Rapport Project Sudoku

Author : Louis Malmassary 22101183

Table of content:

Problematic

In this project, our objective was to develop an algorithm to solve Sudoku puzzles using four different design patterns. I selected Java as the programming language for this implementation. The project requirements included specific programming constraints, which I will discuss in detail later. To begin, let’s examine the approach for solving a Sudoku puzzle.

Solving a Sudoku

An example of a solved Sudoku:

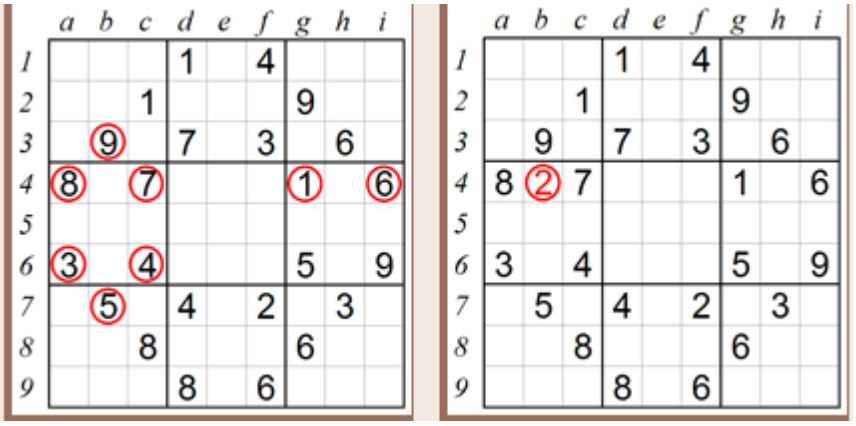
$$\begin{bmatrix} 6 & 8 & 1 & 5 & 2 & 3 & 4 & 7 & 9 \\ 5 & 4 & 2 & 7 & 6 & 9 & 8 & 1 & 3 \\ 7 & 9 & 3 & 4 & 8 & 1 & 6 & 5 & 2 \\ 9 & 3 & 6 & 7 & 2 & 8 & 5 & 4 & 1 \\ 4 & 2 & 7 & 1 & 9 & 5 & 3 & 6 & 8 \\ 1 & 5 & 8 & 3 & 4 & 6 & 9 & 2 & 7 \\ 8 & 7 & 9 & 6 & 1 & 4 & 2 & 3 & 5 \\ 3 & 1 & 4 & 9 & 5 & 2 & 7 & 8 & 6 \\ 2 & 6 & 5 & 8 & 3 & 7 & 1 & 9 & 4 \end{bmatrix}$$

In a correctly solved Sudoku puzzle, each number appears only once in every row, column, and 3×3 sub-grid.

To solve a Sudoku puzzle, we applied three primary deduction rules. If these rules did not suffice to complete the puzzle, the user would be prompted to input a number. Here are the deduction rules in detail:

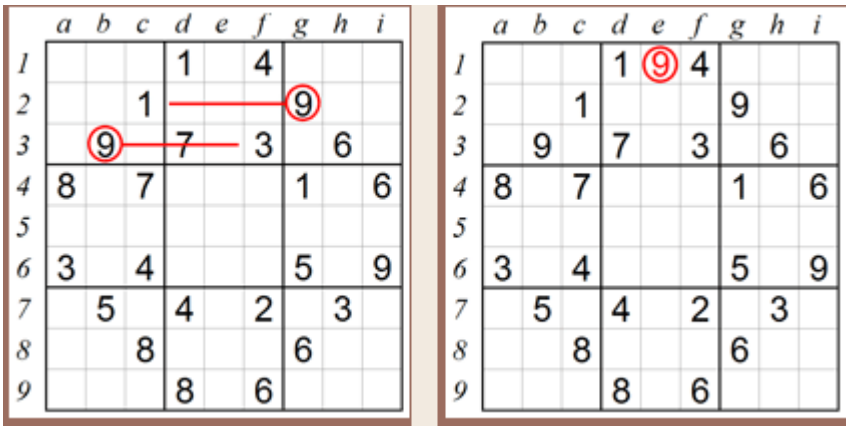
Rule 1 (Inclusion)

> Often only one number can be in a square because the remaining eight are already used in the relevant row, column and box. Taking a careful look at square b4 we can see that 3, 4, 7 and 8 are already used in the same box, 1 and 6 are used in the same row, and 5 and 9 are used in the same column. Eliminating all the above numbers leaves 2 as the single candidate for square b4.



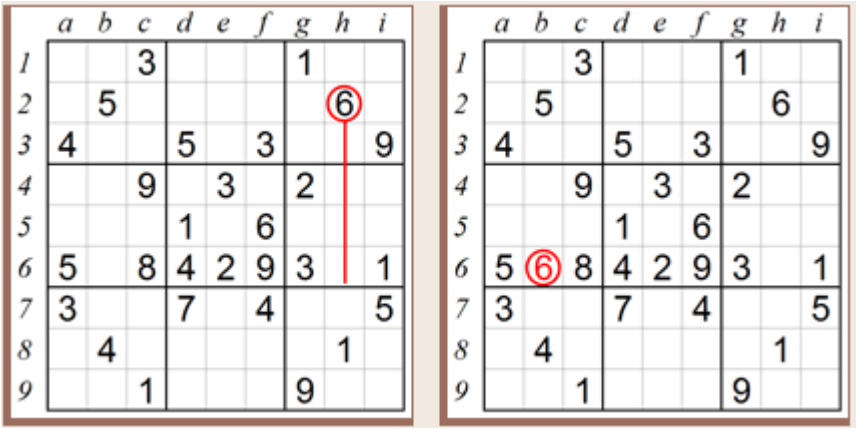
Rule 2 (Exclusion)

> In our first example we will focus on box 2, which like any other box in Sudoku must contain 9. Looking at box 1 and box 3 we can see there are already 9s in row 2 and in row 3, which excludes the two bottom rows of box 2 from having 9. This leaves square e1 as the only possible place into which 9 can fit in.



Rule 3 (Missing number)

> This method can be particularly useful when rows (and columns) are close to completion. Let's take a look at row 6. Seven of the nine squares contain the numbers 1, 2, 3, 4, 5, 8 and 9, which means that 6 and 7 are missing. However, 6 cannot be in square h6 because there is already 6 in that column. Therefore the 6 must be in square b6.



Constraint

The classes are subject to certain constraints in this project. Specifically, we are required to implement four design patterns and establish a deduction rule by extending a class named *DeductionRule*. We also need to linearise our array of the Sudoku and represent the absence of number by putting $[-1]$ and not $[0]$, i.e we won't use matrix,

---

Architecture

Classes

Main

The only things that the main class does is initialize every instance used during the whole code, then execute a toolbox class that will apply the rules and control the whole execution.

Rules

The rules packet contain every classes related to the rules describe above, every rules are derived from an abstract class named "DeductibleRule", each rule will be applied on after another in the code to determine the difficulty of said Sudoku,

Sudoku

This class contain the Sudoku, this class is Singleton which is really useful for our code as we continuously modifies it as we go trough each rule, so there is no other instance of our Sudoku, thanks to that it is not possible to have multiple modification that would override older one. This is one of the design pattern we used this project, to create a singleton look here.

Toolbox

This class is final, it's an extension of the main class which execute code in an imperative manner, we will find inside this class the function needed to transform the $\boxed{.\text{txt}}$ file into something the Sudoku class can understand. It also control the reading of the user input, and the application of the rules, it has an

internal state that edict what the course of action is. this class implement two design pattern, the first one is State and the other Adapter.

Files

A $\boxed{.txt}$ file describing the Sudoku that the program will have, in the code it transform it in an list, not a matrix

Comment

This project presented a valuable opportunity to apply theoretical concepts from software design in a hands-on context. Implementing the algorithm with constraints on both the programming approach and design patterns was challenging but ultimately rewarding. Each design pattern chosen brought distinct advantages to the structure and clarity of the code, and adapting the algorithm to include user input at certain steps allowed for flexibility in solving complex puzzles.

Throughout the project, we were able to explore the balance between modularity and performance, as well as gain deeper insights into Java's capabilities for structuring complex data interactions. This experience reinforced the importance of careful design and testing, especially when working with

Conclusion

In this project, we developed an algorithm to solve Sudoku puzzles by implementing four key design patterns and applying several logical deduction rules. This approach not only provided a systematic way to solve Sudoku puzzles but also demonstrated how design patterns can be applied effectively in practical problem-solving scenarios. By using Java, we were able to create a structured and modular solution, ensuring that each component of the program had a clear role and that the solution adhered to the established project constraints.

The Singleton pattern was particularly useful in managing the state of the Sudoku puzzle, ensuring there was only one active instance, which streamlined updates and reduced the risk of conflicting modifications. Additionally, the State and Adapter patterns allowed us to handle different stages of the program's execution and user interactions in a flexible manner. Overall, this

project emphasized the importance of design patterns in structuring complex software solutions, while highlighting the challenges and advantages of object-oriented programming for problem-solving in computational puzzles.