Matching Game

Written and Implemented By

Kohiro Sannomiya (CMDC1H)

3rd Semester | 4th December 2023

Professor Bouafia

Faculty of Informatics

Programming Technology | IP-18fPROGTEG

Eötvös Loránd University

Contents

Task

You can develop your own game idea. The minimum criterie is that your game should satisfy the "Common requirement" and it has to have at least one database table, where the highscore of the players are saved. Moreover, it has to be able to read back the saved results from the database, and show them on the screen (rows can be filtered like e.g. top 10 scores).

Analysis of the Task

The application's architecture is structured around the Model-View-Controller (MVC) pattern to ensure a clean separation of concerns. The Model layer includes classes like Level, Database, and HighScore, which are responsible for managing the game's logical structure, data interactions, and maintaining high score records. The View layer, composed of NameWindow, HighScoreWindow, GameBoard, and Tile, presents the game's user interface, handling the visual representation of game levels, player interactions, and score displays. MemoryGame serves as the Controller, orchestrating game initialization, managing user inputs, and facilitating communication between the Model and View. An auxiliary component, HighScoreTableModel, assists in presenting the high score data within the user interface. This organized division allows for efficient management, scalability, and modification of the game application.

Implementation of the Task

1. **Game Loop (Controller Implementation)**: In **MemoryGame**, a central game loop is crucial, likely handled by a Swing Timer or similar mechanism. This loop is responsible for the continuous update of game states, including flipping tiles, checking for matches, and progressing through levels. It ensures that the game remains interactive and responsive to user actions.

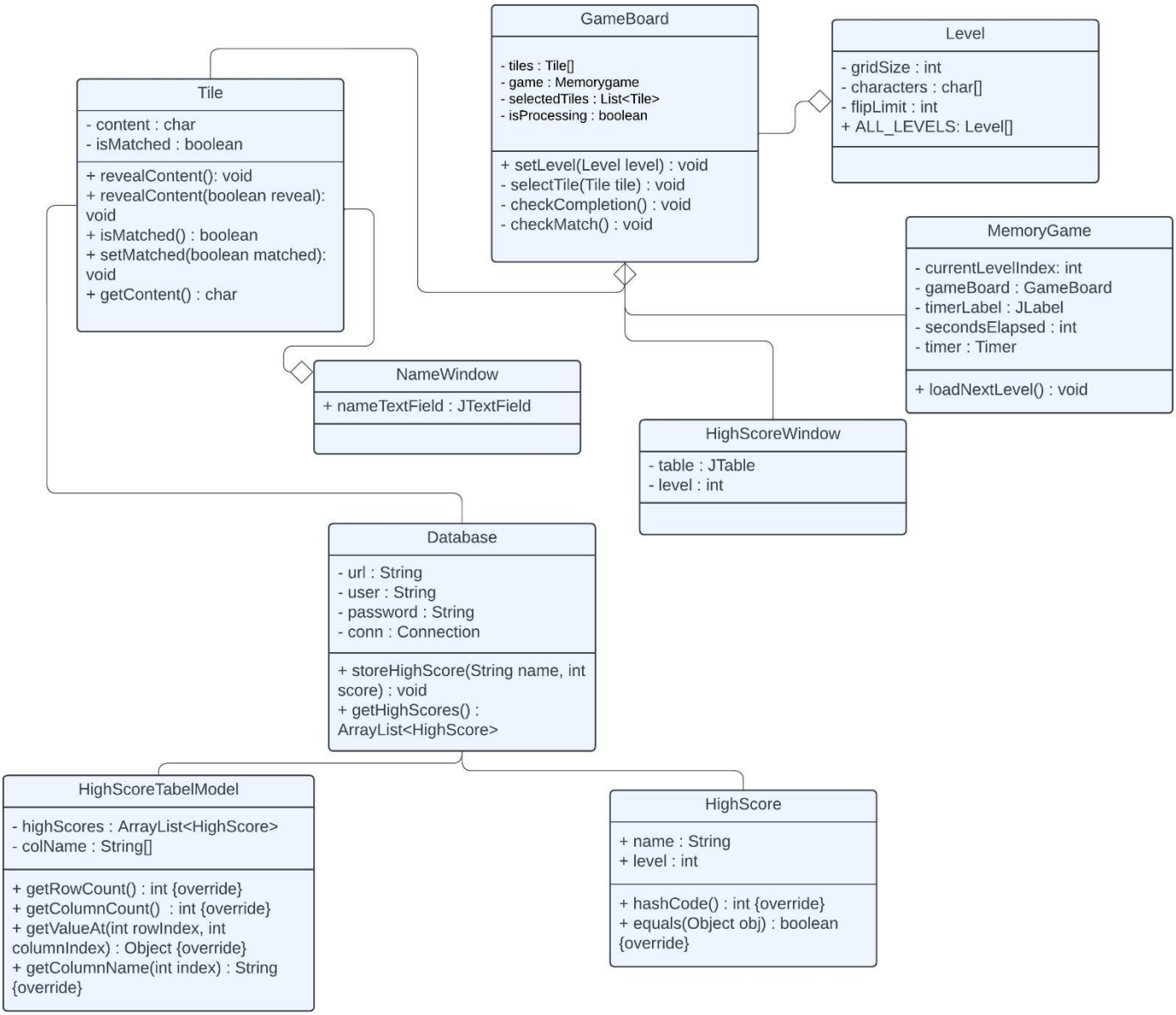
2. **Tile and Match Management (Model Implementation):** The Tile class represents each element in the game board, holding the content that players aim to match. A significant portion of the game logic, including checking for matches, revealing tiles, and managing matched pairs, is likely handled within the **GameBoard** class. This includes methods to flip tiles back if they don't match and lock them if they do.

3. **Level Progression (Model Implementation):** The **Level** class defines the difficulty and characteristics of each level in the game, such as grid size and the set of characters or images. The **MemoryGame** class likely includes methods for transitioning between levels, loading new levels as players succeed, and resetting the game state when all levels are completed.

4. **High Score Management (Model Implementation):** The **Database** class interacts with a database to store and retrieve high scores. It includes functionality to insert new high scores upon game completion and fetch the top scores for display. The **HighScoreWindow** class is used to display these scores to the player, creating a competitive element by encouraging players to beat high scores.

5. **User Interface (View Implementation): NameWindow** and **HighScoreWindow** are part of the view layer, handling user input for player names and displaying high scores, respectively. **GameBoard** also contributes to the view by visualizing the game grid and tile states. These components ensure that the game is not only functional but also engaging and easy to navigate for the user.

UML Diagram


GameBoard
- tiles : Tile[]
- game : Memorygame
- selectedTiles : List<Tile>
- isProcessing : boolean
+ setLevel(Level level) : void
- selectTile(Tile tile) : void
- checkCompletion() : void
- checkMatch() : void
Level
- gridSize : int
- characters : char[]
- flipLimit : int
+ ALL_LEVELS: Level[]
Tile
- content : char
- isMatched : boolean
+ revealContent(): void
+ revealContent(boolean reveal): void
+ isMatched() : boolean
+ setMatched(boolean matched): void
+ getContent() : char
MemoryGame
- currentLevelIndex: int
- gameBoard : GameBoard
- timerLabel : JLabel
- secondsElapsed : int
- timer : Timer
+ loadNextLevel() : void
NameWindow
+ nameTextField : JTextField
HighScoreWindow
- table : JTable
- level : int
Database
- url : String
- user : String
- password : String
- conn : Connection
+ storeHighScore(String name, int score) : void
+ getHighScores() : ArrayList<HighScore>
HighScoreTabelModel
- highScores : ArrayList<HighScore>
- colName : String[]
+ getRowCount() : int {override}
+ getColumnCount() : int {override}
+ getValueAt(int rowIndex, int columnIndex) : Object {override}
+ getColumnName(int index) : String {override}
HighScore
+ name : String
+ level : int
+ hashCode() : int {override}
+ equals(Object obj) : boolean {override}

Description of the methods

MemoryGame (Controller-Like Responsibilities)

•**MemoryGame():** Initializes the main game window, sets up the game board, and establishes a connection to the database.

•**loadNextLevel():** Advances the game to the next level once the current level is completed or restarts from the first level if all levels are completed.

GameBoard (View and Controller Aspects)

•**GameBoard(Level, MemoryGame):** Sets up the game board according to the specified level and game context.

•**setLevel(Level):** Updates the game board to reflect a new level's layout and characteristics.

•**selectTile(Tile):** Manages player interaction with the game board, specifically handling what happens when a tile is selected.

•**checkCompletion():** Checks if all pairs have been found and whether the game should advance to the next level.

•**checkMatch():** Checks if a pair of selected tiles is a match and updates the game state accordingly.

Level (Model)

•**Level(int, char[], int):** Represents the structure of a level including grid size, characters involved, and other level-specific rules or configurations.

Tile (View and Model)

•**Tile(char):** Represents a single tile on the game board, including its content and matched state.

•**revealContent():** Controls the visibility of the tile's content.

•**setMatched(boolean):** Sets the tile's matched status, affecting its interactability and display.

Database (Model)

•**Database():** Establishes a connection to the database used for storing and retrieving high scores.

•**storeHighScore(String, int):** Stores a new high score entry in the database.

•**getHighScores():** Retrieves a list of high scores from the database, typically for display purposes.

HighScore (Model)

•**HighScore(String, int):** Constructs a high score entry with the player's name and score.

HighScoreWindow (View)

•**HighScoreWindow(ArrayList<HighScore>, JFrame**): Creates and displays a window showing high scores, typically called at the end of a game or from a menu option.

HighScoreTableModel (View and Controller Aspects)

•**HighScoreTableModel(ArrayList<HighScore>):** Manages how high scores are displayed in a table, interfacing between the HighScore data and the JTable view.

NameWindow (View)

•**NameWindow():** Manages the input window for player names, allowing users to start the game with their chosen identifiers.

Event-Handler Connections

MemoryGame (Controller)

•**Keyboard Event Handlers:** The MemoryGame class listens for keyboard events to handle player interactions. It might use KeyListeners to detect when a player presses a key to flip a tile or make a selection.

•**keyPressed(KeyEvent e):** Triggers actions for tile selection or navigation in the game.

•**keyReleased(KeyEvent e):** Might be used to handle continuous key presses or provide a smoother gaming experience.

GameBoard (View)

•**Tile Selection Handler**: Each Tile in the GameBoard likely has a MouseListener or ActionListener. When the player clicks a tile, the event handler calls a method to flip the tile and check for matches.

•**actionPerformed(ActionEvent e):** Handles the event when a tile is clicked, possibly calling methods to reveal the tile's content and check for a match.

HighScoreWindow (View)

•**High Score Display:** The HighScoreWindow might have buttons or menu items for the player to interact with, such as a button to refresh the high scores or close the window. These would use ActionListeners.

•**actionPerformed(ActionEvent e):** Handles actions like refreshing high scores or closing the window.

Timer in MemoryGame (Controller)

•**Game Update Timer:** The main game loop in MemoryGame is controlled by a Swing Timer that updates the game state at regular intervals. This is crucial for real-time games where the state needs to be updated continuously.

•**actionPerformed(ActionEvent e):** Regularly called by the Timer to update the game state, flip back non-matching tiles, or advance to the next level.

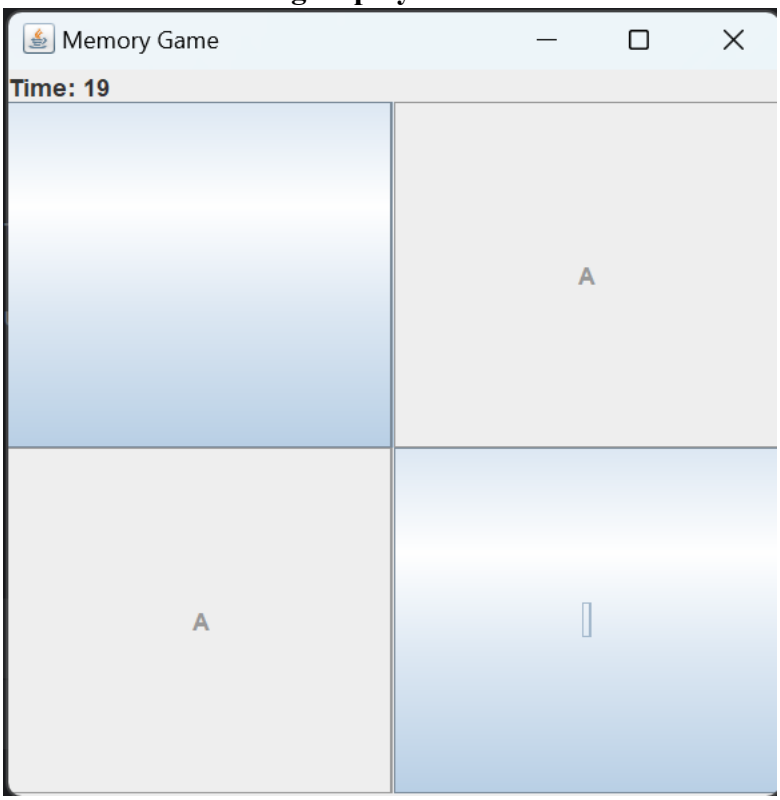
Database Interactions (Model)

•**Score Updating:** After the game ends or a level is completed, the MemoryGame or GameBoard might call methods in the Database class to update the high scores. While this isn't an event-handler in the traditional sense, it's an important interaction following significant game events (like game completion).

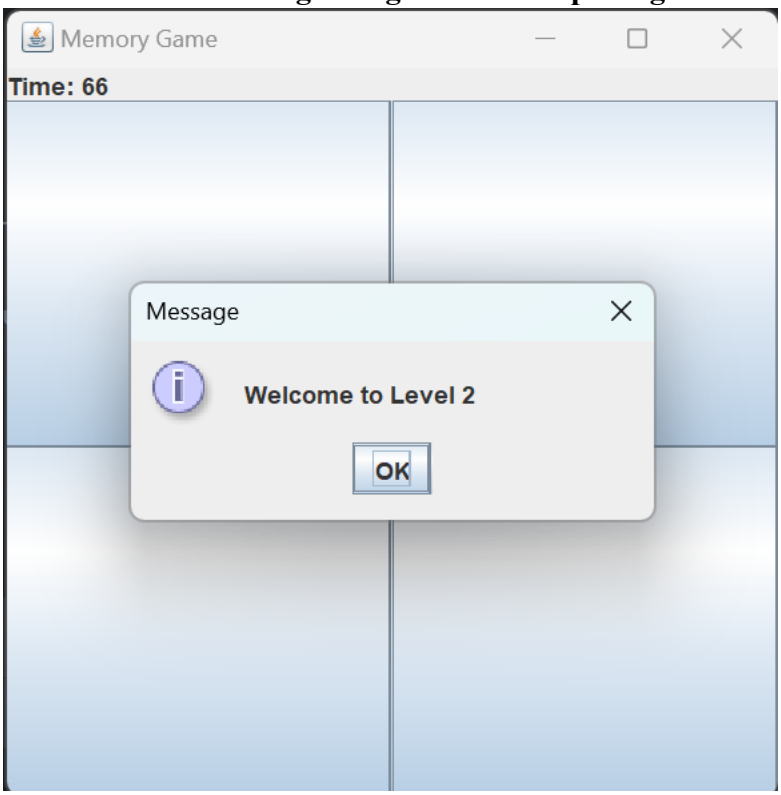
•**storeHighScore(String, int):** Updates the database with new high scores.

Tests

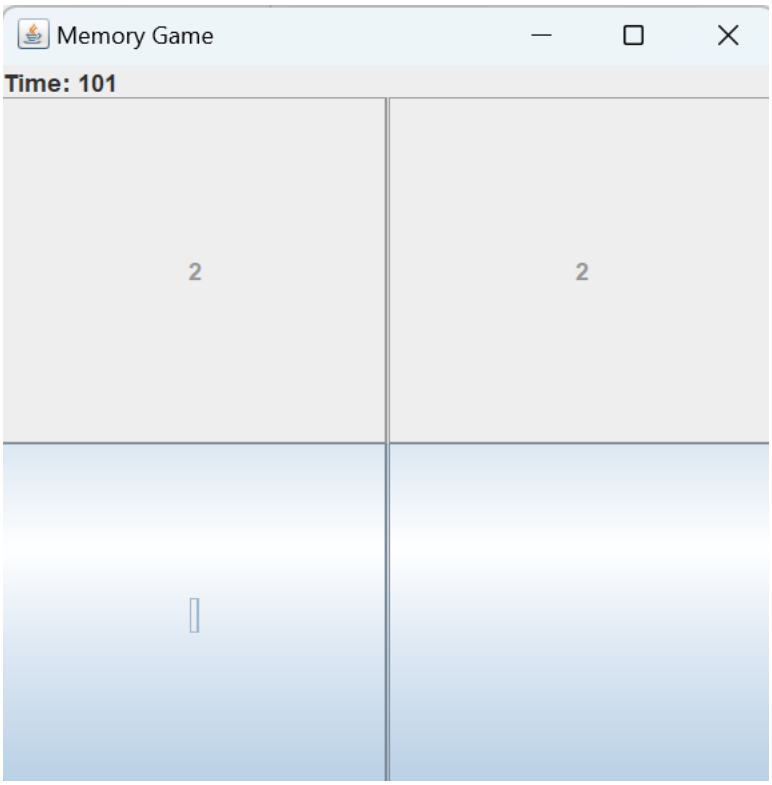
- **Case 1 – Time is being displayed**



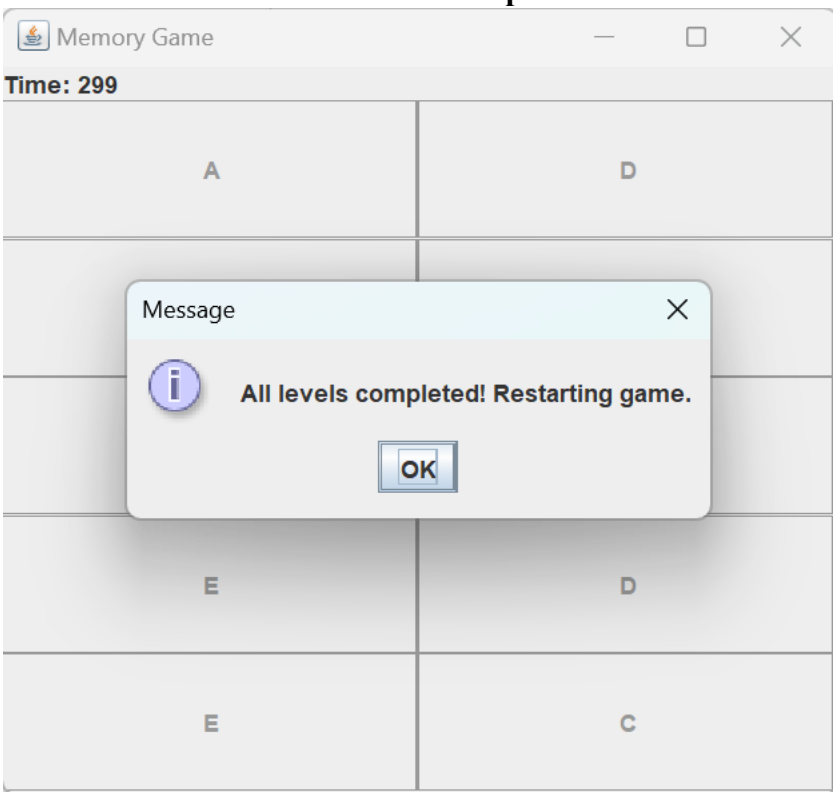
- **Case 2 – Level is being changed after completing one**



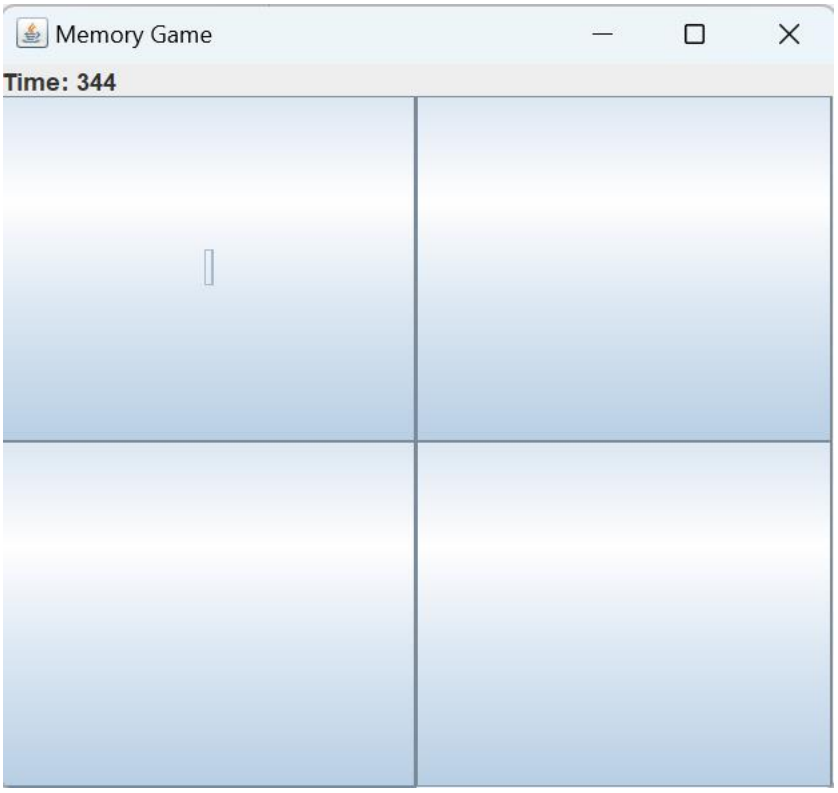
- **Case 3 – Selected ones and unselected ones**



- **Case 4 – Game is restarted after completion**



- **Case 5 – Restarting the game**


Memory Game
Time: 344