mines(5, 5)
        safe_zone = [(r, c) for r in range(4, 7) for c in range(4, 7)]
        mine_count = sum(1 for row in board.grid for cell in row if cell.is_mine)
        self.assertEqual(mine_count, 10)
        for r, c in safe_zone:
            self.assertFalse(board.grid[r][c].is_mine, f"Cell ({r}, {c}) should be safe")

    def test_count_adjacent_mines(self):
        board = Board(num_mines=10, difficulty="easy")
        board.grid[5][5].is_mine = True
        board.grid[5][6].is_mine = True
        count = board.count_adjacent_mines(4, 5)
        self.assertEqual(count, 2)

    def test_reveal_cell_single(self):
        board = Board(num_mines=10, difficulty="easy")
        board.grid[5][5].adjacent_mines = 2
        board.reveal_cell(5, 5)
        self.assertTrue(board.grid[5][5].is_revealed)
        self.assertFalse(board.grid[5][4].is_revealed)  # No flood for non-zero

    def test_reveal_cell_flood(self):
        board = Board(num_mines=10, difficulty="easy")
        board.reveal_cell(5, 5)
        self.assertTrue(board.grid[5][5].is_revealed)
        for r in range(4, 7):
            for c in range(4, 7):
                self.assertTrue(board.grid[r][c].is_revealed, f"Cell ({r}, {c}) should be revealed")

    def test_reveal_all_mines(self):
        board = Board(num_mines=10, difficulty="easy")
        board.grid[5][5].is_mine = True
        board.reveal_all_mines()
        self.assertTrue(board.grid[5][5].is_revealed)
        self.assertFalse(board.grid[0][0].is_revealed)  # Non-mine stays hidden

    def test_neighbors(self):
        board = Board(num_mines=10, difficulty="easy")
        cell = board.grid[5][5]
```

```
        neighbors = list(board.neighbors(cell))
        self.assertEqual(len(neighbors), 8)
        expected_coords = [(r, c) for r in range(4, 7) for c in range(4, 7) if (r, c) != (5, 5)]
        for neighbor in neighbors:
            self.assertIn((neighbor.row, neighbor.col), expected_coords)

    def test_uncover_cell_easy(self):
        board = Board(num_mines=10, difficulty="easy")
        random.seed(42)
        cell = board.uncover_cell(is_first=True)
        self.assertTrue(isinstance(cell, Cell))
        self.assertTrue(cell.border)
        self.assertIs(board.last_cell, cell)
        self.assertFalse(cell.is_revealed)  # is_first=True prevents reveal

    def test_uncover_cell_medium_safe(self):
        board = Board(num_mines=10, difficulty="medium")
        board.grid[5][5].is_revealed = True
        board.grid[5][5].adjacent_mines = 1
        board.grid[5][6].is_flagged = True  # Mine flagged
        cell = board.uncover_cell()
        self.assertFalse(cell.is_flagged)
        self.assertTrue(cell.border)
        self.assertTrue(cell.is_revealed)  # Safe cell revealed

    def test_uncover_cell_medium_flag(self):
        board = Board(num_mines=10, difficulty="medium")
        board.grid[5][5].is_revealed = True
        board.grid[5][5].adjacent_mines = 1
        board.grid[5][6].is_mine = True  # Only one neighbor is a mine
        cell = board.uncover_cell()
        self.assertTrue(cell.is_flagged)  # Should flag the mine
        self.assertTrue(cell.border)
        self.assertFalse(cell.is_revealed)

    def test_uncover_cell_hard(self):
        board = Board(num_mines=10, difficulty="hard")
        random.seed(42)
        cell = board.uncover_cell()
        self.assertFalse(cell.is_mine)
        self.assertTrue(cell.border)
        self.assertTrue(cell.is_revealed)

    # --- Game Class Tests ---
```

```
    @patch('minesweeper.Audio')
    def test_game_initialization(self, mock_audio):
        game = Game(num_mines=10, difficulty="easy", is_interactive=True)
        self.assertEqual(game.num_mines, 10)
        self.assertEqual(game.difficulty, "easy")
        self.assertTrue(game.is_interactive)
        self.assertTrue(game.running)
        self.assertFalse(game.game_over)
        self.assertFalse(game.win)
        self.assertTrue(game.first_click)
        self.assertEqual(game.flags_placed, 0)
        pygame.display.set_mode.assert_called_with((540, 640))
        pygame.display.set_caption.assert_called_with("EECS 581 Minesweeper")

    def test_reset_game(self):
        game = Game(num_mines=10, difficulty="easy", is_interactive=True)
        game.game_over = True
        game.win = True
        game.reset_game()
        self.assertEqual(game.last_game_status, "Victory!")
        self.assertFalse(game.game_over)
        self.assertFalse(game.win)
        self.assertTrue(game.first_click)
        self.assertEqual(game.flags_placed, 0)

    def test_check_win_condition(self):
        game = Game(num_mines=10, difficulty="easy", is_interactive=True)
        for r in range(10):
            for c in range(10):
                game.board.grid[r][c].is_revealed = True  # Reveal all cells
        game.board.grid[0][0].is_mine = True  # One mine
        game.check_win_condition()
        self.assertTrue(game.game_over)
        self.assertTrue(game.win)

    @patch('minesweeper.Audio')
    def test_handle_events_restart(self, mock_audio):
        game = Game(num_mines=10, difficulty="easy", is_interactive=True)
        mock_event = Mock(type=pygame.MOUSEBUTTONDOWN, pos=(500, 50))  # Restart
button
        pygame.event.get.return_value = [mock_event]
        game.handle_events()
        mock_audio.return_value.play.assert_called_with("restart")
        self.assertTrue(game.first_click)
```

```
        self.assertEqual(game.flags_placed, 0)

    @patch('minesweeper.Audio')
    def test_handle_events_left_click(self, mock_audio):
        game = Game(num_mines=10, difficulty="easy", is_interactive=True)
        mock_event = Mock(type=pygame.MOUSEBUTTONDOWN, button=1, pos=(100, 200))  #
Cell (1, 1)
        pygame.event.get.return_value = [mock_event]
        game.handle_events()
        self.assertFalse(game.first_click)
        self.assertTrue(game.board.grid[1][1].is_revealed)
        mock_audio.return_value.play.assert_called_with("click")

    @patch('minesweeper.Audio')
    def test_handle_events_right_click(self, mock_audio):
        game = Game(num_mines=10, difficulty="easy", is_interactive=True)
        mock_event = Mock(type=pygame.MOUSEBUTTONDOWN, button=3, pos=(100, 200))  #
Cell (1, 1)
        pygame.event.get.return_value = [mock_event]
        game.handle_events()
        self.assertTrue(game.board.grid[1][1].is_flagged)
        self.assertEqual(game.flags_placed, 1)
        mock_audio.return_value.play.assert_called_with("flag")

    # --- Audio Class Tests ---
    def test_audio_init_success(self):
        audio = Audio()
        pygame.mixer.pre_init.assert_called_with(44100, -16, 2, 256)
        pygame.mixer.init.assert_called()
        self.assertTrue(audio.enabled)
        self.assertEqual(len(audio.sounds), 6)

    @patch('minesweeper.pygame.mixer.init', side_effect=Exception("No audio device"))
    def test_audio_init_failure(self):
        audio = Audio()
        self.assertFalse(audio.enabled)
        self.assertEqual(len(audio.sounds), 6)
        self.assertTrue(all(snd is None for snd in audio.sounds.values()))

    def test_audio_play_enabled(self):
        audio = Audio()
        mock_sound = Mock()
        audio.sounds["click"] = mock_sound
        audio.play("click")
```

```
        mock_sound.play.assert_called_once()

    def test_audio_play_disabled(self):
        audio = Audio()
        audio.enabled = False
        audio.play("click")
        # No assertion needed; just verify no errors

    # --- CLI Helper Tests ---
    @patch('builtins.input', side_effect=["15"])
    def test_get_val_valid_input(self, mock_input):
        result = get_val("Enter a number: ", int, validate=lambda x: 10 <= x <= 20)
        self.assertEqual(result, 15)

    @patch('builtins.input', side_effect=["5", "15"])
    def test_get_val_invalid_then_valid(self, mock_input):
        result = get_val("Enter a number: ", int, validate=lambda x: 10 <= x <= 20,
                         error="Must be 10–20")
        self.assertEqual(result, 15)

    @patch('builtins.input', side_effect=["invalid", "15"])
    def test_get_val_value_error(self, mock_input):
        result = get_val("Enter a number: ", int, validate=lambda x: 10 <= x <= 20)
        self.assertEqual(result, 15)

if __name__ == '__main__':
    unittest.main()
```