INF 122: W24

Professor Garcia

Final Project - Game Engine

- Connect Four & Memory -

Aiden On

Nathan Serrano

Do Yeon Yun

Jem Eduria

Kalman Wong

Date: 03/21/2024

Table of Contents

Instructions **3**
- Private TMGE Repo Link: 3
- How to Play: Memory (1 Player) 3
- How to Play: Connect Four (2 Players) 3

Diagrams **4**
- Main UML 4
- Memory 6
  - UML 6
  - Sequence 7
- Connect Four 8
  - UML 8
  - Sequence 9

New Game Implementation Plan **10**

Code Implementation **11**
- BasicTile.java 11
- Board.java 11
- ConnectFour.java 11
- ConnectFourBoard.java 12
- DiagonalMatch.java 12
- Disappearable.java 13
- DisappearingTile.java 13
- Endable.java 13
- Game.java 13
- GameEngine.java 13
- HorizontalMatch.java 13
- Matchable.java 14
- Memory.java 14
- MemoryBoard.java 14
- Player.java 15
- SameTileMatch.java 15
- SimpleDisappear.java 15
- Tile.java 15
- UserManager.java 15
- ValueTile.java 15
- VeritcalMatch.java 16

Reflection **17**
- High Points 17
- Low Points 17
- Major Challenges 17

Instructions

Private TMGE Repo Link:

https://github.com/jemeduria/tmge

Read the README.MD file to find instructions on running the program

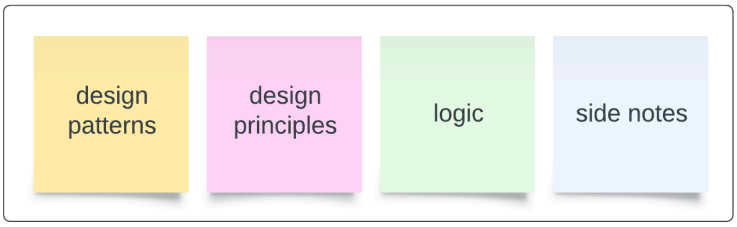
How to Play: Memory (1 Player)

1. Enter a username.
2. Choose Memory by typing “1”.
3. You have 20 seconds to memorize the numbers and its location.
4. When prompted, type the row and column for the two spots you want to match.
5. You have 3 lives. You lose a life when you fail to match.
6. GOAL: Match all the numbers.

How to Play: Connect Four (2 Players)

1. Enter a username.
2. Choose Connect Four by typing “2”.
3. Determine who will play player 1 and player 2.
4. Player 1 places “X” and Player 2 places “O”. Players alternate turns.
5. 4 (or more) Xs or Os in a row are considered a match.
6. Players gain a point for every X or O matched.
7. GOAL: Be the first player to reach 25 points.

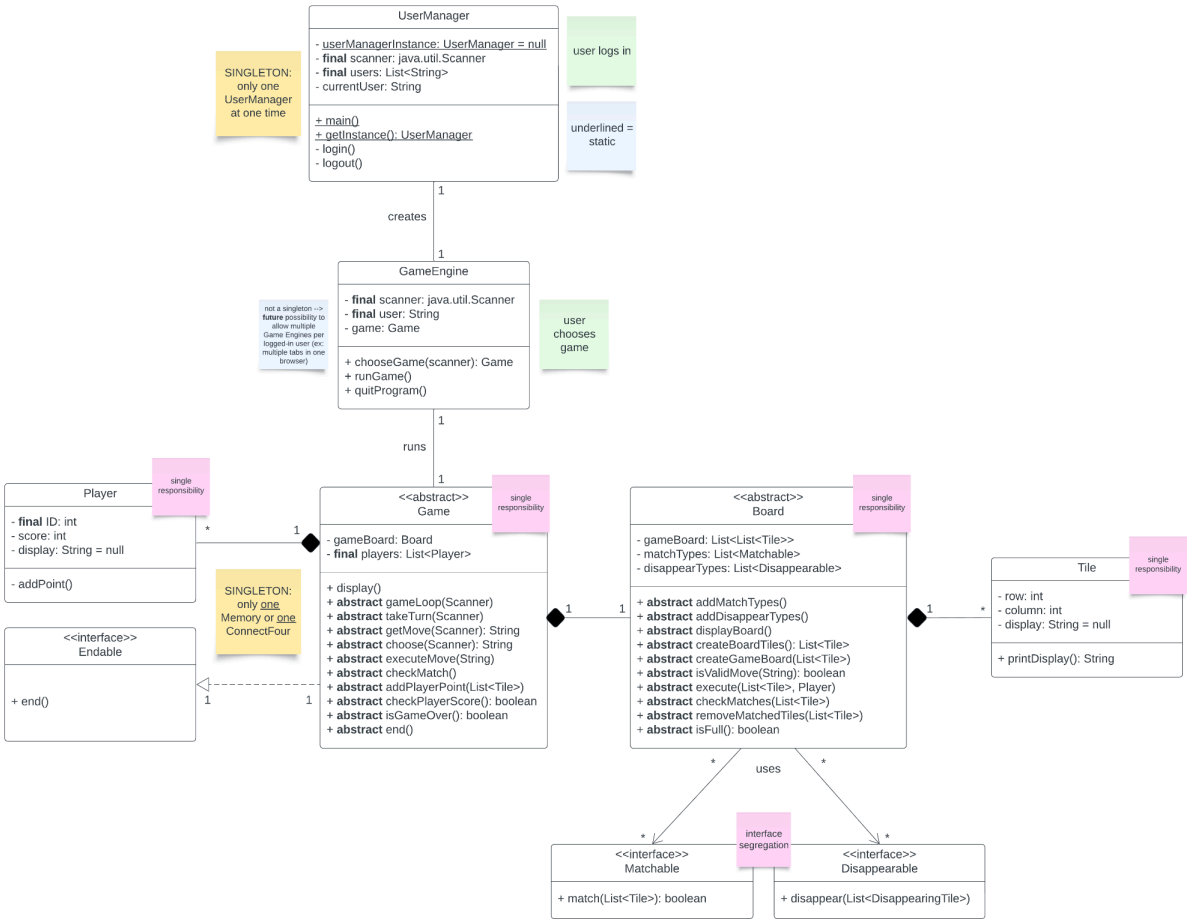
Diagrams



All UML diagrams are labeled with varying colors of post-it notes:

- **yellow** to label design patterns
- **pink** for design principles
- **green** for logic
- **blue** for other notes

Main UML



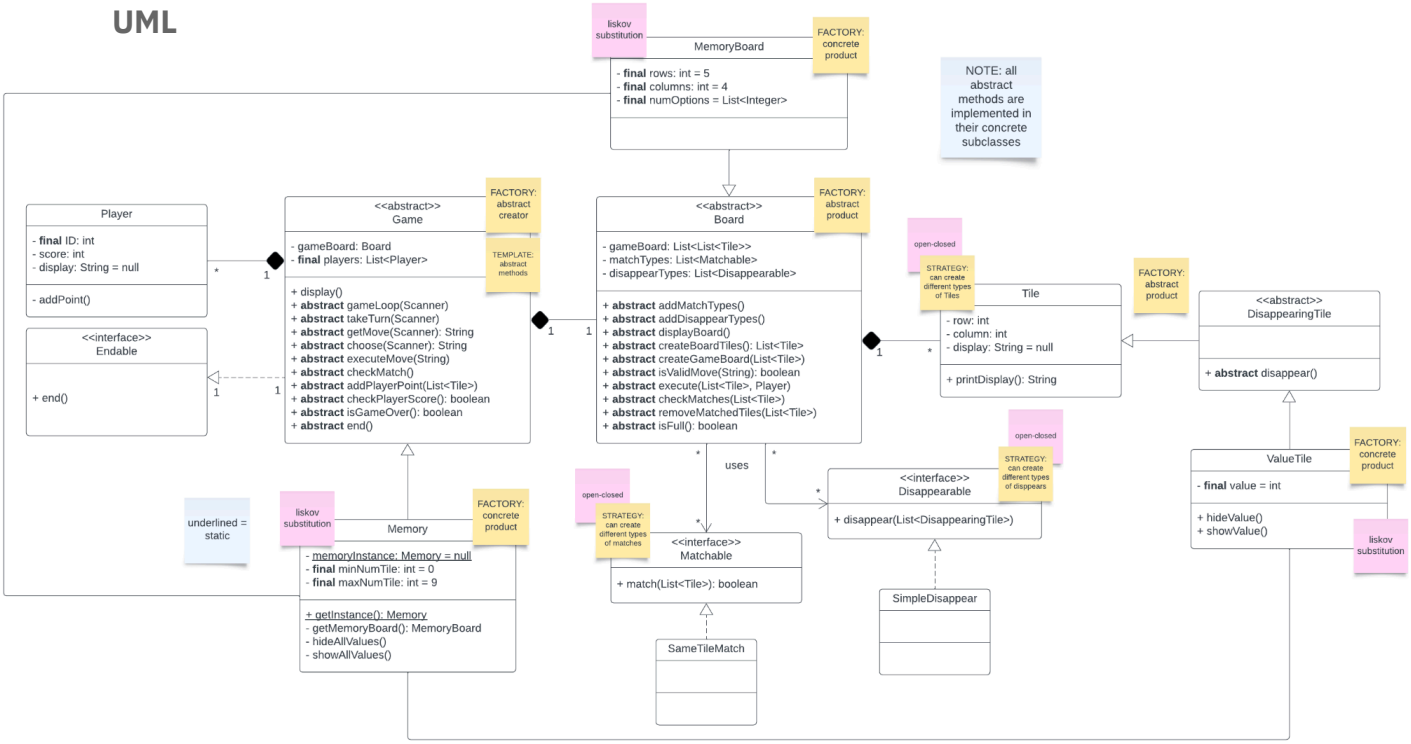
Our implementation of the Tile Matching Game Environment (TMGE) utilizes the Java programming language. The main reason the team decided on Java, as opposed to Python, is to make use of the language's flexibility in object-oriented programming. Java allows for hidden access to super- and subclasses' methods and attributes, while allowing for shared abstract override implementations.

The team decided to implement a singleton design pattern to ensure that only one UserManager class (or the main program) can be run and one game can be played at any one time. The two design principles shown in the Main UML diagram are single responsibility and interface segregation. The team separated the abstract and/or parent classes with the single responsibility design principle in mind to make sure that one class managed its attributes, and therefore split responsibilities of the full game. Interface segregation was used to ensure that operations (match and disappear) were made clear and separate actions.

Initially, our design had much fewer classes. However, after speaking to the professor and critically thinking about how the code would be implemented, we realized the importance of integrating the design patterns and ensuring that our design conforms to design principles discussed in the lectures.

Memory

UML



In comparison to the Main UML, the Memory UML includes the actual components of the Memory game: the Memory, MemoryBoard, SameTileMatch, SimpleDisappear, DisappearingTile, and ValueTile classes.

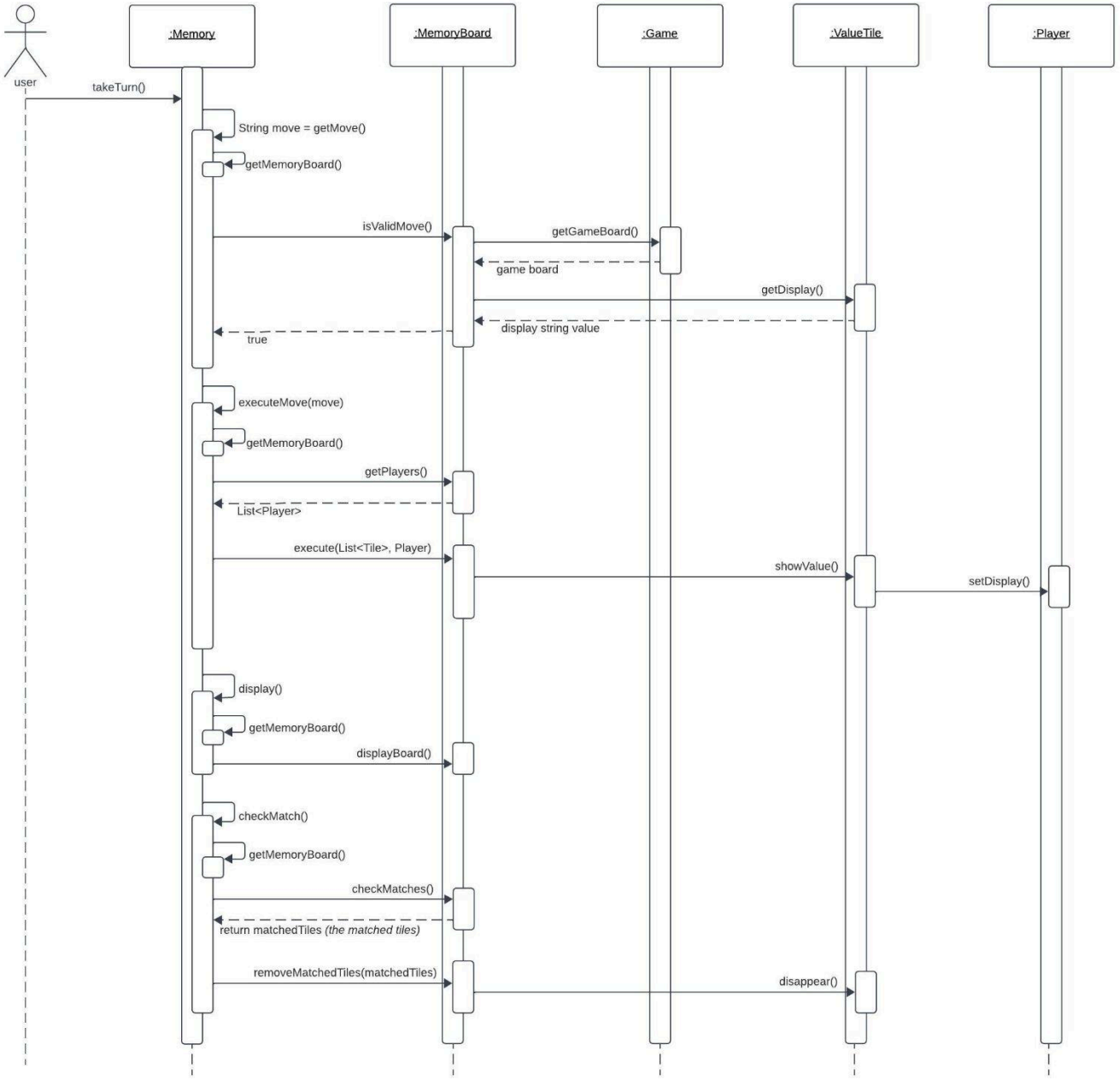
The Template design pattern is utilized in all the abstract classes, allowing for concrete implementations of the operations in the concrete classes. Similarly, the Strategy design pattern is used to give a maintainable and “upgradeable” ability for the TMGE and therefore applies to the open-close design principle. Since the TMGE currently only has two concrete game implementations, there are not many different varying subclasses of Tile and DisappearingTile. However, with more games, the Strategy pattern can be more obvious.

The team decided to design an Abstract Factory design pattern in the TMGE with the actual components of the Memory game. The Abstract Factory Creator is Game and the Concrete Factory Creator is Memory. As for the products, the Abstract Products are Board and Tile (or DisappearingTile) and the Concrete Products are MemoryBoard and ValueTile. With subclasses and superclasses, the TMGE follows the Liskov substitution design principle as well.

Sequence

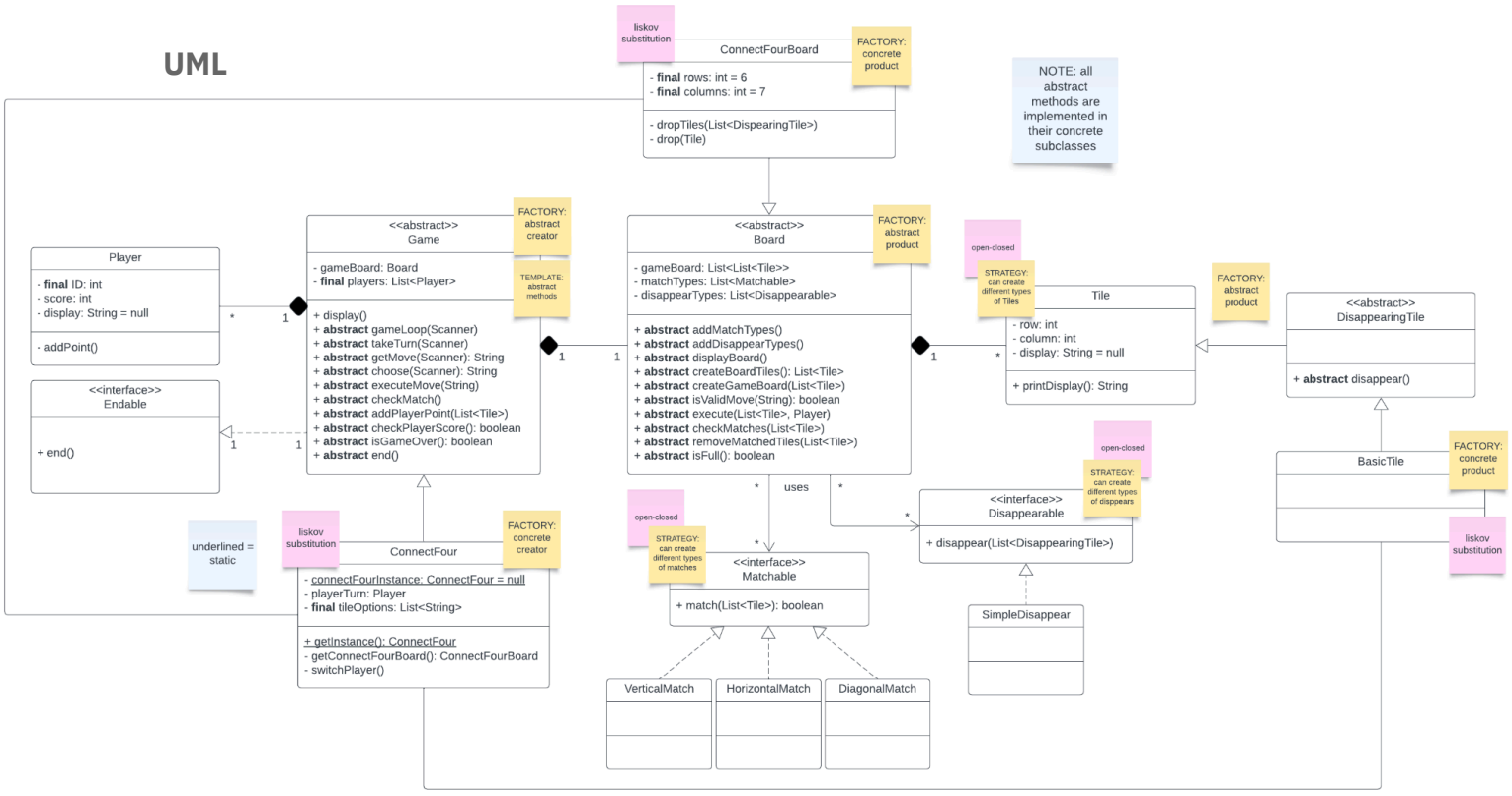
The following sequence diagram showcases a scenario in which a player takes a turn and successfully matches tiles. Thus, resulting in the disappearance of the matched tiles.

mid-gameLoop()
memory:
player taking one turn and getting a match.



Connect Four

UML

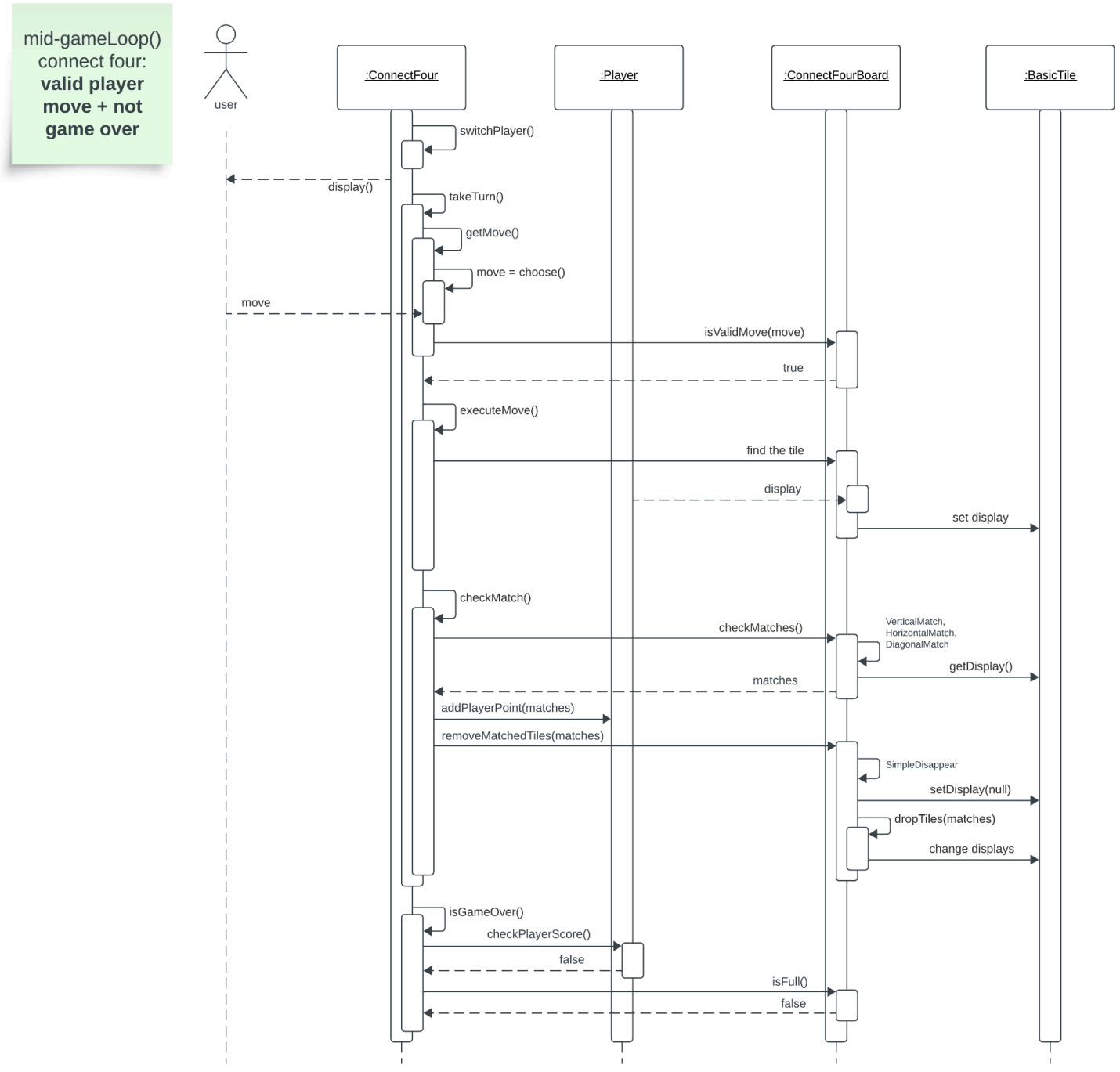


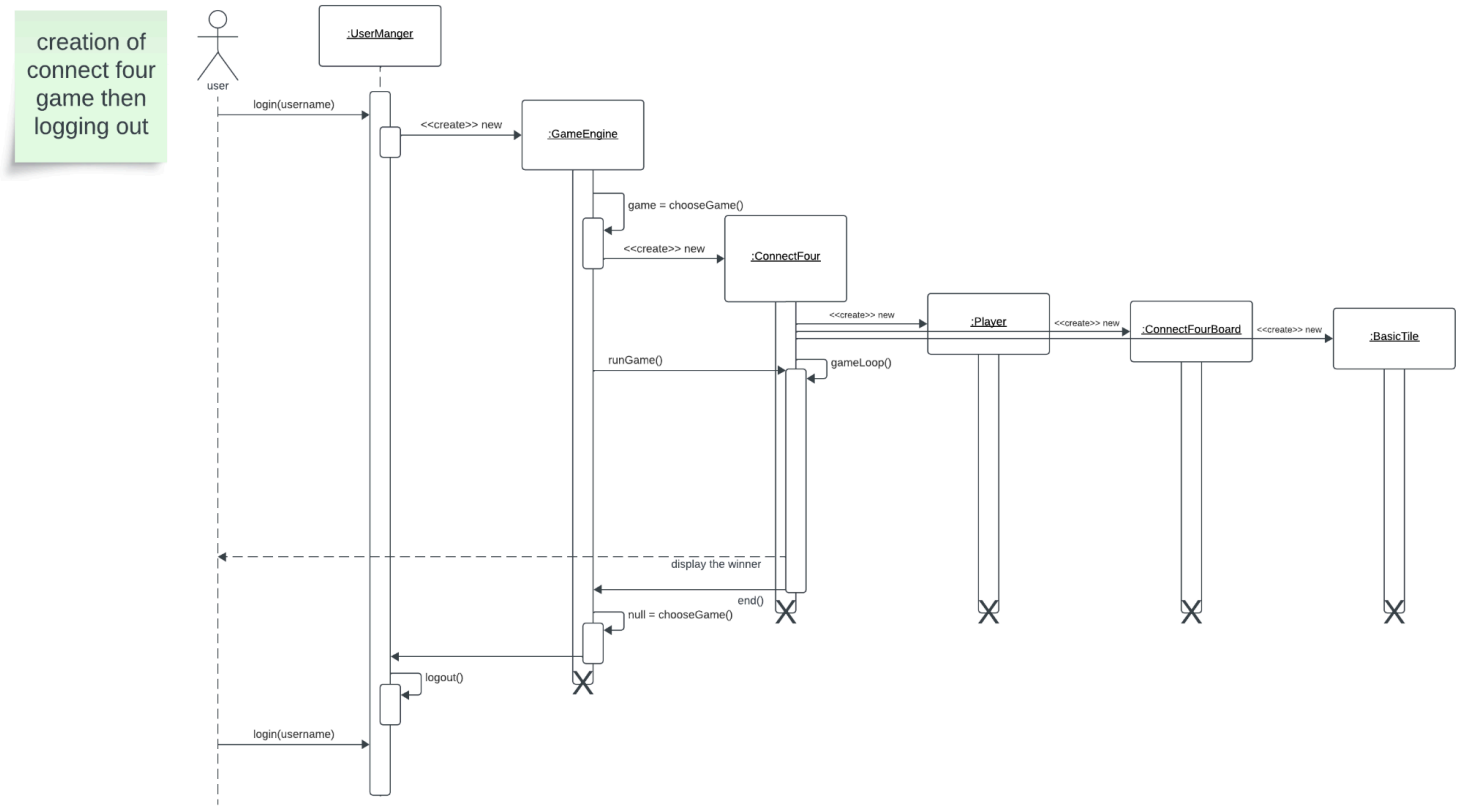
In comparison to the Main UML, the Connect Four UML includes the actual components of the Connect Four game: the ConnectFour, ConnectFourBoard, VerticalMatch, HorizontalMatch, DiagonalMatch, SimpleDisappear, DisappearingTile, and BasicTile classes.

Many of the design patterns are utilized in a similar way to the Memory UML implementation. The team decided to design an Abstract Factory design pattern in the TMGE with the actual components of the Connect Four game. The Abstract Factory Creator is Game and the Concrete Factory Creator is ConnectFour. As for the products, the Abstract Products are Board and Tile (or DisappearingTile) and the Concrete Products are ConnectFourBoard and BasicTile. With subclasses and superclasses, the TMGE follows the liskov substitution design principle as well.

Sequence

The two following sequence diagrams showcase two main situations based on the actual code: a single turn in the Connect Four game loop and the creation (and ending) of the Connect Four.





New Game Implementation Plan

The current TMGE only implements two games: a variation of Connect Four and a variation of Memory. The use of the Abstract Factory design pattern strengthens our TMGE in the possibility of implementing a new game(s). Reusable components of a new game include the varying Tile, DisappearingTiles, Matchable, and Disappearable implementations. However, if the existing components would not be helpful in the new game, the creation of new subclasses extending the superclass (or implementing an interface) can be used.

The two new main subclasses required for a new game are the classes that extend Game and Board. The concrete implementations of Game and Board are specific to the Connect Four and Memory games.

Code Implementation

BasicTile.java

BasicTile is the most basic implementation of DisappearingTile that allows the display to “disappear”.

Board.java

Board is an abstract class for the game boards of the different games. The *gameBoard* attribute is represented as a nested list of Tile objects. Other attributes include *matchTypes* (a list of Matchable types) and *disappearTypes* (a list of Disappearable types), classes that implement the **Matchable** and **Disappearable** interface, performing match checking and Tile disappearing for the games. The Matchable types applicable to the specific game will be found in the list. Board contains all the functions that specific game boards must implement.

ConnectFour.java

ConnectFour contains the functionality for the Connect Four game and inherits from **Game**. The attributes are *playerTurn* (which tracks which player is currently making a move) and *tileOptions* (which is a list of possible display options for tiles). The constructor initializes tileOptions, players, and the game board by creating a new ConnectFourBoard. This class uses basic getters and setters aside from the getConnectFourBoard functions that checks if the parent class’s getGameBoard function returns an instance of ConnectFourBoard before type casting and returning a ConnectFourBoard. The **switchPlayer()** function works by taking the remainder of dividing the current player's ID by the total number of players in the game, ensuring every player gets a turn. The game loop takes a valid user input, executes the move, and checks for matches on every iteration. Player points are added for every matched tile. The game ends when the board is full or a player has 25 points or more. This is implemented using **isGameOver()** in the game loop.

ConnectFourBoard.java

ConnectFourBoard class inherits from **Board** and is how the game board for **ConnectFour** is implemented. Static variables are kept to maintain the size of the board. In addition to the inherited constructor, ConnectFourBoard's constructor calls **addMatchTypes(), addDisappearTypes(), createBoardGame(),** and **createBoardTiles()**. The function **addMatchTypes()** populates the *matchTypes* list with the types of possible matches for this game: **VerticalMatch, HorizontalMatch,** and **DiagonalMatch**. The same type of function is also called for the disappear types. The function **isValidMove()** is used to check that the user's input is valid by checking for free space in the appropriate column. **execute()** is the function that "places" the tile but in reality only the *display* field is being changed to represent the tile being dropped as the board is fully populated with tile objects at all times. **createBoardTiles()** and **createBoardGame()** work in conjunction to create all the necessary board tiles, adding them to the board itself. **checkMatches()** iterates through the list of *matchTypes* and then calls the respective match functions on the board, returning a list of all the tiles that occurred in a match. **removeMatchedTiles()** iterates through the list of matched tiles, type casts it to a DisappearingTile, and then calls the disappear function from the tile. **dropTiles()** and **drop()** work together to drop tiles after any tiles disappear. First, the matching tiles are ordered by the lowest row number. The tiles are "dropped" by swapping the *display* value of the disappeared tiles with the *display* value of the tile in the row above it. This is done for every disappearing tile, effectively dropping every tile that has a space below it.

DiagonalMatch.java

This class uses the **Matchable** interface and checks the board for all diagonal matches, returning a list of matched tiles. The implementation of **match()** takes in a nested list of Tiles and detects diagonal matches of tiles on the game board by iterating through each diagonal line, first from the top-left to the bottom-right corner, and then from the top-right to the bottom-left corner. For each diagonal, it starts at the appropriate edge and moves diagonally, checking each tile along the way. It maintains a list of matched tiles and clears it at the start of each diagonal. As it iterates through the tiles, it checks if they are non-empty and if they match the previous tiles in the sequence or if they represent the beginning of a new sequence.

If a match of four or more tiles is found, it returns the list of matched tiles; otherwise, it continues scanning.

Disappearable.java

Interface for disappearing. Contains the **disappear()** function.

DisappearingTile.java

An abstract class that inherits from Tile and outlines a function for Tiles that can disappear.

Endable.java

Interface for ending a game. Contains the **end()** function.

Game.java

Game is an abstract class that implements the Endable interface. It serves as an interface for the games within the game engine with functions that these game classes must implement. It defines the **getUserInput()** function and the **parseNumber()** functions to take correct inputs from the user. It has a Board attribute called *Board* and another attribute called *players*, which is a list of Players.

GameEngine.java

GameEngine contains functionality to take user input for choosing a game, logging out, or quitting the program. Based on the input, instances of the selected game are created. It contains a function for starting the game's running loop and a function to end the program as well as getters and setters.

HorizontalMatch.java

This function implements the Matchable interface. It traverses each row, analyzing consecutive tiles to identify matches. At the start of each row, it resets the matches list. While traversing each row, it checks for consecutive matching tiles. If a sequence of at least three matches is found, it includes the current tile and returns

the matches list once it finds a sequence of four or more tiles. If no match is found after iterating through all rows, it returns null.

Matchable.java

Interface for the types of matches. Contains the **match()** function.

Memory.java

This class contains the logic for running Memory and it inherits from Game. The constructor initializes the game board, players, and a list of possible tile options. **gameLoop()** runs the main game loop until the game is over. It displays the game board, takes turns from players, and checks if the game is over. **takeTurn()** prompts the current player to choose two tiles to check for a match. It then executes the move, updates the game board, checks for matches, and hides matched tiles for the next turn. **getMove()** and **choose()** ensures the user inputs information for 2 tiles and that the chosen tiles are valid. **executeMove()** parses the chosen tiles and calls board.exceute() to update the game board. **checkMatch()** calls Board's checkMatches() and removeMatchedTiles() to remove matched tiles. **isGameOver()** checks for an empty board or a full board which both end the game.

MemoryBoard.java

Inherits from Board. The constructor initializes the game board by calling methods to add matches, disappear types, and create the board game with tiles. **addMatchTypes()** and **addDisappearTypes()** methods add match and disappear types specific to the game. **isValidMove()** works by parsing the user input and checking if the chosen tile has been matched or disappeared. **execute()** checks if the tile is an instance of ValueTile and if so, it casts it to a ValueTile and displays the value. **createBoardTiles()** creates all the tiles for the game. It puts these tiles into a list and calls Collections.shuffle to get a random order for the game and then returns this list. **createBoardGame()** adds the tiles from the previous function into the actual game board. **checkMatching()** and **removeMatchedTiles()** work the same way as they did in ConnectFour, except MemoryBoard's **removeMatchedTiles()** does not drop tiles. **isFull()** checks that all the tiles' displays have been set to "X", denoting a disappeared tile.

Player.java

Player is a class representing individual players for two player games. It contains attributes *ID*, *score*, and *display* for the value that is shown on screen.

SameTileMatch.java

Implements the matchable interface. **match()** iterates over each tile, checking if its display value matches a designated value stored in the variable *toBeMatched*. If a match is found, the tile is added to the matched list. If exactly two matching tiles are identified, the method returns a list containing these tiles; otherwise, it returns null.

SimpleDisappear.java

Implements Disappearable. It calls the tile.**disappear()** function for desired tiles.

Tile.java

This is an abstract class for game tiles. It holds values for the position of a tile with *row* and *column* fields, as well as a value for the tile's *display* value.

UserManager.java

User manager is the main entry point to the program. The class takes user inputs from the console and allows users to log in by comparing the inputted username to an arraylist of valid usernames. For every login, a new GameEngine class instance is created

ValueTile.java

Extends DisappearingTile. On top of its parent class's features, it also has a *value* attribute for the "card value" in Memory. **disappear()** works by changing the display field to "X" and hiding the tile works by setting *display* to null.

VeritcalMatch.java

Implements Matchable. It iterates over each column and row of the game board, checking if there are four consecutive tiles with the same display value vertically. It does this by keeping an arraylist of matched tiles for each column and comparing a current tile to a value in that arraylist. If such a match is found, it returns a list containing the matching tiles.

Reflection

High Points

Taking into account the whole course of the project, the high points are mainly during the design phase. With lots of help from the professor, the team was able to easily understand the importance of the design patterns and quickly implement them in the TMGE UML.

Low Points

The low points of the project are mainly concentrated in the developmental phase of the project, during the coding phase. Our team faced difficulty in coding session schedules and deciding on algorithms. Ultimately, we persevered through the challenges and finished the project.

Major Challenges

One major challenge our team faced was how to deal with disagreements about algorithms. Our current TMGE only utilizes a few algorithms, mainly in the Connect Four implementation. Our team was split when it came to deciding decisions on the logic and which algorithm to code. By emphasizing clear communication among the team members about the algorithms along with the professor's discretion, we were able to overcome this challenge.

Another major challenge we eventually overcame was coding with a hybrid schedule. The team agreed to have two coding sessions: one on Tuesday and another on Thursday. Tuesdays were remote coding sessions where we peer-programmed. Thursdays were in-person coding sessions and allowed the members to easily discuss tasks and seek help from the professor in person when needed. Although this schedule helped the members with varying transportation issues and outside commitments, this inconsistent schedule made it difficult to code when members needed clarity of the algorithms and overall how the code would work. However, by focusing on our goals and conforming to task deadlines, our team was able to split up the work among the team members, finishing the code through peer programming.