

**NumJig Project**

**Submitted by:**

**Haseeb ullah F20232661009**

**Abdullah abid F2023266735**

**Submitted to:**

**M. Owais Khan**

**Project Report:**

**Data Structure**

**Section:**

**V8**

**NumJig Game Code Documentation**

**Table of Contents**

1. [Introduction](#introduction)
2. [Extreme Programming (XP)](#extreme-programming-xp)
3. [Methodology](#methodology)
   * [3.1 Requirement Gathering](#31-requirement-gathering)
   * [3.2 Iteration Planning](#32-iteration-planning)
   * [3.3 Design and Prototyping](#33-design-and-prototyping)
   * [3.4 Development and Iteration](#34-development-and-iteration)
   * [3.5 Testing and Continuous Integration](#35-testing-and-continuous-integration)
   * [3.6 Deployment and Feedback](#36-deployment-and-feedback)
4. [Functional and Non-Functional Requirements](#functional-and-non-functional-requireme)
5. [Work Breakdown Structure (WBS)](#work-breakdown-structure-wbs)
6. [Schedule and Gantt Chart](#schedule-and-gantt-chart)
7. [Diagrams](#diagrams)
   * [7.1 ERD Diagram](#71-erd-diagram)
   * [7.2 Class Diagram](#72-class-diagram)
   * [7.3 Sequence Diagram](#73-sequence-diagram)
   * [7.4 Use Case Diagram](#74-use-case-diagram)
   * [7.5 DFD Diagram](#75-dfd-diagram)
8. [Test Cases](#test-cases)
9. [Conclusion](#conclusion)

**1. Introduction**

This game is a console-based picture puzzle implemented in C++. The game leverages dynamic data structures—such as doubly linked lists and stacks—to manage the puzzle board state and supports user interactions (via keyboard input) to move pieces on the board. The board is rendered using ASCII art, and the gameplay involves swapping pieces until a target numerical sequence is achieved.

**2. Extreme Programming (XP)**

* **Iterative Development:** The code is designed for rapid iterations, with separate modules for UI, game logic, and data handling.
* **Continuous Feedback:** By testing moves, rendering, and game state updates continuously, the design allows for swift modifications based on tester and stakeholder feedback.
* **Pair Programming & Simplicity:** The modular structure promotes simplicity and ease of understanding, essential for pair programming and agile adjustments.

**3. Methodology**

**3.1 Requirement Gathering**

* **Functional:**
  + Display an interactive game board using ASCII art.
  + Process user input for moving puzzle pieces.
  + Update game state dynamically (score, moves, and sequence checking).
* **Non-Functional:**
  + Fast, responsive console output.
  + Clear and maintainable code structure.
  + Minimal resource usage.

**3.2 Iteration Planning**

* **Core Modules:** UI functions (e.g., gotoXY, textColor), data structures (Player, Node, Stack, doublyLinkedList), and game logic (GameLogics).
* **Iterations:** First build the UI and data structures; next, integrate movement and input processing; finally, add undo functionality and win/lose conditions.

**3.3 Design and Prototyping**

* **Prototype:** Use simple ASCII-based interfaces to simulate the game board.
* **Key Designs:**
  + A doubly linked list represents the puzzle pieces.
  + A stack is used to implement the undo feature.
  + The GameLogics class integrates rendering, input handling, and game state validation.

**3.4 Development and Iteration**

* **Data Structures:** Implementation of Player, Node, NodeS, and Stack to encapsulate game state.
* **UI Functions:** Utility functions (gotoXY, textColor, showCursor) manage console output.
* **Game Flow:** The game loop in GameLogics::Play() continuously renders the board, processes moves, and checks game conditions.

**3.5 Testing and Continuous Integration**

* **Unit Tests:** Validate individual functions such as Swap, indexAddress, and searchZero.
* **Integration Tests:** Run the full game loop to ensure smooth transitions between welcome screen, game play, and win/lose states.
* **Continuous Integration:** Automated builds and manual testing on target console environments.

**3.6 Deployment and Feedback**

* **Deployment:** The game is deployed as a console application.
* **Feedback:** User testing focuses on input responsiveness, clarity of ASCII graphics, and overall gameplay experience. Iterative changes are made based on user feedback.

**4. Functional and Non-Functional Requirements**

* **Functional Requirements:**
  + **Game Board Display:** Render a board with score and puzzle pieces.
  + **Movement Logic:** Process valid moves using arrow keys and update the board accordingly.
  + **Undo Operation:** Allow players to reverse their previous moves.
  + **Win/Lose Conditions:** Trigger appropriate endgame screens when the puzzle is solved or the score is depleted.
* **Non-Functional Requirements:**
  + **Performance:** Immediate feedback in the console.
  + **Maintainability:** Code is modular and well-commented to facilitate future updates.
  + **Usability:** Clear visual output and intuitive keyboard controls.

**5. Work Breakdown Structure (WBS)**

* **Module 1:** **UI Rendering**
  + Functions: gotoXY, textColor, showCursor
* **Module 2:** **Data Structures**
  + Classes: Player, Node, NodeS, Stack, doublyLinkedList, Graph, vectors
* **Module 3:** **Game Logic**
  + Methods: opreate, Swap, checkSequence, searchZero
* **Module 4:** **Game Flow Control**
  + Class: GameLogics
  + Method: Play
* **Module 5:** **Application Entry**
  + Function: main

**6. Schedule and Gantt Chart**

A simplified schedule for the project might look like:

* **Week 1:** Requirement Gathering and Prototyping
* **Week 2:** Develop UI functions and basic data structures
* **Week 3:** Implement core game logic and input handling
* **Week 4:** Integrate undo functionality, perform testing, and refine UI

*(A detailed Gantt chart can be created using project management software if needed.)*

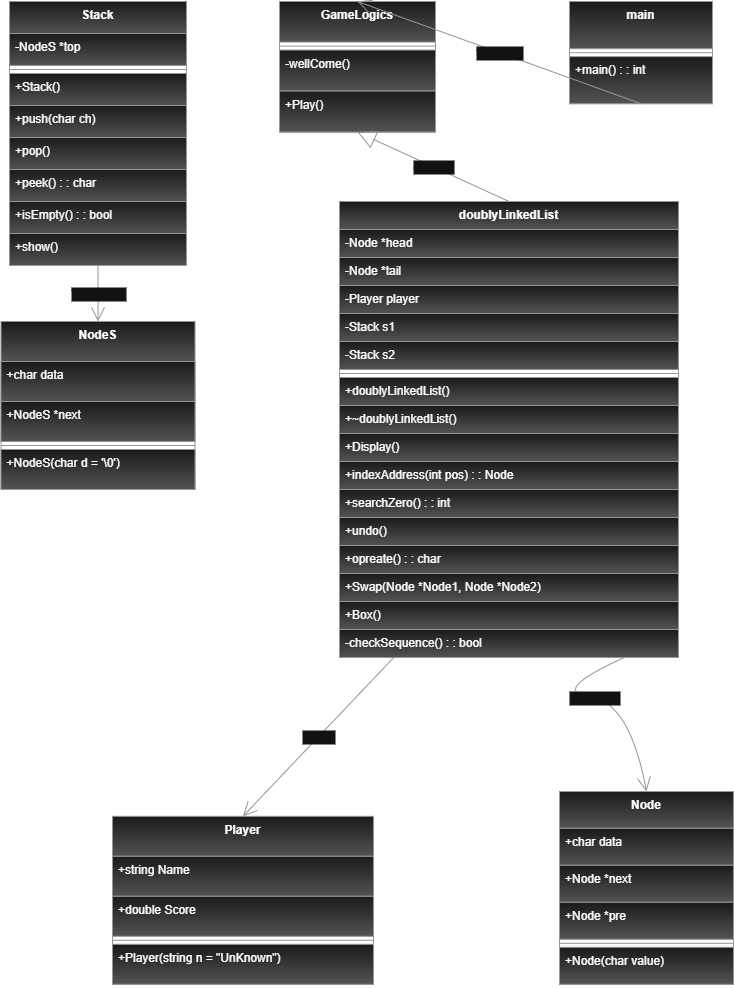
**7. Diagrams**

**7.1 ERD Diagram**

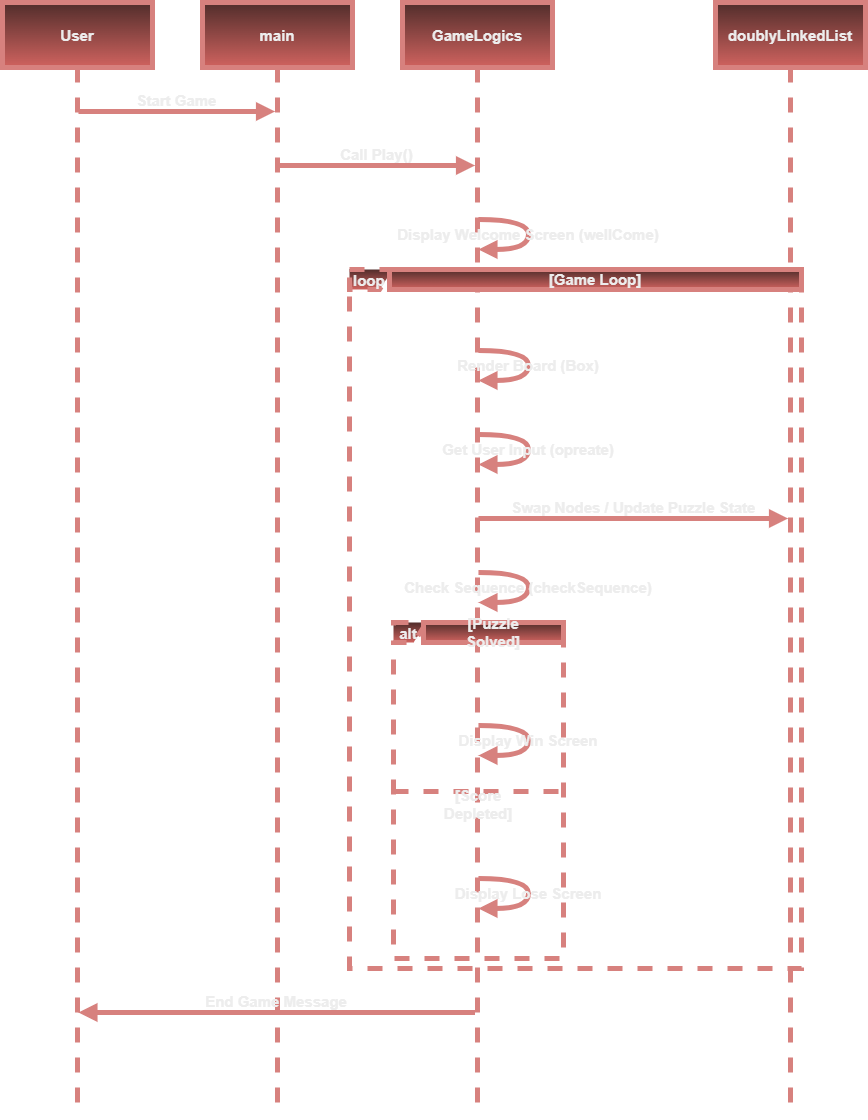
*Note: As this is a console game with in-memory data structures rather than persistent storage, the ERD is conceptual.*

* **Entities:**
  + **Player:** Represents a game participant.
  + **Node/NodeS:** Represent elements of the puzzle.
  + **Stack:** Manages the undo history.
* **Relationships:**
  + The doublyLinkedList holds multiple Node instances.
  + The Stack manages NodeS objects for undo operations.
  + GameLogics integrates these components to drive game play.

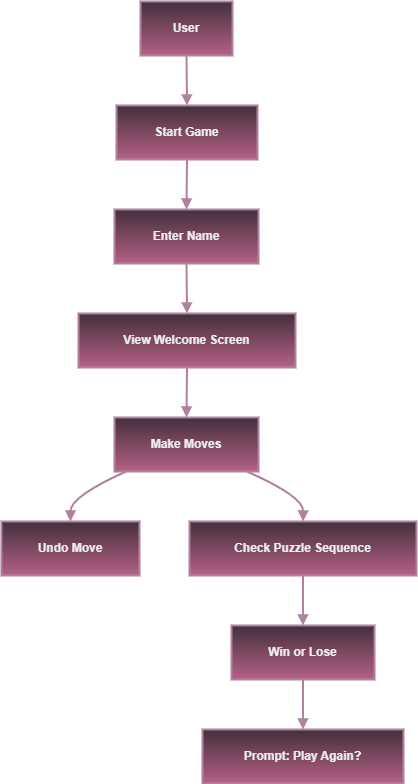
**7.2 Class Diagram**

****

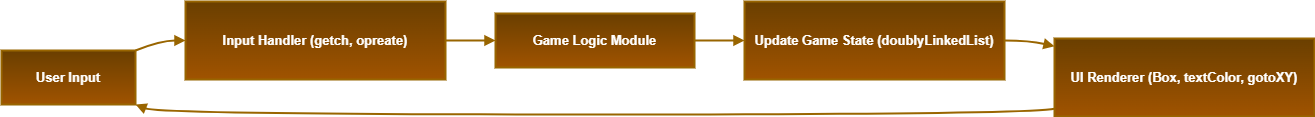
**7.3 Sequence Diagram**

****

**7.4 Use Case Diagram**



**7.5 DFD Diagram**



## 8. Test Cases

| **Test Case ID** | **Input** | **Expected Result** |
| --- | --- | --- |
| **1: Valid Move** | Press an arrow key corresponding to a valid move. | The board updates with swapped nodes; score decrements appropriately. |
| **2: Invalid Move** | Press an arrow key that would cause an out-of-bound move. | Display message "Move not allowed" and no state change. |
| **3: Undo Operation** | Press the 'u' key after making a move. | The last move is reversed and the previous board state is restored. |
| **4: Winning Sequence** | Series of valid moves that arrange the puzzle in the target sequence. | Trigger the win screen with the message "YOU WIN". |
| **5: Losing Condition** | Repeated moves that deplete the player’s score. | Trigger the lose screen with the message "YOU LOSE". |

**9. Conclusion**

This documentation presents an in-depth look at the architecture and design of a console-based picture puzzle game implemented in C++. By employing a modular design using classes such as Player, Node, Stack, and GameLogics, the game demonstrates sound design principles, clean separation of concerns, and robust handling of user input. The accompanying Mermaid diagram and detailed documentation clearly illustrate both the technical depth and high-level overview—making it accessible for both technical reviewers and non-technical stakeholders alike.