«class»
Element

// The type of the Element as integer ID [0..n].
+ elementType : int
// The Unity visualization of the Element in the ElementGrid
// (the associated GameObject wich represents the Element in the scene).
+ visuals : GameObject

// Constructor
+ Element(visuals : GameObject, elementType : int)
// Updates the position of the elements visualization.
+ UpdateElement(worldSpacePosition : Vector3)
// Destroies the element. Removes the Elements represantation from the Unity scene.
+ DestroyElement()

Achtung:

Dieses Diagramm stellt nur einen groben Überblick dar. Benutze die gegebene HTML-Dokumentation um ausführlichere Hinweise zu den gegebenen Klassen zu bekommen.

«class»
Level

// Possible States of the currentLevel.
// NoElementsLeft: The Level is "empty".
// NoMoreMovesPossible: There are still elements in the level but the player can't perform further moves.
// FurtherMovesPossible: This is the DEFAULT state. This State remains as long as the player
// has the possibility to remove further elements from the level.
+ enum LevelState { NoElementsLeft, NoMoreMovesPossible, FurtherMovesPossible};

// Playerscore (points the player has made).
- _points : int
// The playground.
- _grid : ElementGrid

+ «property» points : int

// Constructor
+ Level(origin : Vector3,
cellSize Vector2 ,
cellCount_X : int,
cellCount_Y : int,
seedForRandomNumberGenerator : int,
prefabs : GameObject[],
parent : Transform)

// This Function implements the funcionality for MouseHover events.
+ HoverCells(worldPosition : Vector3) : int
// This Function implements the funcionality for MouseClick events.
+ SelectCells(worldPosition : Vector3) : int
// Checks the state of the current Level (NoElementsLeft, NoMoreMovesPossible, FurtherMovesPossible).
+ CheckLevelState() : LevelState

// Returns the indices of adjacent cells with the same elementType.
+ GetAdjacentCellsOfSameType(cellIndex : int) : int[]
// Calculates the points for each move.
+ CalculatePoints(numElements int) : int

«class»
ElementGrid

// Returns the total number of cells in the grid.
+ «property» CellCount : int {get}

// constructor
+ ElementGrid(origin : Vector3, cellSize : Vector2, cellCount_X : int, cellCount_Y : int)

// Returns the center (point in worldSpace) of the cell with the corresponding cellIndex.
+ GetCellCenter(cellIndex : int) : Vector3
// Transforms a worldspace position into a cellIndex.
// Attention: If the point is outside of the grid bounds, the returned index is not valid!
+ PointToIndex(positionInWorldSpace : Vector3) : int

// Sets the Element instance for this cell and updates the position of the visuals (GameObject).
+ SetElement(cellIndex : int, element : Element)
// Returns the Element instance from the cell with the corresponding index.
+ GetElement(cellIndex : int) : Element

// Removes the elements of the cells given by the cellIndices. The Content of these cells is null afterwards.
// The ElementGrid also moves the remaining Elements down and the remaining (not empty) columns to the left
// if there are empty columns in between.
// If Elements are moved, the position of their visuals (GameObjects in the scene) are updated automatically
+ RemoveElements(cellIndices : int[])

// Returns the (not empty) neighbours ("4er Nachbarschaft") as indices of the neighbouring cells.
// The number of neighbours can vary between 0 and 4.
+ GetNeighbours(cellIndex : int) : int[]
// Determines whether the given cellIndex is a valid index for this grid.
+ IsIndexValid(cellIndex : int) : bool