VIETNAM NATIONAL UNIVERSITY OF HOCHIMINH CITY
THE INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

DATA STRUCTURES AND ALGORITHM
IT094IU

REPORT
TOPIC: BUILD A SUDOKU GAME IN JAVA

By Group 04 – Member List

1	Nguyễn Minh Đạt	ITITIU21323	Code + Slide
2	Lê Trọng Hiếu	ITITIU21311	Code + Report

Instructor: Dr. Vi Chi Thanh and Thai Trung Tin

Ho Chi Minh City, Viet Nam
2024 – 2025

TABLE OF CONTENTS

 Link Github: https://github.com/29Schiller/Sudoku-DSA-Project

Link Github: https://github.com/29Schiller/Sudoku-DSA-Project

Abstract

This report presents the design and implementation of a comprehensive Sudoku game application featuring multiple game variants implemented in Java. The project demonstrates the application of SOLID principles and object-oriented design patterns, particularly the Model-View-Controller (MVC) architecture and Strategy pattern, to create an extensible gaming framework. The implementation includes three distinct Sudoku variants: Classic Sudoku, Ice Sudoku with frozen cell mechanics, and Killer Sudoku with cage-based sum constraints. The system incorporates puzzle generation algorithms, validation mechanisms, user interface design using Java Swing, and comprehensive game state management to deliver a complete gaming experience with varying difficulty levels and interactive features.

Link Github: https://github.com/29Schiller/Sudoku-DSA-Project

Chapter 1: Introduction

1.1 Aims

This project aims to develop a comprehensive Sudoku game application that demonstrates advanced software engineering principles and provides an engaging user experience. The primary objectives include:

- **Educational Purpose**: Implement multiple Sudoku variants to showcase algorithmic complexity and design pattern applications
- **Technical Excellence**: Demonstrate SOLID principles, clean architecture, and efficient algorithm implementation
- **User Experience**: Create an intuitive, responsive interface with multiple difficulty levels and intelligent assistance features
- **Innovation**: Introduce novel game mechanics through the Ice Mode variant that extends traditional Sudoku gameplay

The project serves as a practical demonstration of object-oriented design principles, GUI development, and complex algorithm implementation in Java.

1.2 Background

Sudoku, a logic-based number-placement puzzle, has become one of the most popular puzzle games worldwide since its modern form was introduced in the 1980s. The game consists of a 9×9 grid divided into nine 3×3 subgrids, where players must fill each cell with digits 1-9 such that each row, column, and subgrid contains all digits exactly once.

While traditional Sudoku provides substantial algorithmic challenges in puzzle generation, validation, and solving, this project extends beyond the classic implementation by:

- Implementing multiple game variants with different mechanics
- Creating an intelligent hint system that teaches solving techniques
- Developing a robust architecture that supports easy extension to new game modes
- Providing comprehensive user interface design with modern usability principles

Chapter 2: Background

2.1 Sudoku Game Variants

Traditional Sudoku provides a foundation for numerous variants that introduce additional constraints or mechanics. This project implements two primary variants:

Classic Sudoku: The traditional 9×9 grid format where players fill empty cells following standard Sudoku rules. This mode serves as the baseline implementation and demonstrates fundamental algorithms for puzzle generation, validation, and solving.

Ice Mode Sudoku: An innovative variant where certain cells are initially "frozen" and become editable only when adjacent cells contain correct values. This introduces a strategic layer requiring players to plan move sequences, as the order of number placement affects which cells become available.

2.2 SOLID Principles Application

The project architecture is built around the five SOLID principles of object-oriented design:

Single Responsibility Principle (SRP): Each class has a single, well-defined purpose. For example, **SudokuValidator** only handles constraint validation, while PuzzleGenerator focuses solely on puzzle creation.

Open/Closed Principle (OCP): The system is open for extension but closed for modification. New game modes can be added by implementing the **GameMode** interface without modifying existing code.

Liskov Substitution Principle (LSP): Derived classes can be substituted for their base types. **ClassicMode** and **IceMode** are fully interchangeable through the **GameMode** interface.

Interface Segregation Principle (ISP): Interfaces are small and focused. Rather than one large interface, the system uses specific interfaces like **GameActionListener, TimerListener**, and **GameStartListener.**

Dependency Inversion Principle (DIP): High-level modules depend on abstractions rather than concrete implementations. The controller depends on interfaces rather than concrete classes.

2.3 Model-View-Controller Pattern

The Model-View-Controller pattern separates an application into three core components:

- **Model:** Manages the application's data, business logic, and rules. It's independent of the user interface.
- **View:** Responsible for displaying the data from the Model to the user. It also handles user input and sends it to the Controller.
- **Controller:** Acts as an intermediary between the Model and the View. It receives input from the View, processes it (potentially updating the Model), and then updates the View.

2.3.1. Model (model/, generator/, validator/, mode/, solver/, utils/ related parts)

The Model layer encapsulates all the core logic and data structures related to the Sudoku game itself. It is unaware of how its data is displayed or how user input is handled.

- **model/:**
    - Board.java: Represents the Sudoku grid, holding the current state of the game (numbers in cells). It would contain methods for getting and setting cell values, checking if a cell is fixed (part of the initial puzzle), etc.
    - Puzzle.java: Represents the initial Sudoku puzzle, likely containing the numbers that are pre-filled and cannot be changed by the user.
    - Solution.java: Represents the solved state of the Sudoku puzzle. This is the correct solution that the user is trying to achieve.
- **generator/:**
    - PuzzleGenerator.java: An interface or abstract class for generating Sudoku puzzles.
    - ClassicPuzzleGenerator.java: Concrete implementation for generating classic Sudoku puzzles.
    - IcePuzzleGenerator.java: Concrete implementation for generating "Ice" mode Sudoku puzzles (suggests a variation with specific rules). These generators are part of the Model's logic, as they determine the game's initial state.
- **validator/:**
    - SudokuValidator.java: An interface or abstract class for validating Sudoku rules.
    - StandardSudokuValidator.java: Concrete implementation for checking if a Sudoku board state is valid according to standard Sudoku rules (no repeated numbers in rows, columns, or 3x3 blocks). This is crucial business logic within the Model.

- **mode/:**
    - GameMode.java: An interface or abstract class defining common behavior for different game modes.
    - ClassicMode.java: Implements the GameMode for the classic Sudoku experience. This might involve specific validation rules or puzzle generation logic relevant to classic mode.
    - IceMode.java: Implements GameMode for the "Ice" variation. This reinforces the idea that different game modes have distinct rules or behaviors, which are part of the Model's domain.
- **solver/:**
    - SudokuSolver.java: Contains the algorithms to solve a given Sudoku puzzle. This is pure business logic that operates on Board or Puzzle objects, making it a core part of the Model.
- **utils/ (Model-related parts):**
    - DifficultyLevel.java: An enum representing different difficulty levels (e.g., Easy, Medium, Hard). This influences puzzle generation and is a property of the Model.
    - GameModeType.java: An enum to differentiate between game modes (e.g., Classic, Ice). This is a categorization within the Model.

2.3.2. View (view/)

The View layer is responsible for presenting the Sudoku game to the user and capturing user interactions. It should not contain any game logic or direct manipulation of the game state (Model).

- **WelcomeScreen.java:** Displays the initial screen of the game, likely allowing the user to select game mode, difficulty, or start a new game.
- **GameView.java:** The main game screen where the Sudoku board is displayed. It renders the Board's state (numbers in cells, fixed cells, user-entered numbers). It would likely contain UI elements for inputting numbers, hints, checking progress, etc.
- **EndScreen.java:** Displays the results of the game (e.g., "You Won!", "Game Over," time taken).
- **HintDialog.java:** A dialog window for providing hints to the user.

- **UIUtils.java:** Utility class for common UI operations or styling, ensuring consistency in the View.

- **Listener Interfaces (GameStartListener.java, GameActionListener.java, TimerListener.java):**

    - These interfaces define contracts for events originating from the View. The View implements these listeners to detect user actions (e.g., clicking a cell, entering a number, starting a game, timer events).
    - Crucially, these listeners would then delegate the actual handling of the events to the **Controller**, rather than directly manipulating the Model. This is a common pattern for View-to-Controller communication.
- **HintSystem.java:** While its name suggests system-level logic, in a strict MVC View, this might be a UI component that facilitates hint display and interaction, rather than calculating the hint itself (which would be a Model concern, likely handled by the SudokuSolver or a similar component). If it contains hint *generation* logic, it blurs the line and should ideally be moved to the Model or a service called by the Model. If it purely displays the hint and handles user interaction with the hint feature, then it belongs to the View.

2.3.3. Controller (controller/)

The Controller acts as the bridge between the View and the Model. It receives user input from the View, interprets it, performs necessary actions on the Model, and then updates the View to reflect the changes.

- **SudokuGameController.java:** This is the central controller for the game.

    - It would likely implement the listener interfaces defined in the view/ package (GameStartListener, GameActionListener, TimerListener).
    - When a user action occurs (e.g., clicking a cell, entering a number, requesting a hint):
        1. The GameView (or other View components) would notify the SudokuGameController through the listener interface.
        2. The SudokuGameController would then:
            - **Interact with the Model:** Call methods on Board to update a cell, use SudokuValidator to check validity, use PuzzleGenerator to create a new puzzle, or interact with SudokuSolver for hints or solutions.
            - **Update the View:** After the Model has been updated, the Controller would instruct the GameView (or EndScreen, etc.) to

refresh its display to show the new state of the game. For example, if a number is successfully placed, the Controller tells the GameView to re-render that cell. If the game is won, it tells EndScreen to display.

- **Dependency Management:** The SudokuGameController would hold references to instances of the Model components (e.g., Board, PuzzleGenerator, SudokuValidator) and View components (e.g., GameView, WelcomeScreen, EndScreen).

How they interact:

1. **User Interaction (View to Controller):** The user interacts with the GameView (e.g., clicks a cell, types a number). GameView detects this and, through its implemented listeners, notifies the SudokuGameController.
2. **Processing (Controller to Model):** The SudokuGameController receives the user's input. It then calls appropriate methods on the Model objects (e.g., board.setCellValue(row, col, value)). The Controller might also use SudokuValidator to check if the move is valid or SudokuSolver for hint requests.
3. **State Change (Model):** The Model updates its internal state (e.g., Board object changes a cell's value). The Model itself does not directly communicate with the View.
4. **Updating Display (Controller to View):** After the Model has been updated, the SudokuGameController instructs the GameView to redraw or update specific elements to reflect the changes in the Model. The View queries the Model for the necessary data to display (e.g., gameView.updateCell(row, col, board.getCellValue(row, col))).
5. **Initial Setup (Main.java):** The Main.java file would be responsible for instantiating the core components: creating an instance of SudokuGameController, passing it instances of the GameView, WelcomeScreen, and the necessary Model components (Board, PuzzleGenerator, etc.). This is where the initial wiring of the MVC components occurs.

This clear separation of concerns makes the application:

- **Maintainable:** Changes in the UI don't require changes in the game logic, and vice versa.
- **Testable:** Model components can be tested independently of the UI.
- **Flexible:** Different UIs could be plugged into the same Model, or different game modes/logic could be implemented without altering the UI.
-

2.4 Java Swing Framework

Java Swing provides the foundation for the user interface implementation, offering several advantages for this project:

Rich Component Library: Swing provides comprehensive GUI components including **JTextField** for cell input, **JButton** for actions, and **JComboBox** for mode selection.

Event-Driven Architecture: Swing's event model aligns well with the game's interactive nature, supporting real-time user input validation and feedback.

Layout Management: Flexible layout managers enable responsive design that adapts to different screen sizes and resolutions.

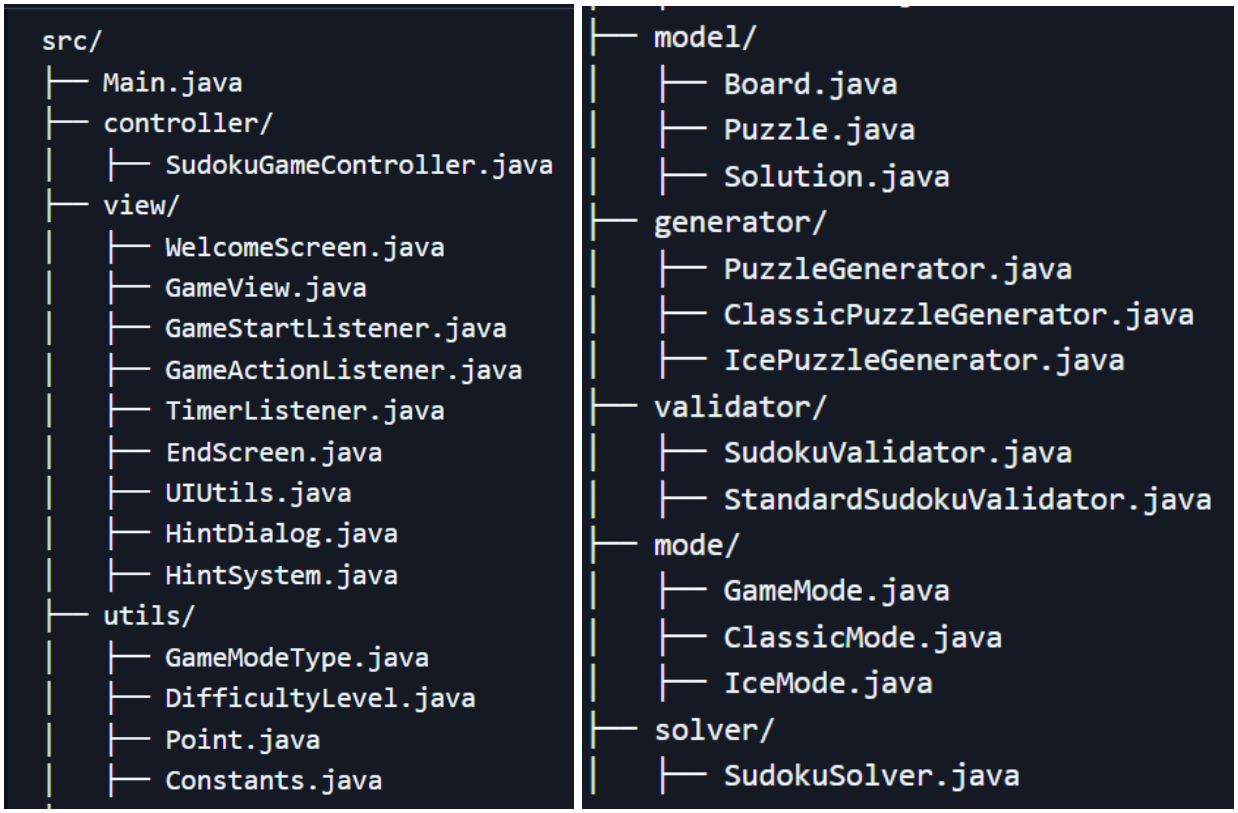
Customization Capabilities: Swing allows extensive customization of component appearance and behavior, enabling the visual differentiation between different cell states (fixed, frozen, editable).

The choice of Swing over more modern frameworks reflects the project's focus on core algorithmic implementation rather than web-based deployment, while still providing a professional desktop application experience.

Chapter 3: Code Structure

3.1 Package Organization

The project follows a clear package structure that reflects the separation of concerns and facilitates maintainability:



Package Dependencies follow a clear hierarchy that respects dependency inversion:

- controller depends on model, view, mode, generator interfaces
- view depends only on model and utils, not on specific mode implementations
- mode depends on model and validator interfaces
- generator depends on model and utils
- validator depends only on model
- model and utils have no external dependencies

3.2 Game Mode Implementation

3.2.1 Classic Sudoku Mode

The Classic Mode implementation provides the foundation for traditional Sudoku gameplay:

Cell Rendering: Distinguishes between fixed (pre-filled) and editable cells through visual styling:

```
public class ClassicMode implements GameMode {
    @Override
    public void renderCell(JTextField cell, int row, int col, Puzzle puzzle, Solution solution) {
        if (!puzzle.getBoard().isEmpty(row, col)) {
            cell.setText(String.valueOf(puzzle.getBoard().getCell(row, col)));
            cell.setEditable(b:false);
            cell.setBackground(Constants.FIXED_CELL_COLOR);
            cell.setFont(new Font(name:"Arial", Font.BOLD, size:16));
        } else {
            cell.setText(t:"");
            cell.setEditable(b:true);
            cell.setFont(new Font(name:"Arial", Font.PLAIN, size:16));
        }
    }
```

Move Validation: Implements standard Sudoku constraint checking:

- Validates input range (1-9)
- Checks row, column, and subgrid constraints
- Maintains puzzle state consistency during validation

Completion Detection: Determines puzzle completion by verifying all cells are filled and all constraints are satisfied.

3.2.2 Ice Sudoku Mode

.The Ice Mode introduces innovative frozen cell mechanics that add strategic depth to traditional Sudoku:

Frozen Cell Management: Cells can be in one of three states:

- **Fixed**: Pre-filled numbers that cannot be changed
- **Frozen**: Empty cells that become editable when adjacent cells have correct values
- **Editable**: Normal empty cells that can be filled immediately

Dynamic State Updates: The core innovation lies in the dynamic unfreezing mechanism:

```java
public void updateFrozenCellState(Puzzle puzzle, Solution solution, int row, int col, JTextField cell) {
    if (puzzle.isFrozen(row, col)){
        boolean shouldUnlock = hasAdjacentUserFilledCorrectCellInSubgrid4Dir(puzzle, solution, row, col);

        if (shouldUnlock) {
            puzzle.setFrozen(row, col, frozen:false); // This line is crucial for internal state
            cell.setBackground(((row / Constants.SUBGRID_SIZE + col / Constants.SUBGRID_SIZE) % 2 == 0)
                    ? Constants.ALTERNATE_CELL_COLOR : Constants.DEFAULT_CELL_COLOR);
            cell.setEditable(b:true);
            cell.setForeground(Color.BLACK);
        } else {
            // If it shouldn't unlock, ensure it visually stays frozen.
            cell.setBackground(Constants.FROZEN_CELL_COLOR);
            cell.setEditable(b:false);
        }
    }
}
```

3.2.3 Mode Validation System

Both game modes share a common validation foundation while allowing mode-specific extensions:

Unified Interface: All modes implement the same validation interface, ensuring consistency:

```java
@Override
public boolean isValidMove(Puzzle puzzle, Solution solution, SudokuValidator validator, int row, int col, int value) {
    if (value < 1 || value > 9) {
        return false; // Explicit range check
    }
    // Save original value to restore after validation check
    int originalValue = puzzle.getBoard().getCell(row, col);
    puzzle.getBoard().setCell(row, col, value:0); // Temporarily clear current cell for validation
    boolean isValid = validator.isValid(puzzle.getBoard(), row, col, value);
    puzzle.getBoard().setCell(row, col, originalValue); // Restore original value

    // Note: This checks local validity (row, column, subgrid).
    // It does not guarantee the move aligns with the unique solution.
    return isValid;
}
```

Error Detection: The validation system provides immediate feedback:

- Invalid moves are highlighted in red
- Score penalties are applied for incorrect entries
- Status messages guide player understanding

Solution Checking: Comprehensive verification against the complete solution:

- Identifies incorrect values in filled cells
- Provides error count for progress tracking

- Maintains game state integrity

3.3 Generation Algorithm Design

The puzzle generation system employs a sophisticated two-phase approach:

Phase 1: Complete Solution Generation Uses a randomized backtracking algorithm to create a fully solved 9×9 Sudoku grid:

```
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;
}
```

Phase 2: Strategic Cell Removal Systematically removes cells while maintaining solution uniqueness:

Link Github: https://github.com/29Schiller/Sudoku-DSA-Project

```java
private Board createPuzzleFromSolution(Board solution, int cellsToFill) {
    Board puzzle = new Board(solution.getGrid());
    int cellsToRemove = Constants.BOARD_SIZE * Constants.BOARD_SIZE - cellsToFill;
    List<Point> positions = new ArrayList<>();
    for (int row = 0; row < Constants.BOARD_SIZE; row++) {
        for (int col = 0; col < Constants.BOARD_SIZE; col++) {
            positions.add(new Point(row, col));
        }
    }
    Collections.shuffle(positions, random);

    for (int i = 0; i < cellsToRemove && i < positions.size(); i++) {
        Point pos = positions.get(i);
        int value = puzzle.getCell(pos.x, pos.y);
        puzzle.setCell(pos.x, pos.y, value:0);
        if (!hasUniqueSolution(puzzle)) {
            puzzle.setCell(pos.x, pos.y, value); // Revert if multiple solutions
        }
    }
    return puzzle;
}
```

6.2 Difficulty Level Implementation

The difficulty system is implemented through an enum that controls puzzle complexity:

```java
public enum DifficultyLevel {
    EASY(cellsToFill:47),
    MEDIUM(cellsToFill:35),
    HARD(cellsToFill:25),
    EXPERT(cellsToFill:17);

    private final int cellsToFill;

    DifficultyLevel(int cellsToFill) {
        this.cellsToFill = cellsToFill;
    }

    public int getCellsToFill() {
        return cellsToFill;
    }
}
```

Difficulty Progression:

- **Easy**: 47 pre-filled cells (36 empty) - Provides ample clues for logical solving
- **Medium**: 35 pre-filled cells (46 empty) - Requires intermediate techniques
- **Hard**: 25 pre-filled cells (56 empty) - Demands advanced logical reasoning
- **Expert**: 17 pre-filled cells (64 empty) - Minimal clues requiring expert techniques

6.3 Variant-Specific Generation

Classic Generation: Focuses on creating puzzles with unique solutions that can be solved through logical deduction:

```java
@Override
public Puzzle generatePuzzle(DifficultyLevel difficulty) {
    Board solutionBoard = generateCompleteSudoku();
    this.solution = new Solution(solutionBoard);
    Board puzzleBoard = createPuzzleFromSolution(solutionBoard, difficulty.getCellsToFill());
    boolean[][] fixedCells = new boolean[Constants.BOARD_SIZE][Constants.BOARD_SIZE];
    for (int row = 0; row < Constants.BOARD_SIZE; row++) {
        for (int col = 0; col < Constants.BOARD_SIZE; col++) {
            fixedCells[row][col] = puzzleBoard.getCell(row, col) != 0;
        }
    }
    return new Puzzle(puzzleBoard, fixedCells, frozenCells:null);
}
```

Ice Mode Generation: Extends classic generation by adding frozen cell placement:

```java
@Override
public Puzzle generatePuzzle(DifficultyLevel difficulty) {
    Puzzle classicPuzzle = classicGenerator.generatePuzzle(difficulty);
    this.solution = classicGenerator.getSolution();
    Board puzzleBoard = classicPuzzle.getBoard();
    boolean[][] fixedCells = new boolean[Constants.BOARD_SIZE][Constants.BOARD_SIZE];

    for (int row = 0; row < Constants.BOARD_SIZE; row++) {
        for (int col = 0; col < Constants.BOARD_SIZE; col++) {
            fixedCells[row][col] = puzzleBoard.getCell(row, col) != 0;
        }
    }
    boolean[][] frozenCells = new boolean[Constants.BOARD_SIZE][Constants.BOARD_SIZE];
    int cellsToFreeze = (int) ((Constants.BOARD_SIZE * Constants.BOARD_SIZE - difficulty.getCellsToFill()) * 0.2);
    List<Point> candidateFrozenCells = new ArrayList<>();
    for (int row = 0; row < Constants.BOARD_SIZE; row++) {
        for (int col = 0; col < Constants.BOARD_SIZE; col++) {

            if (puzzleBoard.isEmpty(row, col) &&
                    hasAdjacentEmptyFillableCellInSubgrid4Dir(puzzleBoard, fixedCells, row, col)) { // ONLY this condition rem
                candidateFrozenCells.add(new Point(row, col));
            }
        }
    }
    Collections.shuffle(candidateFrozenCells, random);
    for (int i = 0; i < Math.min(cellsToFreeze, candidateFrozenCells.size()); i++) {
        Point p = candidateFrozenCells.get(i);
        frozenCells[p.x][p.y] = true;
    }
    return new Puzzle(puzzleBoard, fixedCells, frozenCells);
}
```

3.4 User Interface Design

3.4.1 Screen Management System

The application implements a multi-screen architecture that guides users through different phases of the game experience:

Welcome Screen: The entry point that allows mode and difficulty selection:

- Clean, intuitive layout with clear options
- Validates user selections before proceeding
- Provides visual feedback for user choices

Game Screen: The main gameplay interface featuring:

- 9×9 Sudoku grid with visual subgrid separation
- Control panel with action buttons (New Game, Hint, Check, Solve)
- Status panel showing time, score, and game messages
- Real-time feedback for user actions

End Screen: Post-game summary and continuation options:

- Displays final score and completion time
- Offers options to start new games with different configurations
- Maintains game flow continuity

Hint Dialog: Educational overlay system:

- Detailed explanations of solving techniques
- Interactive options to show or apply hints
- Progressive cost system encouraging learning

3.4.2 Game Board Visualization

The game board implementation emphasizes clarity and usability:

```java
import java.awt.Color;

public final class Constants {
    public static final int BOARD_SIZE = 9;
    public static final int SUBGRID_SIZE = 3;
    public static final String CLASSIC_MODE = "CLASSIC";
    public static final String ICE_MODE = "ICE";
    public static final Color FROZEN_CELL_COLOR = new Color(r:173, g:216, b:230);
    public static final Color FIXED_CELL_COLOR = new Color(r:200, g:200, b:200);
    public static final Color DEFAULT_CELL_COLOR = Color.WHITE;
    public static final Color ALTERNATE_CELL_COLOR = new Color(r:240, g:240, b:240);
    public static final Color ERROR_CELL_COLOR = new Color(r:255, g:200, b:200);
    public static final Color HINT_CELL_COLOR = new Color(r:0, g:150, b:0);
    public static final Color SOLVE_CELL_COLOR = Color.BLUE;
    public static final String[] DIFFICULTIES = {
        DifficultyLevel.EASY.name(),
        DifficultyLevel.MEDIUM.name(),
        DifficultyLevel.HARD.name(),
        DifficultyLevel.EXPERT.name()
    };

    private Constants() {} // Prevent instantiation
}
```

3.4.3 User Interaction Handling

The interaction system provides immediate feedback and validation:

Input Validation: Real-time constraint checking as users type:

```
cells[row][col].addKeyListener(new KeyAdapter() {
    @Override
    public void keyTyped(KeyEvent e) {
        char ch = e.getKeyChar();
        if (!Character.isDigit(ch) || ch == '0' || !cells[r][c].getText().isEmpty()) {
            e.consume();
        }
    }
});
```

Focus Management: Automatic validation when users leave cells:

```
cells[row][col].addFocusListener(new FocusAdapter() {
    @Override
    public void focusLost(FocusEvent e) {
        if (actionListener != null) {
            actionListener.onCellInput(r, c, cells[r][c].getText());
        }
    }
});
```

Visual Feedback: Immediate response to user actions:

- Color changes for valid/invalid moves
- Status message updates
- Score adjustments with penalties for errors

Accessibility Features: Design considerations for usable interface:

- Keyboard navigation support
- Clear visual indicators for different states
- Consistent interaction patterns across all screens

3.5 Implementation Details

3.5.1 Core Model Classes

The model layer provides the foundation for game state management through well-designed data structures:

Board Class: Represents the fundamental 9×9 grid:

```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;
    }
}
```


Link Github: https://github.com/29Schiller/Sudoku-DSA-Project

Design Principles:

- **Immutability**: Deep copying prevents external modification of internal state
- **Encapsulation**: Private grid access through controlled methods
- **Defensive Programming**: Returned arrays are copies to prevent external tampering

Puzzle Class: Extends Board with game-specific metadata:

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

    publ                                rd, boolean[][] fixedCells, boolean[][] frozenCells) {
                model.Puzzle
                final Puzzle this
                                    ixedCells;
        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;
        }
    }
}
```

3.5.2 Controller Architecture

The controller serves as the central coordinator implementing the game logic:

Component Initialization: The controller manages dependency injection:

```
public SudokuGameController(SudokuValidator validator, SudokuSolver solver, GameView gameView, WelcomeScreen welcomeScreen) {
    this.validator = validator;
    this.solver = solver;
    this.gameView = gameView;
    this.welcomeScreen = welcomeScreen;
    puzzleGenerators = new HashMap<>();
    puzzleGenerators.put(Constants.CLASSIC_MODE, new ClassicPuzzleGenerator(validator));
    puzzleGenerators.put(Constants.ICE_MODE, new IcePuzzleGenerator(validator));
    gameModes = new HashMap<>();
    gameModes.put(Constants.CLASSIC_MODE, new ClassicMode());
    gameModes.put(Constants.ICE_MODE, new IceMode());
    initializeListeners();
}
```

Event Handling: Centralized processing of user interactions:

```
@Override
public void onCellInput(int row, int col, String value) {
    if (puzzle.isFixed(row, col)) {
        return;
    }
    // ADDED: Prevent direct input into frozen cells and provide feedback.
    if (puzzle.isFrozen(row, col)) { //
        gameView.showStatus(status:"This cell is frozen and cannot be changed directly."); //
        return; //
    }

    if (value.isEmpty()) {
        puzzle.getBoard().setCell(row, col, value:0); //
        gameView.updateCell(row, col, value:"", Color.BLACK, getDefaultBackground(row, col), editable:true); //
        if (gameView.getMode().equals(Constants.ICE_MODE)) { //
            checkAdjacentFrozenCells(row, col); //
        }
        return;
    }
    int num = Integer.parseInt(value); //
    mode.GameMode gameMode = gameModes.get(gameView.getMode()); //
    puzzle.getBoard().setCell(row, col, num); //

    if (gameMode.isValidMove(puzzle, solution, validator, row, col, num)) { //
        Color background = getDefaultBackground(row, col); //
        Color textColor = Color.BLACK; // Default to black //

        if (solution.getBoard().getCell(row, col) != num) { //
            textColor = Color.RED; //
            background = Constants.ERROR_CELL_COLOR; // Or a specific error color if defined //
            score = Math.max(a:0, score - 10); //
            gameView.showStatus(status:"Correct number for game rules, but not the solution! Check your moves."); //
```

```
        } else {
            gameView.showStatus(status:"Valid move!"); //
        }

        gameView.updateCell(row, col, String.valueOf(num), textColor, background, editable:true); //

        if (gameView.getMode().equals(Constants.ICE_MODE)) { //
            ((IceMode) gameModes.get(Constants.ICE_MODE)).updateFrozenCellState(puzzle, solution, row, col, gameView.cells[row]
            checkAdjacentFrozenCells(row, col); //
        }

        if (solution.getBoard().getCell(row, col) == num && gameMode.isPuzzleComplete(puzzle, validator)) { //
            gameView.stopTimer(); //
            int bonus = Math.max(a:0, 1000 - timeElapsed - hintCount * 50); //
            score += bonus; //
            gameView.updateScore(score); //
            gameView.showStatus(status:"Congratulations! Puzzle solved!"); //
            gameView.showVictoryMessage("Congratulations!\nTime: " + formatTime(timeElapsed) + //
                    "\nScore: " + score); //
            gameView.dispose(); //
            EndScreen endScreen = new EndScreen(score, timeElapsed); //
            endScreen.setListener(this); //
            endScreen.setVisible(b:true); //
        }
    } else {
        score = Math.max(a:0, score - 10); //
        gameView.updateScore(score); //
        gameView.showStatus(status:"Invalid move! Try again."); //
        gameView.updateCell(row, col, String.valueOf(num), Color.RED, Constants.ERROR_CELL_COLOR, editable:true); //

        puzzle.getBoard().setCell(row, col, value:0); //
    }
}
```

3.5.3 Event Handling and Game State

State Management: The controller maintains comprehensive game state:

```
private Puzzle puzzle;           // Current puzzle state
private Solution solution;       // Complete solution for validation
private int score;               // Player score with penalties
private int timeElapsed;         // Game duration in seconds
private int hintCount;           // Number of hints used
```

Timer Management: Real-time game timing with UI updates:

```java
@Override
public void onTimerTick() {
    timeElapsed++; //
    gameView.updateTime(formatTime(timeElapsed)); //
}
```

```java
private String formatTime(int seconds) {
    int minutes = seconds / 60; //
    int secs = seconds % 60; //
    return String.format(format:"%02d:%02d", minutes, secs); //
}
```

Score Calculation: Dynamic scoring system with bonuses and penalties:

```java
if (solution.getBoard().getCell(row, col) == num && gameMode.isPuzzleComplete(puzzle, validator)) { //
    gameView.stopTimer(); //
    int bonus = Math.max(a:0, 1000 - timeElapsed - hintCount * 50); //
    score += bonus; //
    gameView.updateScore(score); //
    gameView.showStatus(status:"Congratulations! Puzzle solved!"); //
    gameView.showVictoryMessage("Congratulations!\nTime: " + formatTime(timeElapsed) + //
            "\nScore: " + score); //
    gameView.dispose(); //
    EndScreen endScreen = new EndScreen(score, timeElapsed); //
    endScreen.setListener(this); //
    endScreen.setVisible(b:true); //
}
```

Error Handling: Comprehensive exception management:

Link Github: https://github.com/29Schiller/Sudoku-DSA-Project

```java
private void startNewGame(String mode, String difficulty) {
    try {
        PuzzleGenerator generator = puzzleGenerators.get(mode); //
        mode.GameMode gameMode = gameModes.get(mode); //
        if (generator == null || gameMode == null) { //
            throw new IllegalArgumentException("Invalid mode: " + mode); //
        }
        System.out.println("Initializing game Mode: " + mode + " - Difficulty: " + difficulty); //
        this.puzzle = generator.generatePuzzle(DifficultyLevel.valueOf(difficulty)); //
        this.solution = generator.getSolution(); //
        this.score = 1000; //
        this.timeElapsed = 0; //
        this.hintCount = 0; //
        gameView.updateBoard(puzzle, solution, gameMode); //
        gameView.updateScore(score); //
        gameView.updateTime(formatTime(timeElapsed)); //
        gameView.showStatus(status:"Game started! Good luck!"); //
        gameView.setTitle("Sudoku Game - " + mode + " (" + difficulty + ")"); //
    } catch (IllegalArgumentException e) {
        JOptionPane.showMessageDialog(gameView, "Failed to initialize game: " + e.getMessage(), title:"Error", JOptionPane.ERRO
    }
}
```

Chapter 4: Data Structures and Algorithm

4.1 Data Structure and algorithms:

4.1.1 Generator

The ClassicPuzzleGenerator and IcePuzzleGenerator classes are responsible for generating Sudoku puzzles.

- **Classic Puzzle Generation:**
  - **Backtracking Algorithm (for generateCompleteSudoku and solveSudoku):** This is the core algorithm for creating a valid, complete Sudoku board.
    - It starts with an empty board and tries to place numbers from 1 to 9 into each empty cell.
    - For each cell, it shuffles the numbers 1-9 and attempts to place them.
    - If a number is valid (doesn't violate Sudoku rules in its row, column, or 3x3 subgrid), it places the number and recursively calls itself for the next cell.
    - If the recursive call returns true (meaning a solution was found for the rest of the board), it keeps the number.
    - If the recursive call returns false (meaning no solution could be found with the current number), it backtracks, clears the cell, and tries the next number.
    - This continues until the board is filled or all possibilities have been exhausted for a given cell.

- **Backtracking with Solution Counting (for hasUniqueSolution and countSolutions):** To ensure the generated puzzle has a unique solution, the generator uses a modified backtracking algorithm.
  - It attempts to solve the puzzle, and instead of stopping at the first solution, it continues to explore all possible paths to count how many valid solutions exist.
  - If countSolutions returns anything other than 1, the number previously removed is restored, and another cell is attempted for removal.

- **Ice Puzzle Generation:**
  - The IcePuzzleGenerator leverages the ClassicPuzzleGenerator to first create a standard Sudoku puzzle and its solution.
  - **Heuristic/Rule-based Cell Freezing:** It then identifies "frozen" cells based on a specific rule: a cell can be frozen only if it is empty and has an adjacent empty, fillable cell in the 4 cardinal directions within its subgrid. It randomly selects a certain percentage of these candidate cells to be frozen. This isn't a complex algorithm like backtracking, but rather a set of rules for selecting cells.

4.1.2 Validator

The StandardSudokuValidator class implements the SudokuValidator interface.

- **Constraint Checking (for isValid):** This method checks if placing a given num at a specific row and col is valid according to standard Sudoku rules.
  - It iterates through the entire row to ensure num is not already present.
  - It iterates through the entire col to ensure num is not already present.
  - It iterates through the relevant 3x3 subgrid to ensure num is not already present.
  - This is a direct application of the Sudoku rules as constraints.

Full Board Validation (for isValidSudoku): This method iterates through every cell on the board. For each non-empty cell, it temporarily removes the cell's value and then uses the isValid method to check if that value could be legally placed there considering all other *existing* numbers. This effectively validates if the current state of the board adheres to Sudoku rules.

4.1.3 Hint System

Constraint Checking/Validation (canPlace method):

- **Concept:** This algorithm directly checks if placing a specific number (num) at a given row and col adheres to all the fundamental Sudoku rules.
- **Application:** The canPlace method within HintSystem.HintGenerator is crucial. It iterates through the target row, column, and 3x3 subgrid to ensure that num is not already present in

any of these locations. This is a core component for determining valid moves and generating logical hints.

Iterative Search / Brute-Force (with pruning):

- **Concept:** Systematically checking each possibility or location until a condition is met. While not "brute-force" in the sense of trying all combinations, it iterates through all relevant cells/numbers.
- **Application:**
  - **findNakedSingle:** Iterates through every empty cell on the board. For each cell, it generates a list of candidates. If only one candidate is found (a "Naked Single"), the algorithm stops and returns that hint.
  - **findHiddenSingle:** This involves nested iterations – first through rows/columns, then through numbers (1-9). For each number, it checks every empty cell in that row/column to see if the number can only be placed in *one specific* cell within that row/column (a "Hidden Single").

Prioritized Decision Making (getBestHint):

- **Concept:** A simple control flow algorithm that attempts different strategies in a predefined order of preference until a suitable result is found.
- **Application:** getBestHint first tries to find a "Naked Single" hint. If none is found, it proceeds to look for a "Hidden Single." If neither of these logical hints is available, it defaults to providing a "Direct Answer" (a random correct number). This ensures that more strategic hints are offered before a straightforward solution.

Random Selection:

- **Concept:** Choosing an element from a set or list with equal probability.
- **Application:** Used in getDirectHint to randomly select an empty cell for which to provide the correct answer. Also used in ClassicMode and IceMode for generating hints.

4.1.4 Solver:

Backtracking Algorithm (solveSudoku method):

- **Concept:** This is the core algorithm employed for solving the Sudoku puzzle. Backtracking is a general algorithmic technique for solving problems recursively by trying to build a solution incrementally. It explores possible solutions one by one, and if a solution cannot be completed, it "undoes" (backtracks) the last choice and tries another option.
- **Application in SudokuSolver.java:**
  - The solveSudoku method is a recursive function that attempts to fill the Sudoku grid starting from a given row and col.
  - **Base Case:** If the col index reaches Constants.BOARD_SIZE (meaning the end of a row), it moves to the next row. If row also reaches Constants.BOARD_SIZE (meaning all cells have been processed), it signifies a complete and valid solution, and the method returns true.
  - **Skipping Filled Cells:** If the current cell (grid[row][col]) is not empty, the solver simply moves to the next cell recursively.
  - **Trying Numbers:** For empty cells, it iterates through numbers from 1 to 9.
  - **Constraint Checking:** For each number, it uses the validator.isValid() method to check if placing that number at the current row, col is valid according to Sudoku rules (no duplicates in row, column, or 3x3 subgrid).
  - **Recursive Call:** If a number is valid, it places the number in the cell (grid[row][col] = num) and recursively calls solveSudoku for the next cell.
  - **Backtracking:** If the recursive call returns false (meaning no solution could be found with the current number or subsequent choices), it "backtracks" by resetting the current cell to 0 (grid[row][col] = 0) and tries the next number in the loop.
  - This process continues until a complete valid solution is found or all possibilities for a cell have been exhausted.

4.2 Time Complexity

4.2.1 Generator

Classic Puzzle Generation:

- **Backtracking Algorithm (generateCompleteSudoku and solveSudoku for initial board filling):**
  - **Time Complexity:** The theoretical worst-case time complexity for solving/generating an N×N Sudoku using backtracking can be considered $O(N^{N^2})$, as each of the N^2 cells could potentially try N numbers. However, due to aggressive pruning by the isValid checks, the practical average case is significantly faster. For a 9×9 board, while still exponential, it's efficient enough to run in milliseconds. A tighter, but still exponential, bound is often

given as O(N^K) where K is the number of empty cells, or even more specifically, O(9^{N^2}) in the absolute theoretical upper bound for a 9×9 grid without any filled cells.

- **Backtracking with Solution Counting (hasUniqueSolution and countSolutions):**
  - **Time Complexity:** Similar to the single solution backtracking, but potentially worse if there are many solutions to count before hitting the count > 1 early exit. In the worst case, it might explore a larger portion of the search space than simply finding one solution. Thus, it remains **exponential**, with a practical performance that depends on the number of non-empty cells and the "branchiness" of the solution space.

Ice Puzzle Generation:

- **Heuristic/Rule-based Cell Freezing:**
  - **Time Complexity:**
    - Iterating through cells to find candidates: O(N^2) (for each cell, checks a constant number of neighbors and simple conditions).
    - Shuffling candidateFrozenCells: O(M) where M is the number of candidate frozen cells (at most N^2), due to Fisher-Yates shuffle used by Collections.shuffle. So, O(N^2).
    - Selecting cells to freeze: O(cellsToFreeze), which is a percentage of N^2, so O(N^2).
  - **Overall Time Complexity for Freezing:** O(N^2).

4.2.2 Validator

Constraint Checking (isValid):

- **Time Complexity:** To check a single cell, it iterates through its row (O(N)), its column (O(N)), and its S×S subgrid (O(S^2), which is O(N) since S = $\sqrt{N}$).
- **Overall Time Complexity:** O(N).

Full Board Validation (isValidSudoku):

- **Time Complexity:** It iterates through all N^2 cells on the board. For each cell, it temporarily removes its value and then calls isValid (which is O(N)).
- **Overall Time Complexity:** O(N^2 × N) = O(N^3).

4.2.3 Hint System

Constraint Checking/Validation (canPlace method):

- **Time Complexity:** Same as isValid in the Validator, to check a single cell, it iterates through its row (O(N)), its column (O(N)), and its S×S subgrid (O(S^2): O(N).

Iterative Search / Brute-Force (with pruning):

- **findNakedSingle:**
  - **Time Complexity:** It iterates through all N2 empty cells. For each cell, it calls getCandidates. getCandidates iterates through 9 possible numbers, and for each, it calls canPlace (O(N)). So getCandidates is O(9×N)=O(N).
  - **Overall Time Complexity:** O(N^2×N)=O(N^3).
- **findHiddenSingle:**
  - **Time Complexity:** This method iterates through rows/columns/boxes (O(N) for each set, e.g., N rows). For each row/column/box, it iterates through all N numbers (1-9). For each number, it then iterates through the N cells in that row/column/box, calling canPlace (O(N)) for each check.
  - **Overall Time Complexity:** O(N×N×N×N)=O(N^4) (for each check like rows, columns, or boxes; total remains O(N^4)).

Prioritized Decision Making (getBestHint):

- **Time Complexity:** This algorithm's complexity is determined by the most complex hint generation method it calls. It calls findNakedSingle (O(N3)), then findHiddenSingle (O(N^4)), then getDirectHint (O(N^2)).
- **Overall Time Complexity:** O(N^4).

Random Selection (getDirectHint)

- **Time Complexity:** getEmptyCells iterates through all N2 cells to find empty ones (O(N^2)). Random selection from a list is O(1).
- **Overall Time Complexity:** O(N^2).

4.2.4 Solver

Backtracking Algorithm (solveSudoku method):

- **Time Complexity:** This is the same backtracking algorithm as used in the ClassicPuzzleGenerator for generating a complete board.
- **Overall Time Complexity:** The theoretical worst-case time complexity is **exponential**, often approximated as $O(N^{N^2})$, but in practice, due to effective pruning by the validator.isValid() checks, it solves most Sudoku puzzles very quickly (in milliseconds for a 9×9 board).

Chapter 5: Conclusion

5.1 Summary

This Sudoku game project successfully demonstrates the application of advanced software engineering principles in creating a comprehensive, extensible puzzle game. The implementation achieves several key objectives:

Technical Excellence: The codebase exemplifies SOLID principles through clean architecture, interface-based design, and clear separation of concerns. The MVC pattern enables maintainable code with low coupling between components.

Algorithmic Sophistication: The puzzle generation system employs optimized backtracking algorithms with constraint propagation, achieving efficient performance while maintaining solution uniqueness. The validation system provides O(1) constraint checking for real-time feedback.

Innovation in Game Design: The Ice Mode variant introduces novel mechanics that extend traditional Sudoku gameplay, demonstrating creativity within established frameworks while maintaining logical consistency.

User Experience Focus: The intuitive interface design, comprehensive hint system, and progressive difficulty scaling create an engaging experience that accommodates both beginners and expert players.

Extensible Architecture: The interface-based design enables easy addition of new game modes, solving techniques, and features without modifying existing code, demonstrating scalable software design.

5.2 Future Enhancements

Several opportunities exist for expanding the project's capabilities:

Additional Game Modes:

- **Killer Sudoku**: Implement cage-based sum constraints for additional complexity
- **Diagonal Sudoku**: Add diagonal constraint validation for increased difficulty
- **Multi-Grid Variants**: Develop Samurai Sudoku with overlapping grids
- **Timed Challenges**: Create speed-solving modes with leaderboards

Enhanced User Interface:

- **Modern UI Framework**: Migrate to JavaFX or web-based framework for richer interactions
- **Animation System**: Add smooth transitions and visual effects for state changes
- **Customization Options**: Implement user-configurable themes and color schemes
- **Accessibility Features**: Enhanced keyboard navigation and screen reader support

Advanced Features:

- **Puzzle Database**: Integration with online puzzle repositories
- **Solution Analysis**: Automated difficulty assessment based on required techniques
- **Machine Learning**: AI-powered hint generation using pattern recognition
- **Multiplayer Support**: Real-time collaborative or competitive gameplay

Technical Improvements:

- **Performance Optimization**: Multithreaded puzzle generation for improved responsiveness
- **Data Persistence**: Save/load functionality with progress tracking
- **Testing Framework**: Comprehensive unit and integration test coverage
- **Documentation**: Interactive code documentation with usage examples

---

References

1. Lafore, R. Data Structures and Algorithms in Java. Sams Publishing. Clear explanations and examples of fundamental DSA concepts applied in Java.

2. Malik, D. S. Data Structures Using Java. Cengage Learning. Covers key data structures and their implementations using Java.

3. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. Data Structures and Algorithms in Java. Wiley. Advanced topics in DSA with practical Java implementations.

4. Sudoku Game from Oakever: https://sudoku.com/sudoku-rules/

5. Build a Java Desktop Application - FreeCodeCamp

6. Link Github: https://github.com/29Schiller/Sudoku-DSA-Project

7. Link Slide: https://www.canva.com/design/DAGpfn2jWfc/4rDXJ3O5tMv-wKmzHMk_Nw/