

# Lab 10: Sudoku Game

## Table of Contents

1. Single Sequential Mode .....	1
2. Incomplete State .....	1
3. Game Driver .....	2
3.1. Verification .....	2
3.2. Difficulty Levels .....	2
3.3. Storage .....	3
4. GUI Requirements .....	4
5. Verification, Difficulty, and Storage Flow .....	4
6. Design Guide .....	6
7. In-Game Behaviour: Verify, Solve, and Undo .....	7
8. Solver Requirements .....	8

## 1. Single Sequential Mode

From the previous lab you should have observed that multithreading did not add much to the performance. For this phase, you must remove the different modes and keep only a single sequential mode for verification, which is sufficient.

## 2. Incomplete State

We will continue to assume that the game is sanitised. However, there is now an additional state: **INCOMPLETE**. This is represented on the board by 0 zero digitated cells. For example, if we start from a valid Sudoku solution (a fully correct  $9 \times 9$  board) and remove one value (which is represented by 0) from the 81 cells, the verifier must report the state for this solution as **INCOMPLETE**.

Table 1. An example of empty sudoku board

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

**NOTE**

The GUI would **not** display it as zeros; that functionality is restricted to the application layer.

## 3. Game Driver

### 3.1. Verification

You must implement a Sudoku game driver that takes a **source solution**, which is supposed to be a valid, fully solved Sudoku board.

- The driver must **not** assume that the source solution is valid.
- It must first verify the source solution.
- If the source game is not **VALID** (i.e. it is **INVALID** or **INCOMPLETE**), you must raise an exception and propagate it to the presentation layer.

### 3.2. Difficulty Levels

You must support three difficulty levels: **Hard**, **Medium**, and **Easy**.

These are generated by removing a random number of cells from the solved board:

- Hard: remove **20** cells
- Medium: remove **25** cells
- Easy: remove **10** cells

The random positions to erase must be chosen using a seed based on the current time. The three levels **must** be generated at once.

Use **RandomPairs** class for this task.

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Random;
import java.util.Set;

public class RandomPairs {

    // Range 0..80 for both x and y
    private static final int MAX_COORD = 80;
    private static final int MAX_UNIQUE_PAIRS = (MAX_COORD + 1) * (MAX_COORD + 1);

    private final Random random;

    public RandomPairs() {
        this.random = new Random();
    }
}
```

```

}

/**
 * Generate n distinct random pairs (x, y) where 0 <= x <= 80 and 0 <= y <= 80.
 */
public List<int[]> generateDistinctPairs(int n) {
    if (n < 0 || n > MAX_UNIQUE_PAIRS) {
        throw new IllegalArgumentException(
            "n must be between 0 and " + MAX_UNIQUE_PAIRS + " (inclusive)");
    }

    Set<Integer> used = new HashSet<>();
    List<int[]> result = new ArrayList<>(n);

    while (result.size() < n) {
        int x = random.nextInt(MAX_COORD + 1); // 0..80
        int y = random.nextInt(MAX_COORD + 1); // 0..80

        // Encode pair (x, y) as a single int to track uniqueness
        int key = x * (MAX_COORD + 1) + y;

        if (used.add(key)) {
            result.add(new int[] { x, y });
        }
    }

    return result;
}
}

```

### 3.3. Storage

You must create a folder hierarchy for generated games and save each new game into one of three folders:

- **easy**
- **medium**
- **hard**

There must also be another folder that stores **only the currently played game**, regardless of its level. This is required so that, if the user closes the application and reopens it, you can ask whether they want to continue their last unfinished game.

- If a game becomes completely filled and is verified as valid, it must be removed permanently (deleted).

## 4. GUI Requirements

For this lab, you must provide a **GUI**.

At start-up, the presentation layer must call a **game catalogue** interface. This interface returns two booleans:

1. Whether there is an unfinished game.
2. Whether there exists at least one game of each difficulty level: easy, medium, and hard (there must be at least one per level).

The logic is as follows:

- If the first boolean is **true** (there is an unfinished game), the presentation layer must call the loader to load that unfinished game.
- If the first boolean is **false**, it must then check the second boolean:
  - If the second boolean is **true**, the user is asked which difficulty they prefer (**easy**, **medium**, or **hard**), and then the loader is called to load a game of that type.

The **loader** is responsible for loading a game by type:

- **easy**
- **medium**
- **hard**
- **incomplete** (if there is one)

If the user has no incomplete game and there is not at least one game per difficulty level, the user must see a screen that asks them to provide a path to a solved Sudoku game file. This solved game will then be used to generate the different difficulty levels.

## 5. Verification, Difficulty, and Storage Flow

# Sudoku - Verification, Difficulty, Storage, and GUI Flow



## 6. Design Guide

- The MVC model states that the **View**, **Controller** and **Model** should be kept separate.
- In this lab, it is required to implement two interfaces, one for each layer (presentation and application). These two interfaces must then be integrated so that the **View** can interact with the **Controller**.

```
class Catalog
{
    bool current; // True if there is a game in progress, False otherwise.
    bool allModesExist; // True if there is atleast one game available for each
    difficulty, False otherwise.
}
```

This is the interface of the controller representing the actions that are needed by the viewer

```
interface Viewable
{
    Catalog getCatalog();

    // Returns a random game with the specified difficulty
    // Note: the Game class is the representation of the soduko game in the controller
    Game getGame(DifficultyEnum level) throws NotFoundException;

    // Gets a sourceSolution and generates three levels of difficulty
    void driveGames(Game source) throws SolutionInvalidException;

    // Given a game, if invalid returns invalid and the locates the invalid duplicates
    // if valid and complete, return a value
    // if valid and incomplete, returns another value
    // The exact repersentation as a string is done as you best see fit
    String verifyGame(Game game);

    // returns the correct combination for the missing numbers
    // Hint: So, there are many ways you can approach this, one way is
    // to have a way to map an index in the combination array to its location in the
    board
    // one other way to to try to encode the location and the answer all in just one
    int
    int[] solveGame(Game game) throws InvalidGame;

    // Logs the user action
    void logUserAction(String userAction) throws IOException;
}
```

This is the interface of the viewer representing the actions that are needed by the controller

```

interface Controllable
{
    Catalog getCatalog();

    int[][] getGame(char level) throws NotFoundException;

    void driveGames(int[][] source) throws SolutionInvalidException;

    // A boolean array which says if a specific cell is correct or invalid
    bool[][] verifyGame(int[][] game);

    // contains the cell x, y and solution for each missing cell
    int[][] solveGame(int[][] game) throws InvalidGame;

    // Logs the user action
    void logUserAction(UserAction userAction) throws IOException;
}

```

You must retain the types of the parameters and return value for each interface.

The reason we use two separate interfaces is to simulate that the viewer and the controller are distinct entities, each with its own data representation and potentially different semantics.

## 7. In-Game Behaviour: Verify, Solve, and Undo

In the “happy path” where a game is successfully loaded and presented to the player, the GUI must offer two buttons:

- **Verify**
- **Solve**

The **Solve** button must be **disabled** unless the number of remaining empty cells is **exactly 5**. At that point, it should become enabled.

You must also support **Undo** functionality.

Undo requires a **log file** stored in the same folder as the currently played game. Let this folder be called **incomplete**. This folder must always be in one of two states:

- Empty, or
- Containing exactly two files: the log file and the Sudoku game file.

Logging behaviour:

- Log entries must be written immediately.
- Suppose we have an empty cell at (3, 5) and the user enters the value 3.

- The log file should be appended with a record in the form `(x, y, val, prev)`.
- In this example it would be: `(3, 5, 3, 0)`, where `prev` is the previous value (`0` in this case).

When the user performs **Undo**, you must:

- Remove the last line from the log file.
- Apply the inverse of that change to the board.

## 8. Solver Requirements

For the solver, you may **not** use any sophisticated or “smart” Sudoku-solving techniques. We do not want constraint propagation, backtracking optimisations, or any clever algorithms.

Instead, you must implement the solver using **permutations** only, with the following constraints:

- The solver is **bounded** to cases where there are exactly **5 remaining empty cells**.
- Each of the 5 positions initially has 9 possible values (`1-9`).
- This yields at most  $9^5$  possibilities (roughly 60,000 permutations).

You can think of this as verifying up to 60,000 different boards:

- For each permutation, you fill in the 5 empty cells with one of the combinations.
- For each resulting board, you verify whether it is a valid Sudoku.

You may stop as soon as you find a valid solution. On average, this is expected to require about half the checks ( $\approx 30,000$  verifications). Each verification is a relatively lightweight operation: a loop over 81 cells.

The main concern is **memory management** so that you do not run out of memory whilst handling all these permutations. You are expected to solve this using two design patterns:

- **Iterator**
- **Flyweight**

You must **not** use any of the Java concurrency keywords `synchronized`, `volatile` or thread-safe data structures at all.



# Sudoku Solver with Parallelism - Activity Diagram

