Sudoku Solver Project Report

Yassin Es Saim - 22109654

Fall Semester 2024

Contents

1 Introduction **1**

2 Problem Statement **2**

3 Solution Design **2**
3.1 Deduction Rules 2
3.2 Grid Management 2
3.3 User Interaction 2

4 Design Patterns Used **2**
4.1 Singleton Pattern 3
4.2 Factory Pattern 3
4.3 Template Method Pattern 3
4.4 Observer Pattern 4
4.5 Pattern Interactions 4

5 Implementation Details **4**
5.1 File Handling 4
5.2 Solving Process 5

6 Challenges Encountered **5**
6.1 Technical Challenges 5
6.1.1 Candidate Management 5
6.1.2 DR3 (Locked Candidates) Implementation 6
6.2 Design Challenges 6
6.2.1 Pattern Integration 6
6.2.2 Error Handling 6
6.3 Performance Optimization 7
6.3.1 Grid Traversal 7

7 Conclusion **7**

8 References **8**

1 Introduction

The Sudoku Solver is a Java application that implements a solution for solving Sudoku puzzles using deduction rules and user interaction. The solver combines automated solving techniques with user input when needed, providing a flexible approach to puzzle solving.

2 Problem Statement

The project required implementing a Sudoku solver with the following key requirements:

- Implementation of three distinct deduction rules (DR1, DR2, DR3)
- Integration of user interaction when automated solving cannot progress
- Detection and handling of puzzle inconsistencies
- Implementation using multiple design patterns
- File-based input/output handling

3 Solution Design

3.1 Deduction Rules

Three deduction rules were implemented as subclasses of an abstract DeductionRule class:

1. **DR1 - Single Candidate**:
    - Identifies cells with only one possible candidate
    - Simplest and most basic rule
2. **DR2 - Hidden Singles**:
    - Finds values that can only appear in one position within a unit
    - Checks rows, columns, and boxes
3. **DR3 - Locked Candidates**:
    - Eliminates candidates based on box-line interactions
    - Most complex rule implemented

3.2 Grid Management

- Linear array representation (81 cells)
- Efficient candidate tracking system
- Cell state management with observer pattern

3.3 User Interaction

The solver includes:

- Manual input capability when deduction fails
- Move validation system
- Clear feedback on invalid moves
- Option to restart solving when inconsistencies occur

4 Design Patterns Used

The implementation utilizes four key design patterns to ensure a maintainable and extensible codebase:

4.1 Singleton Pattern

The Singleton pattern is implemented in two key components:

- **SudokuBoard**
  - Ensures only one instance of the game board exists
  - Provides global access point to the board state
  - Maintains consistency across the application
- **RuleFactory**
  - Centralizes rule creation logic
  - Ensures consistent rule instantiation
  - Manages rule creation throughout the solver

4.2 Factory Pattern

The Factory pattern is implemented through the RuleFactory class:

- **Purpose**
  - Creates appropriate deduction rule instances
  - Encapsulates rule instantiation logic
  - Provides flexibility for adding new rules
- **Benefits**
  - Centralizes rule creation
  - Makes rule management more maintainable
  - Simplifies rule addition and modification

4.3 Template Method Pattern

Implemented in the DeductionRule hierarchy:

- **Structure**
  - Abstract base class defines the algorithm structure
  - Concrete rule classes implement specific behaviors
  - Standardizes rule application process
- **Components**
  - DeductionRule: Abstract base class
  - DR1, DR2, DR3: Concrete implementations
  - Common progress tracking mechanism
- **Advantages**
  - Ensures consistent rule application
  - Reduces code duplication
  - Facilitates addition of new rules

4.4 Observer Pattern

Manages state changes and updates between components:

- **Implementation**
  - Cell class acts as the subject
  - CandidateManager implements observer interface
  - Automatic updates when cell values change
- **Benefits**
  - Maintains candidate list consistency
  - Reduces coupling between components
  - Automates update propagation
- **Usage**
  - Tracks cell value changes
  - Updates affected candidates
  - Maintains grid consistency

4.5 Pattern Interactions

The design patterns work together to create a cohesive system:

- **Singleton + Factory**
  - RuleFactory singleton manages rule creation
  - Ensures consistent rule instantiation
- **Template Method + Factory**
  - Factory creates rule instances
  - Template method standardizes their execution
- **Observer + Singleton**
  - Single board instance notifies observers
  - Maintains consistent game state

This combination of patterns creates a flexible and maintainable architecture that effectively supports the solver's functionality while allowing for future extensions and modifications.

5 Implementation Details

5.1 File Handling

The program handles input/output through:

```java
public class FileHandler {
    public static void loadFromFile(
            SudokuBoard board, String filename) {
        // Read comma-separated values
        // Convert 0 to -1 for empty cells
    }

    public static void saveToFile(
            SudokuBoard board, String filename) {
        // Convert grid to output format
        // Save to file
    }
}
```

5.2 Solving Process

The main solving algorithm:

```java
public class SudokuSolver {
    private List<DeductionRule> rules;

    public void solve(SudokuBoard board) {
        while (!board.isFull()) {
            boolean progress = false;
            for (DeductionRule rule : rules) {
                rule.apply(board);
                if (rule.hasMadeProgress()) {
                    progress = true;
                    break;
                }
            }
            if (!progress) {
                getUserInput(board);
            }
        }
    }
}
```

6 Challenges Encountered

6.1 Technical Challenges

6.1.1 Candidate Management

One of the most significant challenges was implementing efficient candidate management:

1. **Problem:**
    - Updating candidates after each move was computationally expensive
    - Need to track 9 possible values for each of the 81 cells
    - Required frequent updates and validations
2. **Solution:**
    - Implemented a dedicated CandidateManager class

- Used HashSet for efficient candidate storage and operations
- Implemented Observer pattern to automatically update candidates when cell values change
- Created an efficient update mechanism that only affects relevant cells

6.1.2 DR3 (Locked Candidates) Implementation

The implementation of the Locked Candidates rule presented complex challenges:

1. **Problem:**
   - Complex logic for identifying locked candidates
   - Need to check both box-line and line-box interactions
   - Required efficient iteration over rows, columns, and boxes
   - Difficulty in maintaining code readability while implementing complex logic
2. **Solution:**
   - Split implementation into box-to-line and line-to-box checks
   - Created helper methods for pattern detection
   - Implemented efficient elimination methods
   - Structured the code to clearly separate different aspects of the rule

6.2 Design Challenges

6.2.1 Pattern Integration

Integrating multiple design patterns required careful consideration:

1. **Challenge:**
   - Ensuring patterns worked together seamlessly
   - Avoiding pattern conflicts
   - Maintaining clean architecture
   - Balancing flexibility with complexity
2. **Solution:**
   - Clear separation of concerns between components
   - Well-defined interfaces for pattern interaction
   - Careful consideration of pattern responsibilities
   - Documentation of pattern relationships

6.2.2 Error Handling

Implementing robust error handling was crucial:

1. **Challenge:**
   - Detecting invalid states in the puzzle
   - Handling incorrect user input
   - Managing puzzle inconsistencies

- Providing meaningful feedback

2. **Solution:**
   - Comprehensive validation system for all moves
   - Clear error messages for users
   - Graceful error recovery mechanisms
   - State validation after each operation

6.3 Performance Optimization

6.3.1 Grid Traversal

Optimizing grid traversal operations was essential:

1. **Challenge:**
   - Frequent iteration over rows, columns, and boxes
   - Need for efficient access patterns
   - Memory usage concerns
   - Balance between speed and code clarity

2. **Solution:**
   - Used linear array representation for the grid
   - Implemented efficient index calculations
   - Cached frequently accessed data
   - Created specialized iterators for different traversal patterns

3. **Results:**
   - Significantly improved solving speed
   - Reduced memory overhead
   - Maintained code readability
   - Better overall system performance

These challenges and their solutions provided valuable learning experiences in software design and implementation.

7 Conclusion

The implemented Sudoku solver successfully meets all project requirements, providing:

- Effective automated solving through deduction rules
- Smooth integration of user interaction
- Robust error handling
- Clean, maintainable code structure

8 References

1. Design Patterns
   https://github.com/CodeJamm/JAVA_Design_Patterns

2. Sudoku Solving Techniques
   https://sudokusolver.app/