Design

# Entity-Relationship Diagram

# Database Design

# Data Structures

## Coordinate Class

|  |
| --- |
| Coordinate |
| -xPos : int -yPos : int  +XPos : int  +YPos : int  -visited : bool  +Visited : bool |
| +Coordinate(int : xPos, int : yPos) +GetCartesianCoordinates(Maze : maze) : Tuple<int, int>  +GetManhattanDistance(endPoint : Coordinate) : float |

The Coordinate class will be used whenever coordinates are necessary within the program. Most algorithms will utilise a stack of coordinates, adding new coordinates to the stack and popping the top value to backtrack.

The Coordinate class is useful since it allows more flexibility than just using tuples or similar data structures to store coordinates. A tuple can only store the two integers for example, but the coordinate class allows the code to get information from the object, which may simplify the more complex solving algorithms.

The private properties xPos and yPos will store the positions of the cell in a 2D array – this means that the Y coordinate will start at the top of the 2d grid and increase as you move down. I am therefore adding the GetCartesianCoordinates method to obtain traditional coordinates to a grid cell, which some solving algorithms will require. It needs to take an argument which contains the current maze in order to compute this. The public properties XPos and YPos will act as read-only accessors to their private counterparts using the get{} keyword in C#. This will allow coordinates to be initialised at creation, but not be accidentally changed afterward. The visited property is useful for some generation algorithms which need to remember if a certain cell has been visited. The method GetManhattanDistance will return a floating-point value equal to the Manhattan Distance between the coordinates of the Coordinate object and Coordinate argument, which will be useful in more complicated algorithms such as Maze-Routing, A\*, or First-Breadth Search.

## Maze Class

|  |
| --- |
| *Maze* |
| -mazeActualWidth : int  +MazeActualWidth : int -mazeActualHeight : int +MazeActualHeight : int  -mazeCellWidth : int  +MazeCellWidth : int  -mazeCellHeight : int  +MazeCellHeight : int #mazeWalls : bool[,]  #mazeCoordinates : Coordinate[,] |
| {abstract} +initMaze() : void {abstract} +buildMaze() : void  {virtual} #cellVisited(cellPos : Coordinate) : bool |

The abstract class Maze will be the parent class of all maze types added and should streamline adding new maze types to the program. It provides essential properties for mazes and forces its subclasses to implement buildMaze.

The first 8 properties will hold the integers representing width and height, measured in array dimensions and number of cells, privately with read-only public access. The 2D array mazeWalls will hold the structure of the maze with booleans representing whether a cell contains a wall. I am using booleans since using char or string would make the maze files bigger and may affect network speed. Also, the maze will be displayed in a Windows Forms GUI so using letters would not add any extra utility to the program. I may also implement the 2D array mazeCoordinates as an array of Coordinate objects, as some algorithms may need the ability to always track the position of cells. The abstract method initMaze will be used to initialise the mazeWalls array with the pre-built maze structure, whilst buildMaze will implement the generation algorithm. The method cellVisited will query if a cell has been visited - with most algorithms this means if it still has 4 walls, but I am making it virtual in case any algorithms need their own definition.

## Hierarchy Chart

//need to decide what algorithms to include before making this

# Key Algorithms

## Initializing the maze

Whilst some algorithms may start with a grid full of walls, most maze generation algorithms will start with a waffle grid. This is the pseudocode for generating this waffle.

|  |
| --- |
|  |

First, the actual dimensions of the maze are calculated from the user input of cell dimensions. The mazeStructure array is then initialized with these sizes. Then, each cell is iterated through and is defined as either a wall or passage, with False representing passages and vice versa.

## Depth-First Generation

### Recursive Implementation

This algorithm takes the waffle and carves a perfect maze into the array. It uses recursion to backtrack through the passages it carved if a dead end is reached via the system call stack.

|  |
| --- |
|  |

A random cell in the waffle is passed to the subroutine as an argument. This cell is marked as visited. Then a list of all unvisited cells neighbouring the current cell is obtained. The pseudocode checks if the cell north, east, south, and west is visited. If it is not visited, it adds it to the list. I am adding 2 instead of 1 to the coordinates as I need to check the next cell, not the wall separating them. Next, a while loop is entered if there are unvisited neighbour cells. The algorithm picks a random cell from the list and destroys the wall between the current cell and the randomly selected one. The subroutine is then called recursively on the randomly selected cell. When a dead end is reached, the size of the list will equal 0, so the while loop will exit and control will return to the previous iteration of buildMaze, effectively using the call stack to backtrack until an unvisited cell is found.

### Stack implementation

The recursive algorithm is short and powerful but may exceed the maximum limit on recursion depth if the maze is too large. In the worst-case scenario, the algorithm will be called on every cell in the maze. If I run into issues with this, I may use an alternate implementation with an explicit stack of coordinates and a constructor which roams the maze, which has a less strict size limitation.

|  |
| --- |
|  |

A random cell is passed to the subroutine. The constructor is set to that cell and the constructor’s position is added to the stack. The number of unvisited cells in the maze is then calculated. Then, while there are still unvisited cells in the maze, the constructor checks if there are unvisited cells around it. If there are, then it picks a random one and travels to it, destroying the separating wall, adding its new position to the stack, and reducing the number of unvisited cells in the process. If, however, the constructor has reached a dead end, it backtracks down the stack until it finds an unvisited cell to enter. Eventually the constructor will have visited every cell and carved out a perfect maze in the process.

## Wilson’s Generation Algorithm

## Depth First Search Solving Algorithm

This is an easy to implement solving algorithm which find a solution, but it may not be the shortest one in a labyrinth-style maze. It is almost identical to the random mouse algorithm but uses the visited feature in the Coordinate class to make sure the solver never goes back down an already explored passage.

|  |
| --- |
|  |

## Tremaux’s Solving Algorithm

//implementing this might be hard without string arrays for structure storage

## Maze Routing Solving Algorithm