Software Testing: Homework Assignment (Further Work & Bonus)

Title: Test-Driven Development of a Sudoku Application

Objective:

This assignment is designed to extend your understanding and application of Test-Driven Development (TDD). Building upon the principles practiced in the initial Calc exercise, you will apply the TDD methodology to a more complex, logic-based problem: creating a Sudoku solver and generator. The goal is to practice breaking down a larger problem into small, verifiable steps, writing tests before implementation, and using those tests to guide your design.

Background:

Sudoku is a logic puzzle played on a 9x9 grid. The rules are simple, yet the problem space is rich, making it an excellent candidate for TDD. The core rules are [1]:

1. Each row must contain the digits 1 through 9, without repetition.
2. Each column must contain the digits 1 through 9, without repetition.
3. Each of the nine 3x3 subgrids (boxes) must contain the digits 1 through 9, without repetition.

A "proper" Sudoku puzzle has one and only one unique solution, which can be reached through logical deduction.[4]

Part 1: Core Sudoku Engine (Required for Bonus)

Your primary task is to develop a core Sudoku engine by strictly following the TDD cycle (Red -> Green -> Refactor) for each feature.

Task 1.1: The Rules Engine
The foundation of any Sudoku application is the ability to validate a move.

- 🔴 **Red:** Write a failing test for a method or function, isValid(board, row, col, num). Your test suite should include cases for a valid placement, a row conflict, a column conflict, and a 3x3 box conflict.[5]
- 🟢 **Green:** Implement the isValid logic with the minimum amount of code required to make your tests pass.

- 🔵 **Refactor:** Review and clean up your implementation and test code.

Task 1.2: The Backtracking Solver
A solver is essential for both playing the game and generating new puzzles. We will use a backtracking algorithm.

- 🔴 **Red:** Write a failing test for a solve(board) method. This test should provide an incomplete but valid puzzle and assert that your method returns true and that the board is modified to match the known correct solution.[6]
- 🟢 **Green:** Implement the backtracking algorithm. This will likely require a helper function to find empty cells. Your implementation should use the isValid method you created in the previous step.
- 🔵 **Refactor:** Ensure your solver is clean and efficient.

Task 1.3: The Puzzle Generator
A generator creates new puzzles for the user. The simplest approach is to generate a full solution and then remove numbers.

- 🔴 **Red:** Write a failing test for a generate(difficulty) function, where difficulty is the number of cells to remove. The test should verify two things:
  1. The returned puzzle board has the correct number of empty cells.
  2. The puzzle is still solvable (you can use your solve method to verify this).
- 🟢 **Green:** To generate different puzzles each time, you will first need to introduce randomness into your solver to create a random, complete board. Then, implement the logic to remove the specified number of cells.
- 🔵 **Refactor:** Review your generation logic for clarity and efficiency.

Part 2: Advanced Features (Further Bonus Work)

For additional credit, implement one or more of the following advanced features, again using the TDD methodology for each.

Bonus 2.1: Uniqueness Guarantee in Generator
A high-quality Sudoku puzzle must have a single, unique solution.4

- **Task:** Refactor your generator. To do this, you must first create a solver that can **count all possible solutions**, not just find the first one. Then, in your generator, after removing each number, use this new solver to verify that the number of solutions remains exactly one. If removing a number results in zero or multiple solutions, you must revert the change.[8]
- **TDD Approach:** Write tests that provide puzzles with known multiple solutions and no solutions, and assert that your solution-counting solver returns the correct count. Then, write a test to ensure your generator produces a puzzle that your solver confirms has

only one solution.

Bonus 2.2: Difficulty Rating Engine
Puzzle difficulty is not determined by the number of given clues but by the complexity of the logical techniques required to solve it.9

- **Task:** Implement a basic difficulty rating engine. Your engine should solve a puzzle by simulating human techniques. A simple approach is to assign a score based on the hardest technique required. For example:
  - **Easy:** Solvable using only "Naked Singles" and "Hidden Singles."
  - **Medium:** Requires "Naked/Hidden Pairs" or "Pointing Pairs."
  - **Hard:** Requires advanced techniques like "X-Wings" or "Swordfish."
- **TDD Approach:** Create tests with known puzzles that require specific techniques. Assert that your rating engine correctly categorizes them.

Bonus 2.3: Robust Error & Edge Case Handling
Your application should handle invalid inputs gracefully.

- **Task:** Implement checks to handle invalid puzzle states.
- **TDD Approach:** Write tests for various edge cases [11]:
  - A puzzle that is unsolvable from its initial state.
  - A puzzle that has invalid characters (e.g., numbers outside 1-9).
  - A puzzle whose initial "given" numbers already violate Sudoku rules (e.g., a duplicate in a row).
    Assert that your program identifies these puzzles as invalid rather than crashing or producing incorrect output.

Submission Requirements:

As with the Calc assignment, you must submit the following:

1. **Source Code:** All your final source code files (e.g., Sudoku.java, sudoku.py, sudoku.js).
2. **Test Files:** All corresponding test files.
3. **Screenshot of Passing Tests:** A single screenshot showing the successful execution of your entire test suite.
4. **TDD Narrative (Report):** A document (PDF or Markdown) detailing your TDD process for **each feature** you implemented (both core and bonus). For each cycle, you must describe:
   - The failing test you created and why it defined the requirement (**Red**).
   - The specific code changes you made to make the test pass (**Green**).
   - Any refactoring you performed to improve the code's design while keeping the tests green (**Refactor**).

Good luck, and enjoy the process of building a robust application guided by tests!