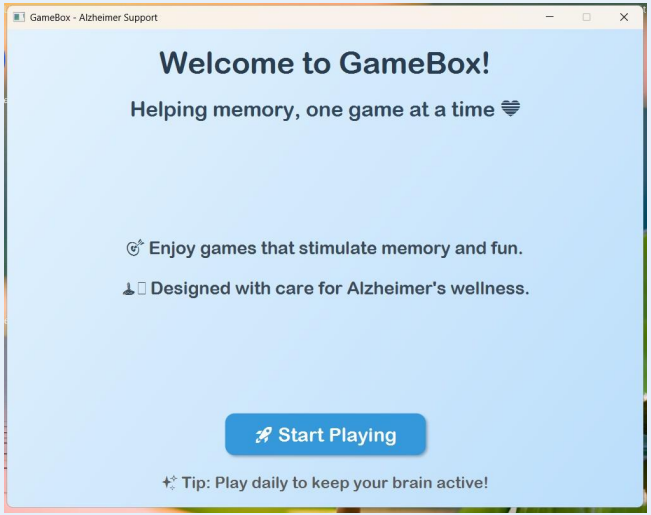
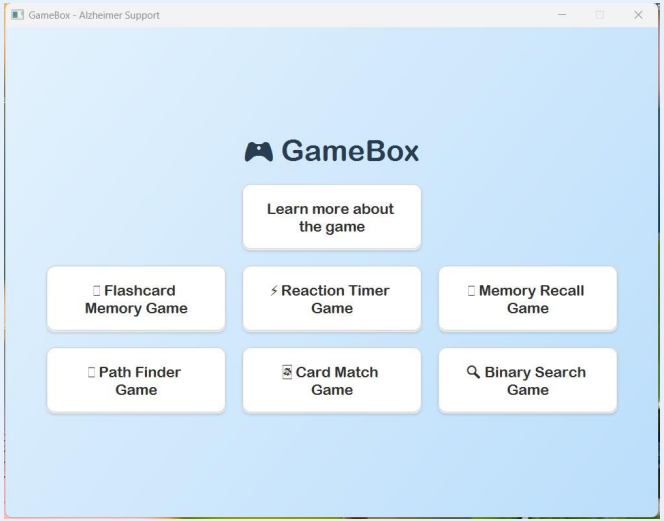
Problem Description





GameBox is a suite of engaging mini-games designed to support memory and cognitive training of Alzheimer Patients. The idea was inspired by the increasing use of game-based learning for neurological reinforcement. Our goal was to build a modular system of games, each targeting a different data structure and memory-enhancing pattern.

GameBox Analysis

User Need: Individuals with **Alzheimer's disease** and other cognitive impairments often benefit from simple, consistent mental stimulation. Traditional therapy can be repetitive or passive interactive games offer a more engaging and accessible alternative.

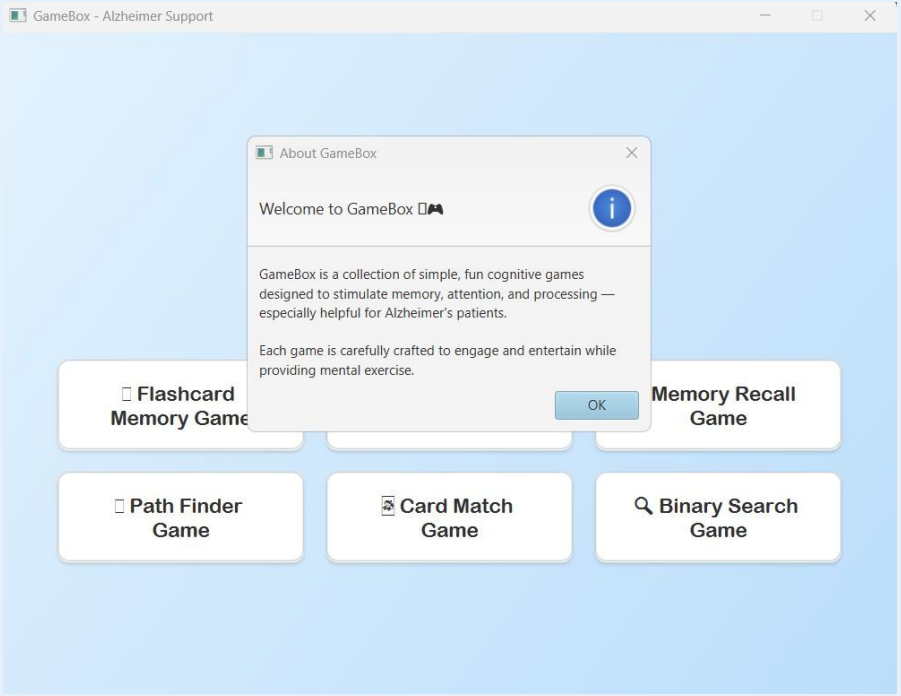
Why This Matters: Alzheimer's patients struggle with memory retention, attention span, and recall.Flashcard-style reinforcement, matching exercises, and quick-response activities are clinically proven to assist with cognitive rehabilitation.

Motivation: We wanted to build an educational and therapeutic tool that combines core Data Structures & Algorithms with user-friendly, game-based interaction. GameBox allows users to challenge themselves through logic and memory games while also showcasing our knowledge of OOP and JavaFX.

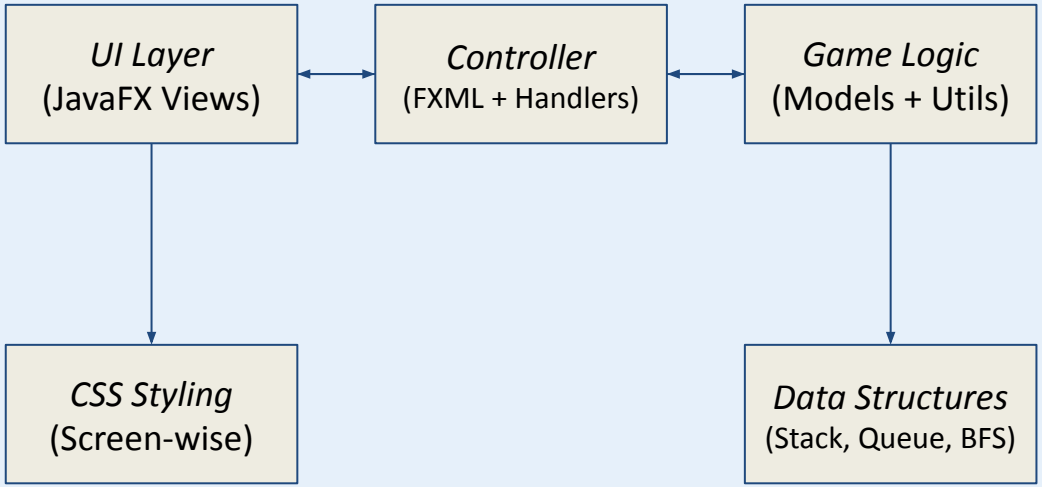
Design Choice: Clean UI, minimal distractions, consistent controls all tailored for ease of use in cognitively sensitive environments.

Research Insights:

- Reaction training supports mental agility.
- Flashcards are an effective reinforcement method.
- Path finding and sorting mimic real-world problem solving.



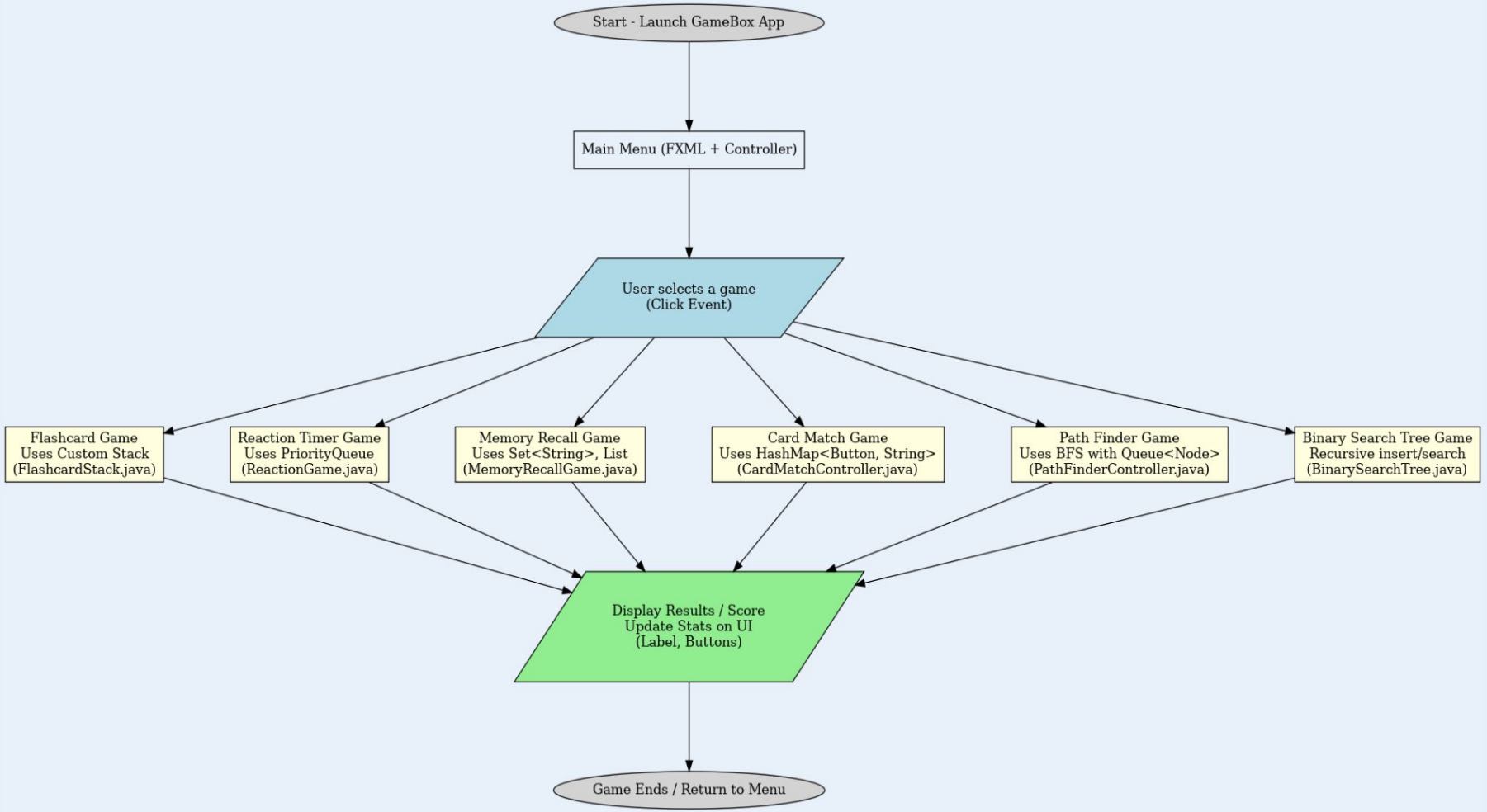
System Design



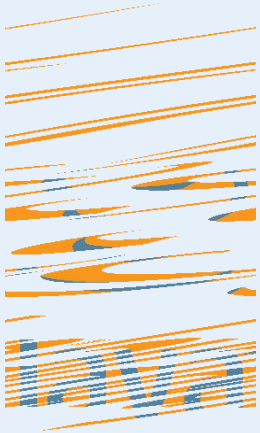
Design Patterns Used

- **MVC** (Model-View-Controller) – for clear separation of UI, logic, and control.
- **Factory Pattern** (optional mention if used for game switching).
- **Stack/Queue/BFS** – used as **core algorithms/data structures per game**.

Flow Chart

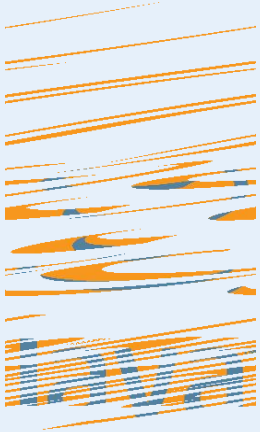

Start - Launch GameBox App
Main Menu (FXML + Controller)
User selects a game
(Click Event)
Flashcard Game
Uses Custom Stack
(FlashcardStack.java)
Reaction Timer Game
Uses PriorityQueue
(ReactionGame.java)
Memory Recall Game
Uses Set<String>, List
(MemoryRecallGame.java)
Card Match Game
Uses HashMap<Button, String>
(CardMatchController.java)
Path Finder Game
Uses BFS with Queue<Node>
(PathFinderController.java)
Binary Search Tree Game
Recursive insert/search
(BinarySearchTree.java)
Display Results / Score
Update Stats on UI
(Label, Buttons)
Game Ends / Return to Menu

Implementation (APIs & Tools)



- **Java Version:** Java 21
- **UI Framework:** JavaFX SDK 21
- **IDE:** Eclipse
- **Animation/Timers:** PauseTransition, System.currentTimeMillis()
- **Custom DS:** Manual stack via LinkedList, BST via Node/Tree structure
- **FXML & MVC:** Each game follows a separate controller-view split

Implementation (Modular APIs)



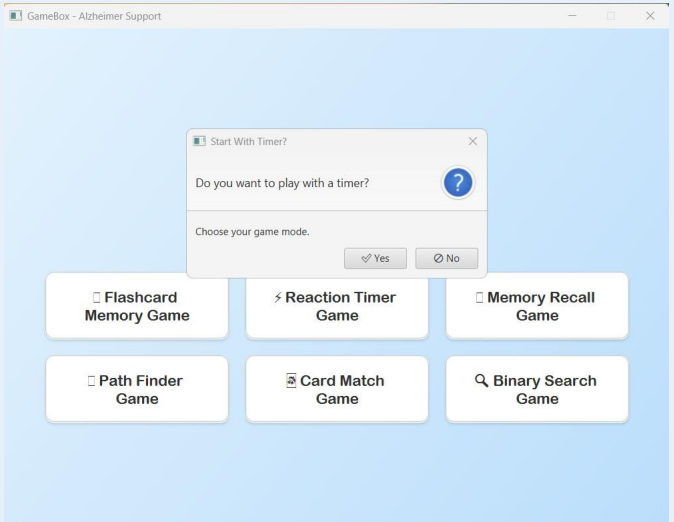
- **Initialize Game State**
  - public void reset() – resets state, rounds, moves, etc.
- **Push/Pop Game Data**
  - public void push(T item) / public T pop() – custom FlashcardStack<T>
- **Run Core Logic**
  - public List<int[]> findShortestPathBFS() – PathFinder logic
  - public void recordReaction(long ms) – ReactionGame logic
- **Retrieve Game Stats**
  - public int getCurrentRound(), public PriorityQueue<Long> getTopReactions()

Timer Mode Selection – Game Mode Entry Point

- Ensures dynamic difficulty adjustment based on user preference.
- Adds replay value by enabling both practice and performance play styles.
- Supports accessibility for users who may want more relaxed gameplay.

UI Breakdown:

- **Purpose**: "Start With Timer?" – Helps players decide whether to play in a timed or untimed environment..
- **Yes Button**: Starts the game in **timed mode**.
- **No Button**: Launches the game without time constraints for relaxed play.

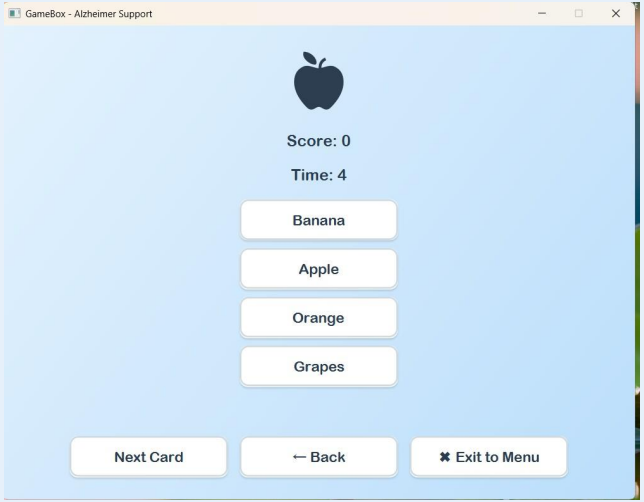


FlashCard Memory Game

- Displays an image (emoji or icon) and provides 4 options to guess the correct label.
- Randomly selects flashcards from a list and tracks the user's answers and score.
- Timed and untimed modes supported for flexible gameplay.

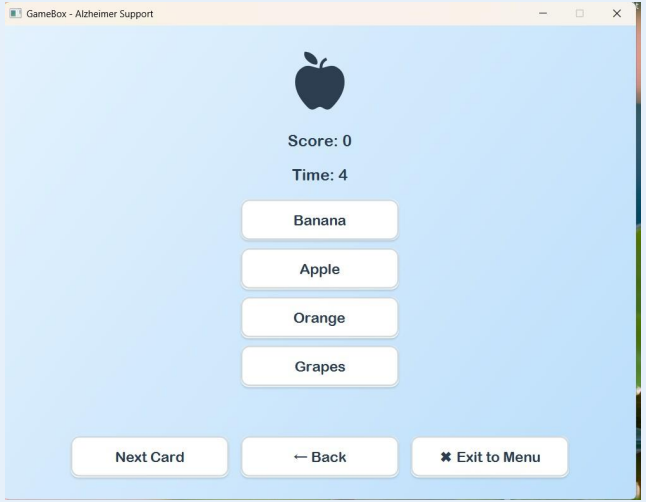
UI Breakdown:

- **Emoji Icon**: Displays current Flashcard visual symbol.
- **Options**: Four clickable answer buttons to choose the matching label.
- **Score/Timer**: Updates live based on game mode and answers.
- **Navigation**: NextCard button to progress, Back and Exit to Menu for navigation controls



FlashCard Memory Game Algorithm

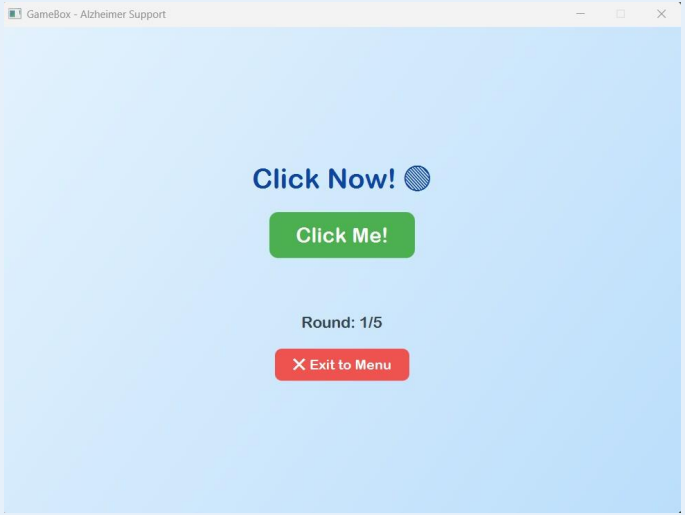
- **Initialize Flashcards**
  - Create a set of flashcards each with:
    - An **emoji**
    - A **correct label**
    - 3 incorrect options
  - Shuffle the list.
- **Start Game**
  - Display the emoji for the current flashcard.
  - Randomly shuffle and show 4 options (1 correct + 3 distractors).
- **User Makes a Guess**
  - User clicks one of the four buttons.
  - Check if the selected option matches the correct label.
- **Provide Feedback**
  - If correct → Show ✅ and increase score.
  - If incorrect → Show ❌ with correct answer.
  - Disable options until "Next" is clicked.
- **Navigate Cards**
  - Use **Next** to go forward.
  - Use **Back** to review previous cards (non-destructive).
  - Optional: track time with timer.
- **Repeat Until End**
  - Loop continues through all flashcards.
  - Score and time are tracked.



Reaction Game

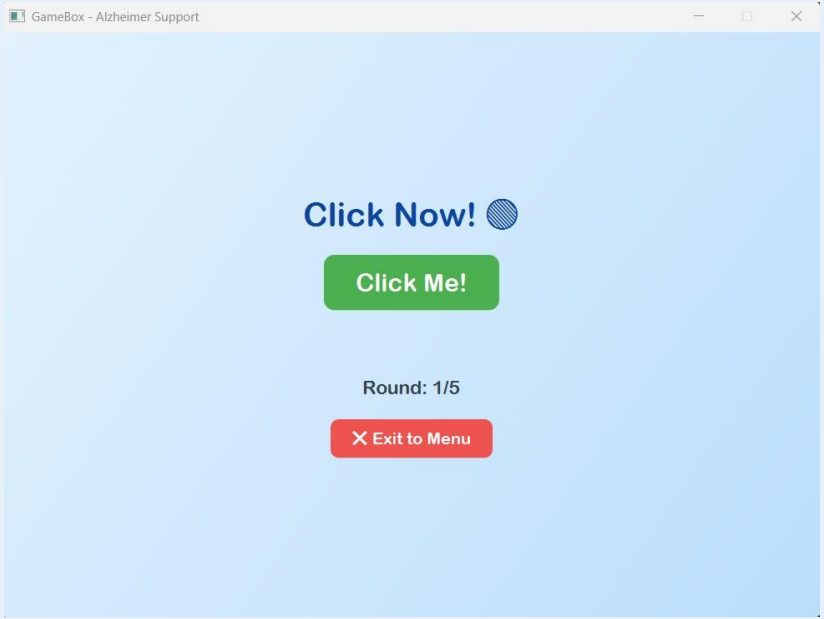
- Game Logic: Players react to visual signals; scores are based on response time
- Backend: Uses PriorityQueue to store top scores and Queue to manage rounds
- Incorporates JavaFX Timers and asynchronous event handling

UI Breakdown:
- Visual Trigger: Click Now! prompt
- Timer starts and button appears
- Click Me!: Captures response time
- Round tracker + Exit button



Reaction Game Algorithm

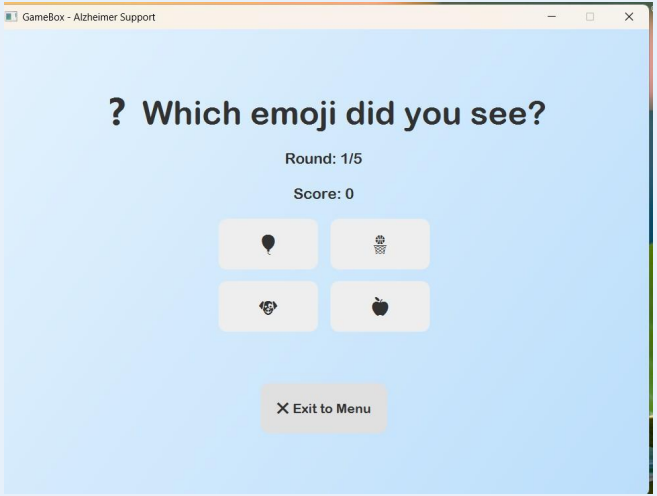
1. **Initialize Game**
   - Set total rounds (e.g., 5).
   - Disable the "Click Me" button initially.
2. **Start Each Round**
   - Show "Get ready..." message.
   - Wait for 1–3 seconds randomly using a pause transition.
   - Enable the "Click Me" button and show "Click Now!" message.
   - Record the start time.
3. **Player Clicks Button**
   - Calculate reaction time: current time - start time.
   - Store reaction time in a priority queue.
   - Show the reaction time for this round.
4. **Repeat Until All Rounds Are Completed**
   - After each round, repeat step 2 until all rounds are done.
5. **Show Final Results**
   - Display the top 3 fastest reaction times.
   - Show buttons to "Play Again" or "Exit to Menu".



Memory Recall Game

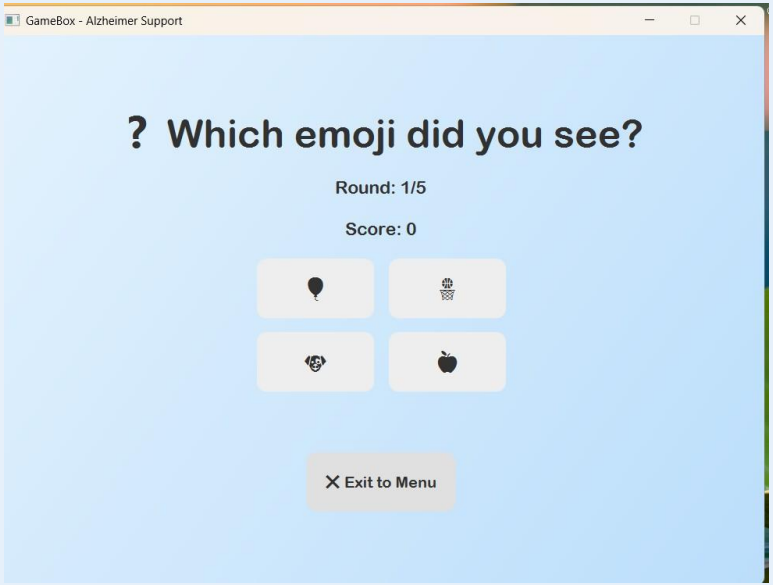
- Game Logic: Displays emojis for memorization, then asks players to recall them
- Backend: Set<String> for uniqueness and List<String> for recall options
- Encourages pattern recognition and memory under time constraints

UI Breakdown:
- Prompt: Memory challenge question
- Four emojis: Choose the correct answer
- Round/Score display
- Exit to Menu: Navigation



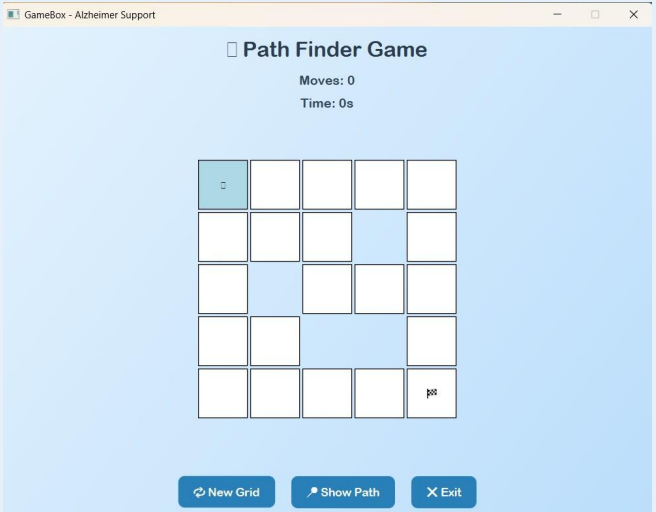
Memory Recall Game Algorithm

- **Initialize Game**
  - Set total rounds (e.g., 5).
  - Define a list of emojis as the memory pool.
  - Reset score and round count.
- **Start Each Round**
  - Randomly show one emoji to the user for 2 seconds.
  - Hide the emoji and display 4 options (1 correct + 3 random).
  - User selects the option they remember seeing.
- **Check User's Answer**
  - If correct → increase score and show ✅.
  - If incorrect → show ❌ feedback.
  - Move to the next round after a brief delay.
- **Repeat**
  - Continue until all rounds are completed.
- **Game Over**
  - Show final score (e.g., 3/5).
  - Offer options to replay or return to main menu.



Path Finder Game

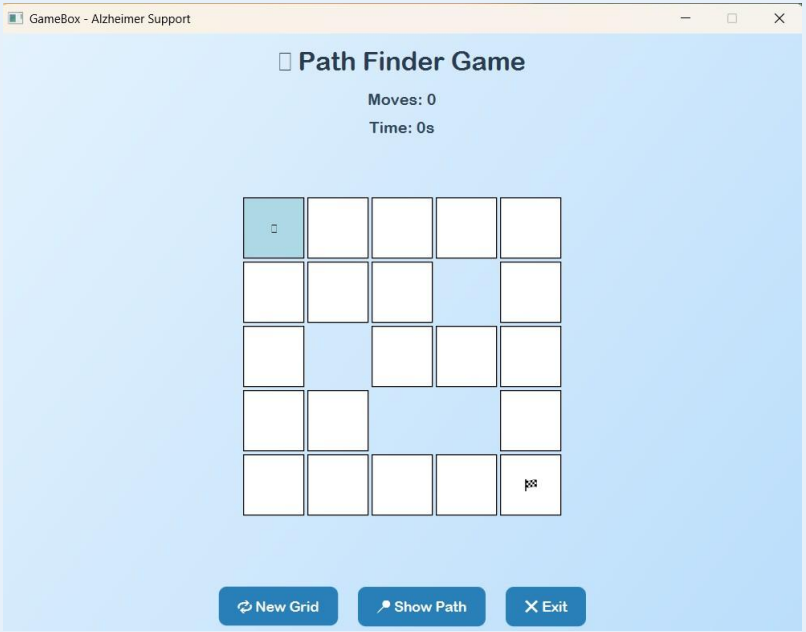
- Game Logic: Navigate a 5x5 grid to find the shortest path
- Backend: Implements BFS using Queue<Node>, models the grid as an unweighted graph
- Uses grid traversal with node connectivity logic



UI Breakdown:
- 5x5 Grid: Represents a graph
- Empty/Blocked tiles visually represented
- Buttons: Generate new grid, show path, exit

Path Finder Game Algorithm

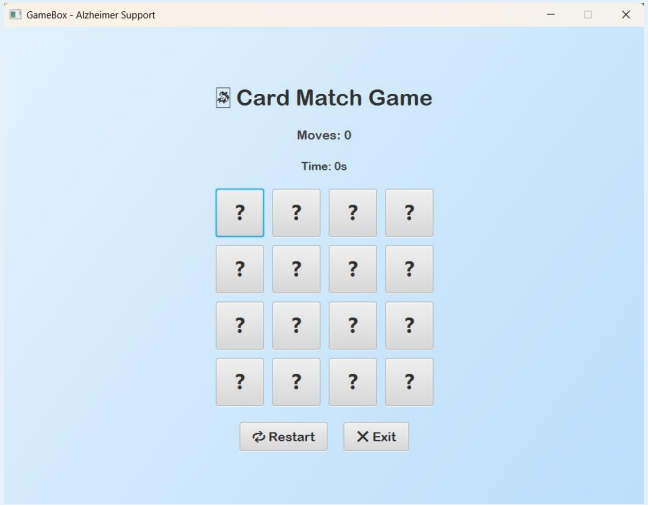
1. **Initialize Grid**
   Create a 5x5 grid and randomly place obstacles (grey blocks). Start (top-left) and goal (bottom-right) are always open.
2. **Player Starts at (0, 0)**
   Show the player icon and highlight accessible cells.
3. **Player Clicks a Cell**
   - If the cell is adjacent and not blocked, move the player.
   - Increase the move counter.
   - Update the visual grid.
4. **Check for Goal**
   - If the player reaches (4, 4), trigger the success dialog.
   - Show time (if timer is enabled) and compare moves to optimal path.
5. **Find Shortest Path (BFS Algorithm)**
   - Use Breadth-First Search to calculate and store the shortest valid path.
   - Optionally highlight this optimal path in green.
6. **Restart or Exit Option**
   After completion, allow the user to start a new game or return to the main menu.



Card Match Game

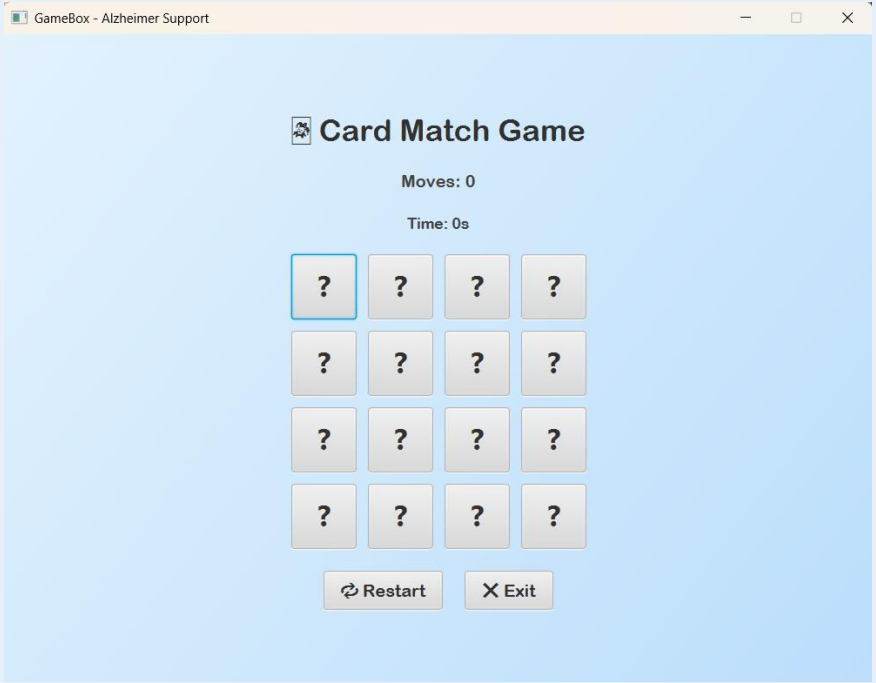
- Game Logic: Players flip cards to find matching emoji pairs
- Backend: Uses HashMap<Button, String> for card mappings and List for emoji storage
- Shuffles content and dynamically updates the board on each turn

UI Breakdown:
- Grid: 16 hidden cards to match
- Move/Time Tracker: Updates in real-time
- Restart/Exit: Game control buttons



Card Match Game Algorithm

- **Initialize Game Board**
  - Create 8 unique emoji pairs (total 16 cards).
  - Shuffle all cards and place them face-down on a 4x4 grid.
- **User Interaction Begins**
  - User clicks a card → it reveals the symbol.
  - Wait for the second card to be clicked.
- **Check Match**
  - If the two cards match:
    - Keep them face-up.
    - Increment move count.
  - If they don’t match:
    - Wait briefly → flip both back.
    - Increment move count.
- **Repeat Until All Pairs Are Matched**
  - Track total moves and optional timer.
  - When all cards are revealed → show success dialog.
- **Post-Game Options**
  - Offer user choice to:
  - Restart the game
  - Return to Main Menu

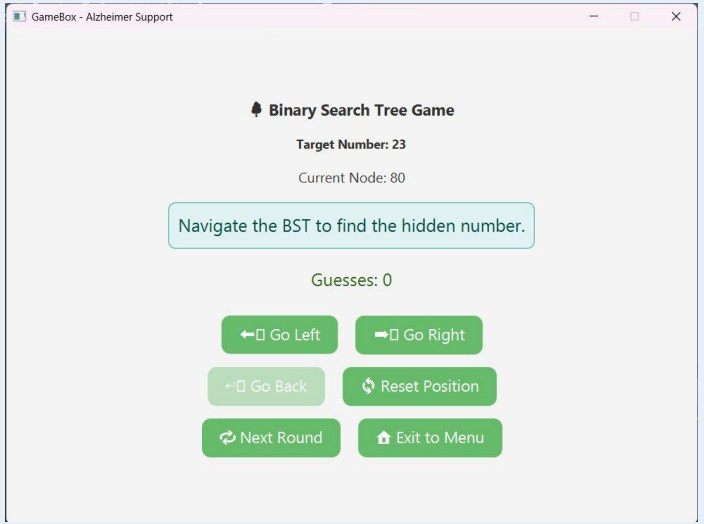


Binary Search Tree Game

- **Game Logic:** Players navigate a randomly generated Binary Search Tree (BST) to find a hidden target number using left/right decisions. The number of guesses is tracked and feedback is given based on node comparisons.
- **Backend:** Implements a custom TreeNode class and builds a BST from a randomized list of integers. Uses a **Stack** to manage backtracking history, and **Recursion** for tree construction and path-finding.
- **Educational Value:** Reinforces understanding of **BST traversal**, **comparative logic**, **recursion**, and **stack-based navigation**. Tree structure and path to the target are printed to the console.

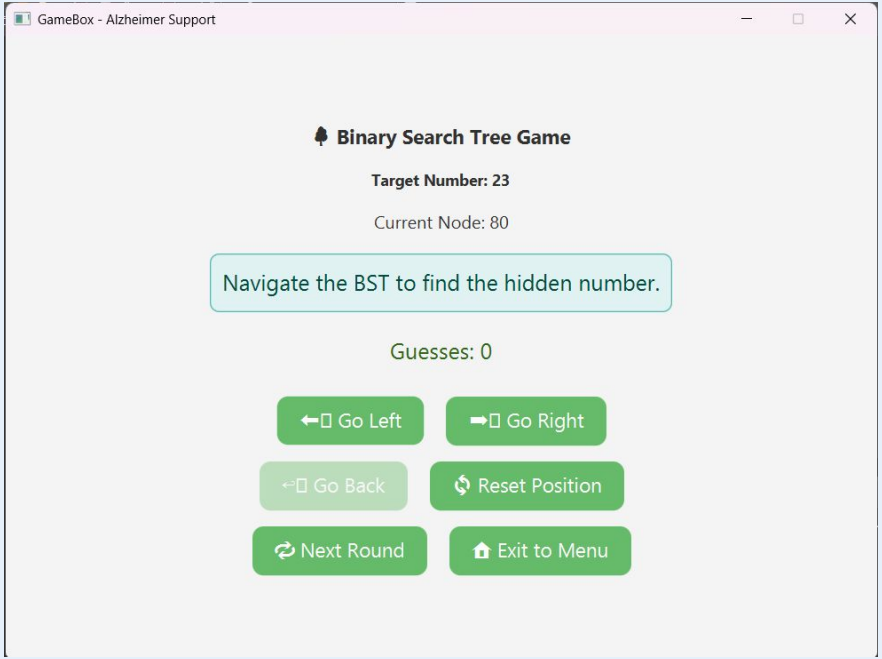
UI Breakdown:

- **Left / Right Buttons:** Navigate the BST (disabled if no child exists)
- **Back Button:** Moves user to the previously visited node using stack
- **Reset Position:** Resets position to root, clears stack
- **Guess Counter:** Tracks number of steps taken
- **Exit Button:** Returns to the GameBox main menu



Binary Search Tree Game Algorithm

1. **Initialize Game:**
   - Generate 10 unique random numbers.
   - Build a Binary Search Tree (BST) from them.
   - Randomly select one number as the *target*.
2. **Start at Root:**
   - Set the current node to the root of the BST.
3. **Repeat Until Target Found:**
   - Compare target with current node:
     - If equal → ✅ Target found → Show success message.
     - If smaller and left child exists → move to left child.
     - If greater and right child exists → move to right child.
     - If desired direction doesn't exist → show “Dead End” message and suggest back or reset.
4. **Allow Navigation Support:**
   - Back: Move to previous node (via history stack).
   - Reset: Return to root and clear history.
   - Next: Start a new game with new numbers.



Results

Impact for Alzheimer's Patients

- **Improved Visual Memory** – Flashcard & Card Match games strengthen recall.
- **Cognitive Speed** – Reaction Game enhances response time.
- **Spatial Navigation** – Path Finder trains directional awareness.
- **Sequencing & Recall** – Alphabet Train (if added) supports step-by-step memory.

Technical Outcome

- MVC architecture implemented
- Custom reusable data structures: FlashcardStack<T>, Queue, PriorityQueue
- Game logic and UI cleanly decoupled
- All games follow consistent design guidelines

Discussions

What Went Well:

- Separated logic and UI cleanly (MVC style)
- Implemented core DSA in real-world UI applications
- Worked smoothly in teams using Git and Eclipse

Challenges:

- JavaFX threading for animations and timed UI prompts
- Fitting academic concepts (like BST, PriorityQueue) into real user interaction models

Learning Outcomes:

- Reinforced algorithm application through gamification
- Learned event-driven programming, FXML
- Debugging and testing JavaFX components independently

Conclusion & Future Work

Conclusion:

- GameBox successfully combined theory and application
- We demonstrated 6 core DSA concepts through playable, user-friendly games

Future Enhancements:

- Add scoring leaderboard
- Store progress per user (login system)
- Convert to web version (JavaScript or JavaFX WebView)
- Introduce adaptive difficulty and sound

Thank You