CS611 Assignment Sliding Puzzle Game Design Documentation

Name: JIA,HAOZHE BU ID:U64160841

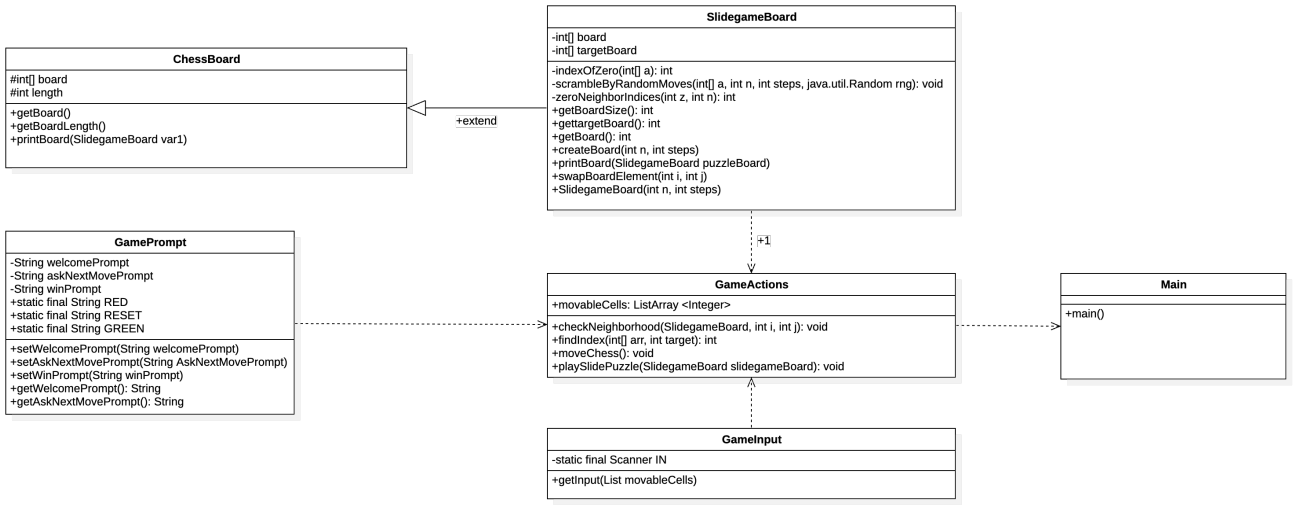
September 19, 2025

Contents

1 Overview

The `CS611_SlidingPuzzle_hw1` project is a console-based sliding puzzle game implemented in Java. The game allows players to solve an $n \times n$ sliding puzzle by moving numbered tiles into an empty space until the board matches the target configuration.

2 Class Structure and UML Design



3 Design Architecture

3.1 Core Components

3.1.1 Abstract Base Class - ChessBoard

- **Purpose**: Provides the foundational structure for any board-based game
- **Key Features**:
  - Abstract methods for board operations
  - Protected board array and length properties
  - Ensures consistent interface for different board implementations

3.1.2 Concrete Implementation - SlidegameBoard

- **Purpose**: Implements the sliding puzzle game logic
- **Key Features**:
  - Board creation and initialization
  - Random scrambling algorithm
  - Board state management (current vs. target)
  - Visual board representation

3.1.3 Game Control Classes

- **GamePrompt**: Manages all user interface messages and styling
- **GameInput**: Handles user input validation and processing
- **GameActions**: Coordinates game mechanics and move validation

3.2 Key Design Patterns

3.2.1 Template Method Pattern

The abstract `ChessBoard` class defines the template for board operations, allowing different implementations while maintaining a consistent interface.

3.2.2 Single Responsibility Principle

Each class has a distinct responsibility:

- `SlidegameBoard`: Board state and logic
- `GamePrompt`: User interface messaging
- `GameInput`: Input validation
- `GameActions`: Game flow control

3.2.3 Static Utility Pattern

Several classes use static methods for utility functions, reducing object instantiation overhead and providing shared functionality.

4 Scalability and Extensibility

4.1 Scalability Features

1. **Dynamic Board Sizing**: The system supports any $n \times n$ board size where $n \geq 2$
2. **Efficient Algorithms**:
   - $O(1)$ neighbor checking
   - Optimized scrambling algorithm
   - Memory-efficient board representation
3. **Configurable Difficulty**: The scrambling process uses a configurable step count for varying difficulty levels

4.2 Extensibility Features

1. **Abstract Base Class**: The `ChessBoard` abstraction allows for easy implementation of other board-based games
2. **Modular Input System**: `GameInput` can be extended to support different input methods (keyboard, mouse, network)
3. **Flexible UI System**: `GamePrompt` supports customizable messages and color schemes
4. **Pluggable Game Logic**: `GameActions` can be extended for different game rules or mechanics

4.3 Future Extension Points

```
// Example extensions:
public class HexagonalBoard extends ChessBoard { ... }
public class NetworkGameInput extends GameInput { ... }
public class GraphicalGamePrompt extends GamePrompt { ... }
```

Listing 1: Example Extensions

5 Architecture Benefits

5.1 Maintainability

- Clear separation of concerns
- Well-defined interfaces between components
- Consistent naming conventions

5.2 Testability

- Modular design allows for unit testing of individual components
- Static methods facilitate testing without object state dependencies
- `Test` class demonstrates debugging capabilities

5.3 Performance

- Efficient algorithms for board manipulation
- Minimal object creation during gameplay
- Optimized memory usage with array-based board representation

6 Technical Implementation Details

6.1 Board Representation

- Uses 1D integer array for 2D board representation
- Empty space represented as 0
- Efficient index calculations: $\text{position} = \text{row} \times \text{length} + \text{column}$

6.2 Move Validation

- Neighbor checking algorithm validates moves in $O(1)$ time
- Maintains list of valid moves for user selection
- Prevents invalid moves through input validation

6.3 Game State Management

- Maintains both current and target board states
- Uses array comparison for win condition checking
- Supports board state restoration and manipulation

7 Algorithm Complexity Analysis

Operation	Time Complexity	Space Complexity
Board Creation	$O(n^2)$	$O(n^2)$
Move Validation	$O(1)$	$O(1)$
Board Scrambling	$O(k)$	$O(1)$
Win Condition Check	$O(n^2)$	$O(1)$
Board Display	$O(n^2)$	$O(1)$

Table 1: Algorithm Complexity Analysis

Where n is the board dimension and k is the number of scrambling steps.

8 Conclusion

The sliding puzzle game demonstrates a well-architected, object-oriented solution that balances simplicity with extensibility. The design supports both horizontal scaling (larger boards) and vertical scaling (additional game features) while maintaining clean code principles and efficient performance characteristics.

The modular architecture ensures that future enhancements can be added with minimal impact on existing code, making this a robust foundation for game development projects. The use of design patterns such as Template Method and Single Responsibility Principle provides a solid foundation for maintainnable and extensible code.

Key strengths of the architecture include:

- **Separation of Concerns**: Each class has a well-defined responsibility
- **Extensibility**: Abstract base classes and modular design support future enhancements
- **Performance**: Efficient algorithms with optimal time complexity
- **Maintainability**: Clean code structure with consistent patterns

This design provides an excellent foundation for building more complex board-based games while maintaining code quality and architectural integrity.