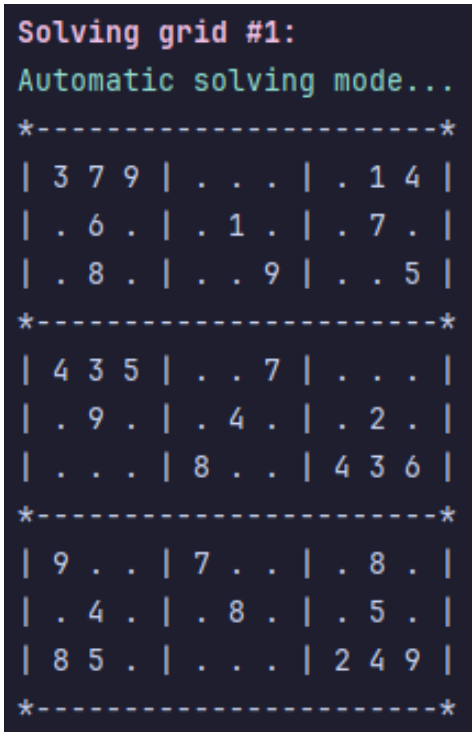
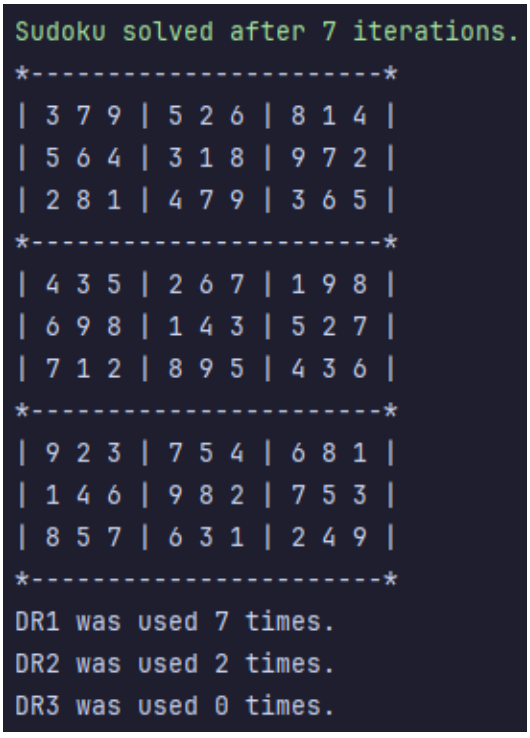
SUDOKU SOLVER

REPORT

SOFTWARE ENGINEERING

September 2024 - November 2024





KHAO Chloé

M1 Informatique

2024-2025

Summary

Introduction

This report outlines the design and implementation of a **Sudoku Solver** for the *Software Engineering Project 2024*. The solver applies three deduction rules—**DR1 (Naked Single), DR2 (Hidden Single), and DR3 (Pointing Pair)**—within a Java-based system to solve a Sudoku grid progressively. If the rules cannot fully solve the puzzle, the program switches to an interactive mode, prompting the user for input. Additionally, the solver classifies each puzzle's difficulty as **Easy, Medium, Hard**, or **Unsolvable,** based on the rules applied during the solving process.

The project aimed to create a user-friendly and flexible tool capable of handling various puzzle complexities. Key design patterns—**Singleton, Strategy, State,** and **Iterator**—were employed to build a modular, adaptable system. This report details the challenges encountered, solutions implemented, and design choices made during the development process.

Problem Description

The primary objective was to develop a flexible Sudoku-solving tool that includes the following features:

1. Accept user input in various formats (single entries, full grids, or multiple grids via file upload).
2. Solve puzzles based on three deduction rules (DR1, DR2, DR3).
3. Handle cases where puzzles remain unsolved by prompting the user for values interactively.
4. Classify puzzles into **Easy, Medium, Hard** or **Unsolvable** categories based on the rules utilized.

Solution Overview

Input Options

The solver provides three input methods:

- **Line-by-line entry**: Each row is entered separately.
- **Single full-grid entry**: The entire Sudoku grid is input at once.
- **File upload**: Multiple grids can be loaded and solved sequentially.

```
Sudoku Solver
Welcome to the Sudoku Solver!

Instructions:
Use 0 or -1 to represent empty cells.
You can enter the grid in one of the following ways, using commas to separate integers:
1. As a single line of 81 integers.
2. Row by row, with 9 integers per row.
3. From a file, with each grid on a separate line.
Type "quit" to exit.

Example for a row by row input:
Enter row 1: 5,3,-1,-1,7,0,-1,-1,-1
(For the entire grid, you would enter 81 integers separated by commas.)

Enter the entire grid, row by row, or read from a file? (E/R/F): f
Enter the file path: C:\Users\chloe\OneDrive\Desktop\FAC\UCA\Master\S1\Soft Eng\Sudoku_Solver\test\test0.txt
```

Operational Modes

The Sudoku Solver has three operational states:

- **Automatic Mode**: Deduction rules are applied to solve the grid without user interruption.
- **Interactive Mode**: If no further progress can be made with deduction rules, the solver prompts the user to enter a value in a specific cell.
- **Final State**: When the grid is fully solved, the final solution is printed.

Puzzle Classification

The difficulty level of each puzzle is classified based on the deduction rules required:

- **Easy**: Requires only DR1.
- **Medium**: Requires DR1 and DR2.
- **Hard**: Requires DR1, DR2, and DR3, plus user input.
- **Hard:** Requires DR1, DR2, and DR3 and may prompt the user for input if necessary.
- **Unsolvable:** All deduction rules (DR1, DR2, and DR3) are applied, but the user opts not to provide additional input, leaving the puzzle unsolved.

```
Total grids: 20
Solved grids: 16
Easy: 0
Medium: 13
Hard: 3
Unsolvable: 4
Percentage solved: 80.00%
```

Design Patterns

The following design patterns were implemented to create a modular and maintainable solution:

Singleton Pattern

Each deduction rule (**DR1**, **DR2**, **DR3**) follows the **Singleton** pattern. Since each rule represents a unique and independent logic for solving Sudoku puzzles, it makes sense to enforce that there is only one instance of each rule across the entire application.

The **Singleton** pattern ensures that we have exactly one instance of each rule, preventing redundant instantiations and reducing memory overhead.

This approach guarantees that the rules are consistently applied without the need for re-creating objects unnecessarily, thus streamlining the rule-checking process and optimizing performance.

Strategy Pattern

The **Strategy** pattern is used to give the solver flexibility in choosing which deduction rule (DR1, DR2, or DR3) to apply based on the state of the Sudoku grid.

Each deduction rule is treated as a separate strategy for solving part of the puzzle, allowing the program to switch between them as needed.

This design makes the solver adaptable, as it can decide at runtime which rule is best suited to the current grid, rather than following a fixed sequence of rules.

This approach helps optimize the solving process, especially when dealing with puzzles of varying difficulty levels.

State Pattern

The **State** pattern is used to manage the transitions between different operational modes of the Sudoku solver: **Automatic**, **Interactive**, and **Final State**.

Each mode has its own state class, encapsulating the specific behavior and logic required for that mode.

The use of this pattern simplifies the management of the solver's states, allowing the program to automatically switch between them as necessary.

In the **Automatic** mode, the solver applies deduction rules to solve the puzzle without user intervention.

When the solver reaches a point where it cannot proceed, it transitions to **Interactive** mode, prompting the user for input.

Once the puzzle is solved, the solver enters the **Final State**, displaying the completed grid.

Iterator Pattern

In my implementation, the **Iterator** pattern is used to traverse the cells of the Sudoku grid whenever a complete scan of the grid is necessary.

This design allows my solver to efficiently iterate over each cell, checking and updating

their values as required for the deduction rules.

By abstracting the traversal logic, the **Iterator** pattern in my code simplifies systematic evaluation, ensuring that all relevant cells are processed without complex manual loops.

This structure ensures the solver can consistently apply rules across the grid in a clean and efficient manner.

Deduction Rules

DR1 : Naked Single

DR1 (Naked Single) was the simplest deduction rule to implement.t identifies cells that have only one possible value remaining. When a cell has a single candidate, that value is immediately assigned, making it an efficient and direct method for solving simpler grids, which are often classified as Easy puzzles.

I encountered no significant challenges in implementing this rule, as its logic is straightforward and operates well in identifying and filling obvious cells.

DR2 : Hidden Single

At first, I found **DR2 (Hidden Single)** challenging to understand and implement, as it requires identifying when a value can only fit in one cell within a row, column, or 3x3 box. Unlike **DR1**, where the solution is obvious, this rule hides the solution among multiple candidates.

The complexity of checking each row, column, and box made it tricky to apply, but it's essential for solving **Medium** difficulty puzzles where simpler strategies no longer work.

DR3 : Pointing Pair

DR3 (Pointing Pair) was the most challenging rule to implement. It involves identifying situations where a candidate number can only appear in two cells in a row or column within a 3x3 box. If this condition is met, that number is eliminated from other cells in the same row or column, but outside the 3x3 box.

The complexity of this rule requires more intricate logic and often leads to the need for

manual input from the user, particularly for **Hard** puzzles.

Challenges Faced and Solutions

Understanding Deduction Rules

The main challenge was understanding the deduction rules, especially DR2 (Hidden Single) and DR3 (Pointing Pair). These rules involve complex logical patterns, which require research and testing to implement correctly.

Managing Operational Modes

Transitioning smoothly between automatic and interactive modes was crucial. The **State pattern** helped manage these transitions efficiently, ensuring the solver could prompt for user input when needed without disrupting the flow.

Cell Traversal

Traversing and updating the grid efficiently was a challenge. The **Iterator pattern** simplified grid traversal, allowing the solver to check and update cells systematically without complex nested loops.

Conclusion

The Sudoku Grid Solver successfully meets project requirements by offering flexible input options, structured deduction-based solving, and interactive assistance for complex puzzles. By applying design patterns, the system is modular, maintainable, and supports efficient rule application, state transitions, and grid traversal.

The project deepened my understanding of Sudoku-solving strategies, with the main challenge being the implementation of advanced deduction rules like DR2 (Hidden Single) and DR3 (Pointing Pair). Ultimately, the solver provides a robust solution for puzzles of varying difficulties, thanks to its modular design and effective use of design patterns.