SWA1 A23 Course assignment 2

Match3

Match 3 games are a popular pastime on mobile devices. They are simple, colourful, easy to make, and easy to monetize. They come in several variations with each their rule set.

Here we are going to consider a simple version of a match 3 game:

- The board is a rectangle consisting of individual tiles. On each tile is a piece.
- The player chooses two tiles to swap. If the move is legal (see below), the game swaps the two tiles, finds all horizontal and vertical matches of at least 3 identical tiles and removes them, causing the tiles above to drop down in their place and new random tiles to drop down from above.
- If dropping tiles form new matches, these, too, are removed, and the process continues until there are no more matches and the player is ready for the next move.
- A move is legal if the two tiles are in the same row or the same column and the swap result in a match.
- We are not implementing any concept of scoring or the game ending at this point.
- The tiles can be anything: Candy, jewels, cookies, cute animal faces. Here we are just going to use letters for illustration.

The required code

The task is to implement a board with rules like above using object-oriented and functional techniques.

In the object-oriented version, implement a Board class with methods to read the status of the board, allow for moving, and listen to events from the board (matches and the board refilling). Also, define some parameter and return types.

In the functional version, implement a Board type, and several functions to create the board, read the status of the board, and allow for moving. The events will be returned from the function to make a move, as functional programming doesn't allow the concept of listening for changes.

Details (Object-oriented)

Generics

Since tiles can be anything, most of the types are parameterized with a type parameter, <T>. This is mandatory.

Generator

New tiles are generated randomly, but they could come for the server. Also, for testing, randomness is impossible to work with. Hence, a generic Generator type is given in the uploaded template.

Position

Position is simply the position of the tile on the form {row: 2, col: 4}.

Match

A match represents a match found on the board. It has two properties: *matched* is the tile found to match, *positions* is an array of the positions found in this match (see below for examples).

BoardEvent

BoardEvent is not given. It represents two kinds of events

- ```
  { kind: 'Match',
    match:
      { matched: 'A',
        positions:
         [{row: 0, col: 0}, {row: 0, col: 1},{row: 0, col: 2}]}}
  ```
  This represents the event that a series of three As have been found on the top row starting in the top left corner.
- ```
  { kind: 'Refill' }
  ```
  This represents that tiles have dropped down in the empty spaces and new tiles have emerged on top. It is fired after each set of matches, but several can happen in a single move (more below)

BoardListener

Not given. A board listener is a function that is called when an event fires.

Board

Not given. Represents the game board. Aside from the constructor, it requires 4 methods:

- *addListener* – adds a listener to react to events
- *piece* – returns the piece on the tile at the given position, or undefined if it is not a position on the board.
- *canMove* – returns true if swapping the tiles at the positions is legal according to the game rules. False, otherwise.

- *move* – perform the move by swapping the tiles.

Examples

Example 1

Imagine a move has resulted in the board below, with the highlighted being the match. The match is removed, and new tiles B, A, and C appear.

A	A	A
B	A	D
A	C	D

This will result in the board below

B	A	C
B	A	D
A	C	D

And the events

- `{ kind: 'Match', match: { matched: 'A', positions: [{row: 0, col: 0}, {row: 0, col: 1},{row: 0, col: 2}]}}`
- `{ kind: 'Refill' }`

Example 2

Imagine a move has resulted in the board below, with the highlighted being the match. The match is removed, and new tiles B, A, and D appear.

A	A	A
B	A	D
A	C	D

This will result in the board below. The right column is a new match

B	A	D
B	A	D
A	C	D

This will result in the events

- `{ kind: 'Match', match: { matched: 'A', positions: [{row: 0, col: 0}, {row: 0, col: 1},{row: 0, col: 2}]}}`
- `{ kind: 'Refill' }`
- `{ kind: 'Match', match: { matched: 'D', positions: [{row: 0, col: 2}, {row: 1, col: 2},{row: 2, col: 2}]}}`
- `{ kind: 'Refill' }`

Example 3

Imagine a move has resulted in the board below. This board has two matches, resulting in two match events.

A	A	A
B	A	D
A	A	D

This will result in the events

- `{ kind: 'Match', match: { matched: 'A', positions: [{row: 0, col: 0}, {row: 0, col: 1},{row: 0, col: 2}]}}`
- `{ kind: 'Match', match: { matched: 'A', positions: [{row: 0, col: 1}, {row: 1, col: 1},{row: 2, col: 1}]}}`
- `{ kind: 'Refill' }`

Differences in the Functional Style

The major difference is the use of functions instead of a class. The other big difference is handling events.

Effects

Effects are the functional style's replacement for events. As events, it comes in two versions. The Match effect is exactly like the Match event, but the Refill effect is different:

- `{ kind: 'Refill', board: … }`
  The board property holds the state of the board after the refill as there is no other place to access this information.

MoveResult

MoveResult is the outcome of performing the move. It holds the value of the final, updated board and an array of effects corresponding to the events from the OO solution.

Board

Board is now just a type (not given), which is supplemented by five functions: A factory function and one for each method of the class. The methods now take board as a parameter. In addition, the move() function takes a generator and returns a MoveResult.

The template

The file template.zip contains two projects: oo and functional. Both projects have a main code template in src/board.ts and a test suite (using Jest) in __test__/board.test.ts. The tests can be run using npm test (remember to npm install).

Write the remaining code in src/board.ts while running the tests to see how you are progressing.

The hand-in

- Groups: 2-4 people. Groups can be a mix of members from X and Y class, but remember to write it in the hand-in.
- Suggested deadline: 23 October. (Real deadline is exam as previously indicated.)
- Hand-in a zip file with the two projects.
- Hints:
    - It's easy to get lost in this one, so use small steps and run the tests all the time.
    - There is not a lot of code, but there are some tricky algorithms. Work together. It is especially ill-advised to write the OO and functional versions in isolation since they have significant overlap.
    - Write as many helper functions and private methods as you need. It's easier to have smaller functions.
    - Recursion is your friend.