PROJECT REPORT

Subject: Sudoku Game Project

Instructor:

Student group implementation:

ITDSIU22127	Lê Trọng Hiếu
ITDSIU22166	Nguyễn Minh Đạt

Presentation content:

1.	**Multiple Game Modes** **Classic and Ice Mode variants**
2.	**Intelligent Hints** **Advanced hint system with logical techniques**
3.	**SOLID Architecture** **Clean, maintainable, and extensible design**
4.	**Efficient Algorithms** **Optimized puzzle generation and solving**

Project Overview

What we built:

- 3 Sudoku Variants in one application
- Classic Sudoku - Traditional 9×9 grid rules
- Ice Sudoku - Progressive cell unlocking mechanics

Key Features:

- Multiple difficulty levels (Easy → Expert)
- Real-time validation & scoring
- Intelligent hint system
- Auto-solve functionality

Architecture & SOLID Principles

SOLID Principles Applied:

Single Responsibility (SRP):

Each class has one clear purpose

Interface Segregation:

Small, focused interfaces

(GameActionListener, TimerListener)

Liskov Substitution (LSP):

ClassicMode and IceMode are interchangeable

Open/Closed (OCP):

Extensible through interfaces

Dependency Inversion

Depends on abstractions, not concretions

SOLID Principles in Code

Interface Segregation Principle

```java
package view;

public interface GameActionListener {
    void onCellInput(int row, int col, String value);
    void onSolve();
    void onHint();
    void onCheck();
}
public interface GameStartListener {
    void onStartGame(String mode, String difficulty);
    void onNewGame(String mode, String difficulty);
}
public interface TimerListener {
    void onTimerTick();
}
```

Benefits: Small, focused interfaces prevent classes from implementing unnecessary methods

Open/Closed Principle

```java
public interface GameMode {
    void renderCell(JTextField cell, int row, int col, Puzzle puzzle, Solution solution);
    boolean isValidMove(Puzzle puzzle, Solution solution, SudokuValidator validator, int row, int col, int value);
    boolean isPuzzleComplete(Puzzle puzzle, SudokuValidator validator);
    Point getHintCell(Puzzle puzzle, Random random);
    int checkSolution(Puzzle puzzle, Solution solution);
}
```

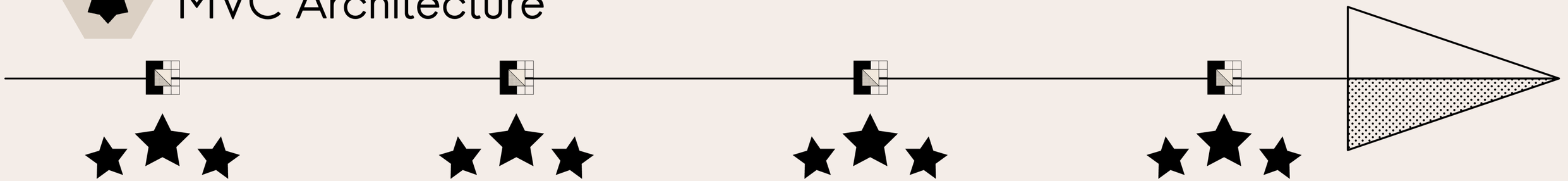
```java
public class IceMode implements GameMode {
```

```java
public class ClassicMode implements GameMode {
```

Benefits: Add new game modes without modifying existing code

Architecture & SOLID Principles

MVC Architecture



#1
Controller Layer

SudokuGameController
→ Handles game logic and coordinates between Model and View

#2
View Layer

GameView, WelcomeScreen, EndScreen, HintDialog
→ User interface and presentation logic

#3
Model Layer

Board, Puzzle, Solution
→ Data structures and business logic

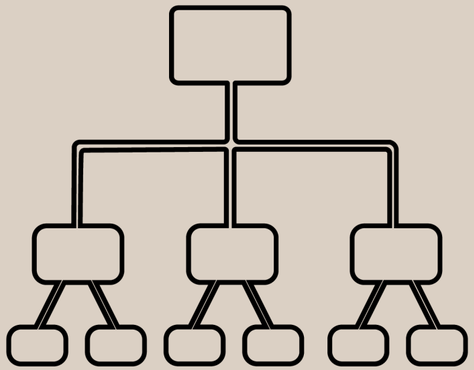
#4
Service Layer

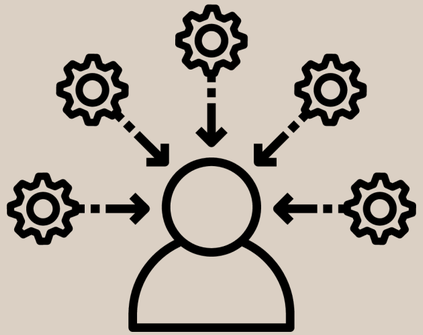
Generator, Validator, Solver, HintSystem
→ Core algorithms and game mechanics

Data Structure Implementation

Board Class - Core Data Structure

```java
public class Board {
    private static final int SIZE = 9;
    private final int[][] grid;

    public Board() {
        this.grid = new int[SIZE][SIZE];
    }

    public Board(int[][] grid) {
        this.grid = new int[SIZE][SIZE];
        for (int i = 0; i < SIZE; i++) {
            this.grid[i] = grid[i].clone();
        }
    }

    public int getCell(int row, int col) {
        return grid[row][col];
    }

    public void setCell(int row, int col, int value) {
        grid[row][col] = value;
    }

    public int[][] getGrid() {
        int[][] copy = new int[SIZE][SIZE];
        for (int i = 0; i < SIZE; i++) {
            copy[i] = grid[i].clone();
        }
        return copy;
    }

    public boolean isEmpty(int row, int col) {
        return grid[row][col] == 0;
    }
}
```





Puzzle Class - Game State

```java
public class Puzzle {
    private final Board board;
    private final boolean[][] fixedCells;
    private final boolean[][] frozenCells;

    public Puzzle(Board board, boolean[][] fixedCells, boolean[][] frozenCells) {
        this.board = board;
        this.fixedCells = fixedCells;
        this.frozenCells = frozenCells;
    }

    public Board getBoard() {
        return board;
    }

    public boolean isFixed(int row, int col) {
        return fixedCells[row][col];
    }

    public void setFixed(int row, int col, boolean fixed) {
        fixedCells[row][col] = fixed;
    }

    public boolean isFrozen(int row, int col) {
        return frozenCells != null && frozenCells[row][col];
    }

    public void setFrozen(int row, int col, boolean frozen) {
        if (frozenCells != null) {
            frozenCells[row][col] = frozen;
        }
    }
}
```

Design Choice: Immutable structure with deep copy constructor ensures data integrity

Design Choice: Separation of concerns - different cell states handled independently

Puzzle Generation Algorithm

ClassicPuzzleGenerator.java - Core Algorithm

```java
private boolean solveSudoku(Board board) {
    for (int row = 0; row < Constants.BOARD_SIZE; row++) {
        for (int col = 0; col < Constants.BOARD_SIZE; col++) {
            if (board.isEmpty(row, col)) {
                List<Integer> numbers = Arrays.asList(...a:1, 2, 3, 4, 5, 6, 7, 8, 9);
                Collections.shuffle(numbers, random);
                for (int num : numbers) {
                    if (validator.isValid(board, row, col, num)) {
                        board.setCell(row, col, num);
                        if (solveSudoku(board)) {
                            return true;
                        }
                        board.setCell(row, col, value:0);
                    }
                }
                return false;
            }
        }
    }
    return true;
}
```

Algorithm Analysis:

- Time Complexity: O(9^(n²)) worst case
- Optimization: Random shuffling creates diverse solutions
- Constraint Propagation: Early validation prevents deep recursion
- Backtracking: Systematic exploration with pruning

Validation System

StandardSudokuValidator.java

```java
@Override
public boolean isValid(Board board, int row, int col, int num) {
    if (num < 1 || num > 9) {
        return false; // Explicit range check
    }
    // Check row
    for (int x = 0; x < Constants.BOARD_SIZE; x++) {
        if (board.getCell(row, x) == num) {
            return false;
        }
    }
    // Check column
    for (int x = 0; x < Constants.BOARD_SIZE; x++) {
        if (board.getCell(x, col) == num) {
            return false;
        }
    }
    // Check 3x3 subgrid
    int startRow = row - row % Constants.SUBGRID_SIZE;
    int startCol = col - col % Constants.SUBGRID_SIZE;
    for (int i = 0; i < Constants.SUBGRID_SIZE; i++) {
        for (int j = 0; j < Constants.SUBGRID_SIZE; j++) {
            if (board.getCell(i + startRow, j + startCol) == num) {
                return false;
            }
        }
    }
    // Note: This method checks local validity (row, column, subgrid).
    // It does not guarantee the move aligns with the puzzle's unique solution.
    return true;
}
```

Performance Optimization:

- Early Exit: Fails fast on first constraint violation
- O(1) Complexity: Fixed 9x9 grid means constant time
- Single Pass: No redundant checking
- Range Validation: Input validation prevents invalid states

Intelligent Hint System

HintSystem.java - Logical Techniques

```
private Hint findNakedSingle(int[][] puzzle) {
    for (int row = 0; row < 9; row++) {
        for (int col = 0; col < 9; col++) {
            if (puzzle[row][col] == 0) {
                List<Integer> candidates = getCandidates(puzzle, row, col);
                if (candidates.size() == 1) {
                    int value = candidates.get(index:0);

                    String explanation = String.format(
                            format:"Cell R%dC%d can only be %d", row + 1, col + 1, value);

                    String detailedSteps = String.format(
                            "Looking at cell R%dC%d:\n" +
                                    "• Row %d already has: %s\n" +
                                    "• Column %d already has: %s\n" +
                                    "• 3×3 box already has: %s\n" +
                                    "• Only %d is possible in this cell!\n\n" +
                                    "This is called a 'Naked Single' - the most basic Sudoku technique.",
                            row + 1, col + 1,
                            row + 1, getRowNumbers(puzzle, row),
                            col + 1, getColumnNumbers(puzzle, col),
                            getBoxNumbers(puzzle, row, col),
                            value
                    );

                    return new Hint(HintType.NAKED_SINGLE, row, col, value,
                            explanation, detailedSteps, costPoints:25);
                }
            }
        }
    }
    return null;
}
```

Advanced Pattern Recognition

```
private Hint findHiddenSingle(int[][] puzzle) {
    // Check rows for hidden singles
    for (int row = 0; row < 9; row++) {
        for (int num = 1; num <= 9; num++) {
            if (!isNumberInRow(puzzle, row, num)) {
                List<Integer> possibleCols = new ArrayList<>();
                for (int col = 0; col < 9; col++) {
                    if (puzzle[row][col] == 0 && canPlace(puzzle, row, col, num)) {
                        possibleCols.add(col);
                    }
                }

                if (possibleCols.size() == 1) {
                    int col = possibleCols.get(index:0);

                    String explanation = String.format(
                            format:"Number %d can only go in R%dC%d in this row", num, row + 1, col + 1);

                    String detailedSteps = String.format(
                            "Looking for number %d in row %d:\n" +
                                    "• Checking each empty cell in the row\n" +
                                    "• Cell R%dC%d is the ONLY place where %d can go\n" +
                                    "• All other empty cells are blocked by column or box constraints\n\n" +
                                    "This is called a 'Hidden Single' - %d is 'hidden' as the only possibility.",
                            num, row + 1, row + 1, col + 1, num, num
                    );

                    return new Hint(HintType.HIDDEN_SINGLE, row, col, num,
                            explanation, detailedSteps, costPoints:30);
                }
            }
        }
    }
```

Ice Mode Innovation

```java
public void updateFrozenCellState(Puzzle puzzle, Solution solution, int row, int col, JTextField cell) {
    if (puzzle.isFrozen(row, col) && !puzzle.isFixed(row, col)) {
        boolean hasAdjacentValidInput = hasAdjacentValidInput(puzzle, solution, row, col);
        if (hasAdjacentValidInput) {
            puzzle.setFrozen(row, col, frozen:false);
            cell.setBackground(((row / Constants.SUBGRID_SIZE + col / Constants.SUBGRID_SIZE) % 2 == 0)
                    ? Constants.ALTERNATE_CELL_COLOR : Constants.DEFAULT_CELL_COLOR);
            cell.setEditable(b:true);
        } else {
            cell.setBackground(Constants.FROZEN_CELL_COLOR);
            cell.setEditable(b:false);
        }
    }
}

private boolean hasAdjacentValidInput(Puzzle puzzle, Solution solution, int row, int col) {
    int[][] directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}}; // Up, down, left, right
    for (int[] dir : directions) {
        int newRow = row + dir[0];
        int newCol = col + dir[1];
        if (newRow >= 0 && newRow < Constants.BOARD_SIZE && newCol >= 0 && newCol < Constants.BOARD_SIZE) {
            if (!puzzle.getBoard().isEmpty(newRow, newCol) &&
                    puzzle.getBoard().getCell(newRow, newCol) == solution.getCell(newRow, newCol)) {
                return true;
            }
        }
    }
    return false;
}
```

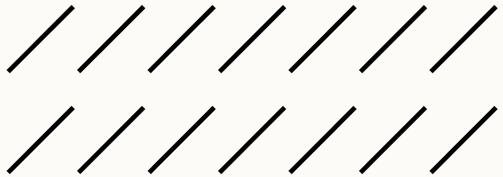
Game Innovation:

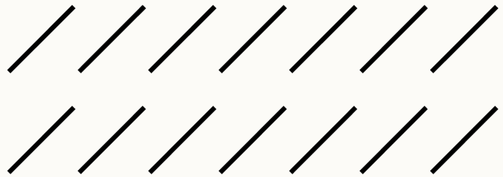
- Progressive Unlock: Cells become editable based on correct adjacent moves
- Strategic Gameplay: Players must plan sequence of moves
- Visual Feedback: Color coding shows cell states clearly
- Unique Mechanic: Adds puzzle-solving dimension beyond classic Sudoku

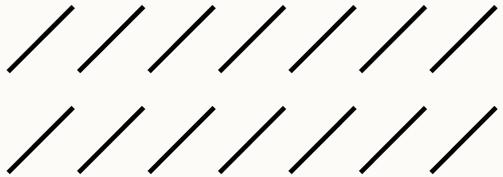
Ice Mode Innovation

Technical Implementation:

- Real-time Updates: Cell states change dynamically during gameplay
- Efficient Checking: O(1) adjacency validation
- State Management: Clear separation of frozen/fixed/normal cells
- UI Integration: Seamless visual state transitions







Core Algorithms Summary

1. Puzzle Generator
2. Validator
3. Hint System
4. Solver

Puzzle Generator

- Complete Solution Generation: Backtracking algorithm to fill entire 9x9 grid
- Cell Removal: Systematically removes cells while maintaining unique solution
- Difficulty Scaling: Controls number of pre-filled cells (Easy: 45, Expert: 17)
- Ice Mode Extension: Adds frozen cells with adjacency-based unlocking

Validator

- Constraint Checking: Validates row, column, and 3x3 subgrid constraints
- Real-time Validation: O(1) complexity for single cell validation
- Complete Board Check: Ensures entire solution is valid

Hint System

- Naked Single: Identifies cells with only one possible value
- Hidden Single: Finds numbers with only one possible position
- Progressive Difficulty: Cheaper hints for logical deductions, expensive for direct answers
- Educational Explanations: Detailed step-by-step solving techniques

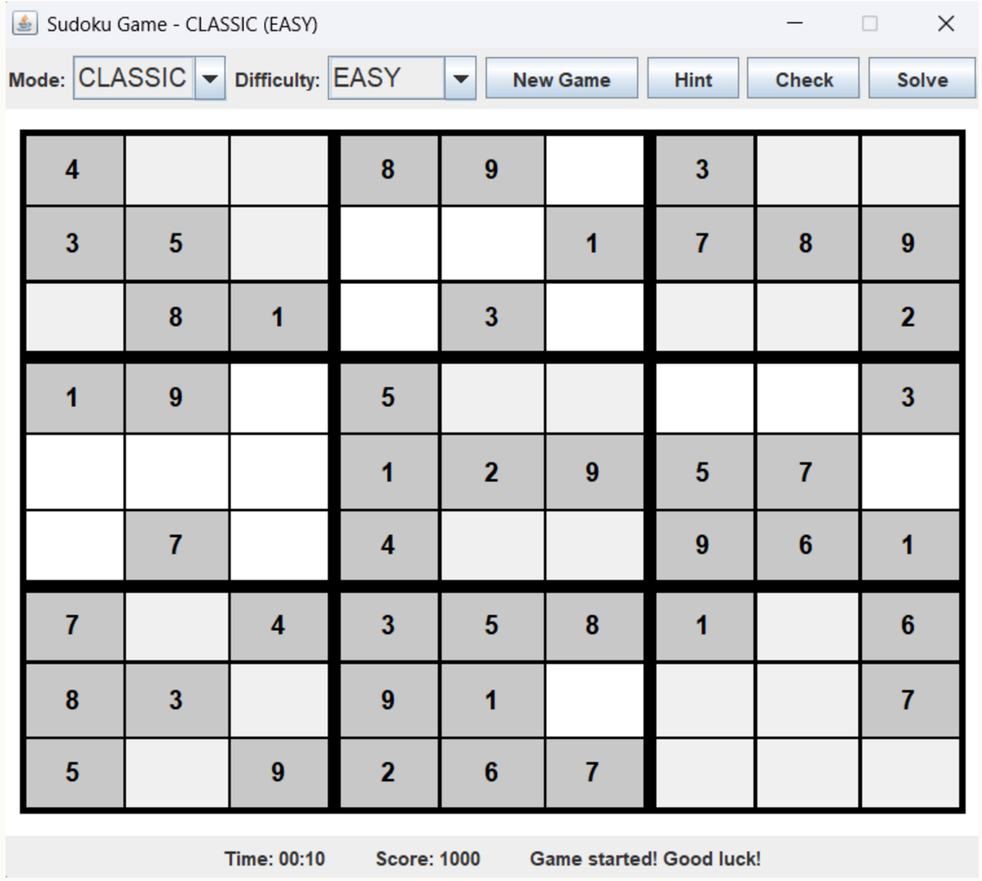
Solver

- Backtracking Algorithm: Systematic exploration of solution space
- Constraint Propagation: Early pruning of invalid branches
- Mode-Aware: Handles both Classic and Ice mode constraints

Complexity Analysis

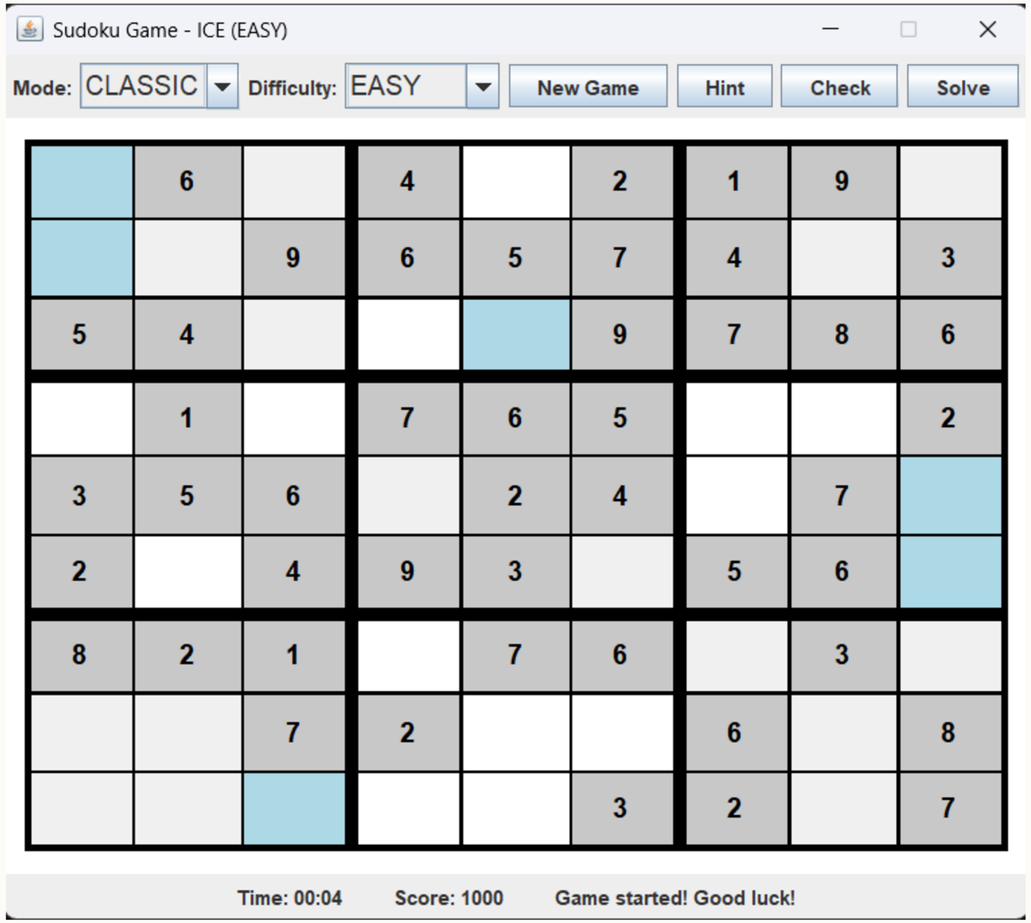
Algorithm	Time Complexity	Space Complexity	Notes
Puzzle Generation	O(9^(n²)) worst case	O(n²)	Backtracking with constraint propagation
Single Cell Validation	O(1)	O(1)	Direct constraint checking
Complete Board Validation	O(n²)	O(1)	Linear scan of all cells
Puzzle Solving	O(9^k) where k = empty cells	O(k)	Recursion depth limited by empty cells
Hint Generation	O(n²)	O(n²)	Board analysis for logical patterns
Ice Mode Updates	O(1)	O(1)	Adjacent cell checking

Demo & Key Features



TRADITIONAL SUDOKU: FILL EMPTY CELLS WITH NUMBERS 1-9

Demo & Key Features



INNOVATIVE TWIST: FROZEN CELLS (BLUE) UNLOCK WHEN ADJACENT CELLS ARE CORRECTLY FILLED

THANK YOU
FOR YOUR LISTENING

NEXT

Q&A

NEXT