Software Design Document

For

Labyrinth of Algorithms

Version 1.3

Prepared by Will Franzen and Lukas Livengood

Case Western Reserve University,
CSDS 393 - Software Engineering

December 9, 2022

Table of Contents

1. Introduction

1.1. Purpose

This Software Design Document serves as a foundation for the initial demo to ensure that the implementation of the features described in *Software Requirement Specifications for Labyrinth of Algorithms (Version 1.2)* meet the goals set for the application's design and functionality. The intended users of this document are the project members.

In this document, the following aspects of the Labyrinth of Algorithms software are described as follows:

- Principal Classes
- Inter-Agent Protocols
- Class Interfaces

1.2. Design Architecture

To allow for the continued facilitation, maintenance, and reusability of the Labyrinth of Algorithm's software, we are utilizing an Object-Oriented Design (OOD) architecture. This design decision will be applied to the backend given our employment of unique data structures holding many individual elements with states able to correspond to a variety of different attributes. However, our frontend work shall conform to an Event-Driven Design architecture given our software's reliance on user interaction with the program's GUI. The event handlers used in through this architecture are necessary for key processes to take place on the frontend, but once they occur, the data transmitted shall then be handled in our OOD backend.

2. Principal Classes

2.1. Tile

The Tile class is responsible for the following:

R-1: Modify the Tile state as defined by the user's selection
R-2: Provide data regarding Tile object attributes to querying classes

Each of the following classes will heavily rely on the Tile objects created by this class. The extent to which these classes shall interact with the Tile class is described both in Sections 2.2 to 2.5 & Section 3. The Tile class shall work in the backend to return queried information about individual Tile states (passable, impassable, initial, goal) for the search algorithms and visualization of the Labyrinth.

2.2. Labyrinth

The design of our software merits only a single Labyrinth class agent, responsible for the following:

R-1: Initialize a two-dimensional of Tile objects for use by the other classes
R-2: Provide Algorithm class a copy of its data structure for calculation usage
R-3: Update the Labyrinth's data structure to contain the Algorithm goal path
R-4: Visualize the calculation process as specified by the Algorithm class

The Labyrinth class' main purpose is that of data storage and visualization. As such, this class will connect the work done on the backend to that of the frontend. The Labyrinth class shall organize multiple Tile objects (the desired amount is to be defined by the user) into a two-dimensional data structure for use by the Algorithm class in its calculations. These calculations are to be returned to the Labyrinth class and visualized in the frontend software.

2.3. Algorithm

In the software's design, the Algorithm class is responsible for the following:

R-1: Query Tile objects to determine consideration in search algorithms

R-2: Move pointers towards the initial and goal state Tile objects
R-3: Calculate and return goal paths according to the selected algorithm
R-4: Verify solvability of the given Labyrinth agent

The Algorithm class shall rely mainly upon the Tile class as defined in Section 2.1 and produce results for the described Labyrinth class in Section 2.2. Given the provided Labyrinth agent, the Algorithm class shall be able to calculate a goal path using one of the search algorithms defined by Section 3.2.1 in *Software Requirements Specification for Labyrinth of Algorithms (Version 1.2)*. All of the work done by the class will occur in the backend; however, the results calculations produced will play a key role in the visualizations as described in Section 3.2.

2.4. Pointer

The software's design additionally calls for the Pointer class, responsible for:

R-1: Individual tile selection by the user
R-2: Individual tile modification by the user
R-3: Execution of the software's visualizations
R-4: Passing user defined movements into the Labyrinth class

The Pointer class gives the user the ability to define certain specifications as to how the software's execution will occur. For modifications made by the user defining how the program should execute, the Pointer class shall rely these characteristic changes to the relevant principle classes as described above here in Section 3. Additionally, in the User Traversal mode as described in Section 3.3.2 of *Software Requirements Specification for Labyrinth of Algorithms (Version 1.2)*, the Pointer class shall track and calculate the validity of user defined movements.

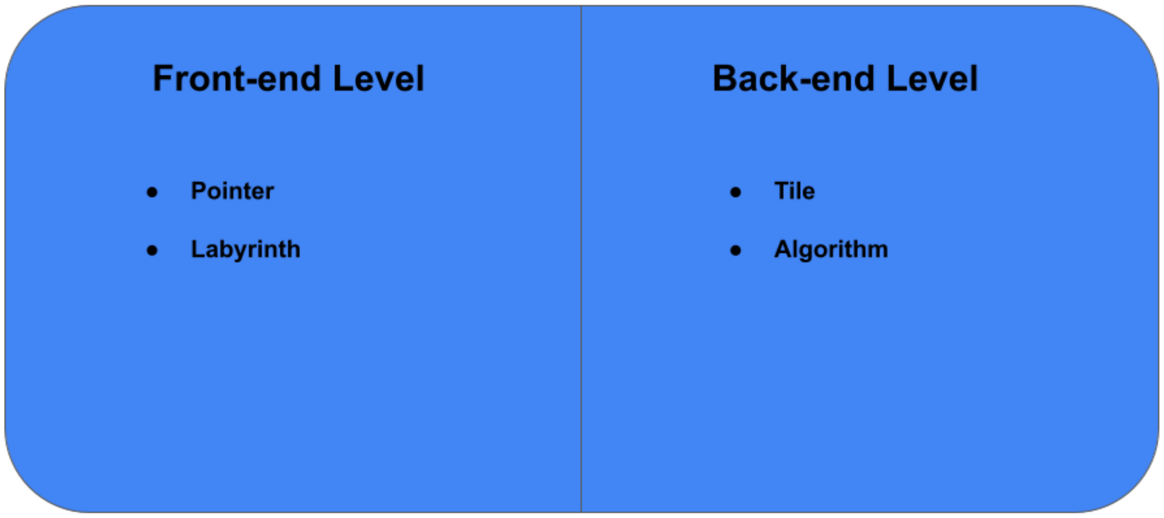

Figure 1. Front-end and Back-end class separation. These are organized according to their descriptions as principle classes.

3. Inter-agent Protocols

3.1 Constructing Labyrinths

The *Labyrinth* class will be a two-dimensional array of *Tile* class objects. The Labyrinth will be editable through the *Select(Int xloc, Int yloc)* method of the *Pointer* class, which will select a Tile object via its index in the array. These tiles in the Labyrinth can be set to obstacles through the *CreateWall()* method. The user can save a Labyrinth when done editing via a button which will trigger the *Save()* method that will store the Labyrinth data locally on the user's device in the folder the application is started from.

3.2 User Traversal

User traversal shall occur as an interaction between the *Pointer* and *Labyrinth* classes. Users will be able to manually cause movement of the Pointer through the Labyrinth, which will be done via the *Move(int movetype)* method to traverse adjacent tiles. Every move call will require *ValidityCheck(Labyrinth lab, int xloc, int yloc, int movetype)*, which will verify that the movement would not take the Pointer either out of the Labyrinth or into an obstacle. If the move would take the pointer onto the goal tile, the *LabyrinthComplete()* method will be executed.

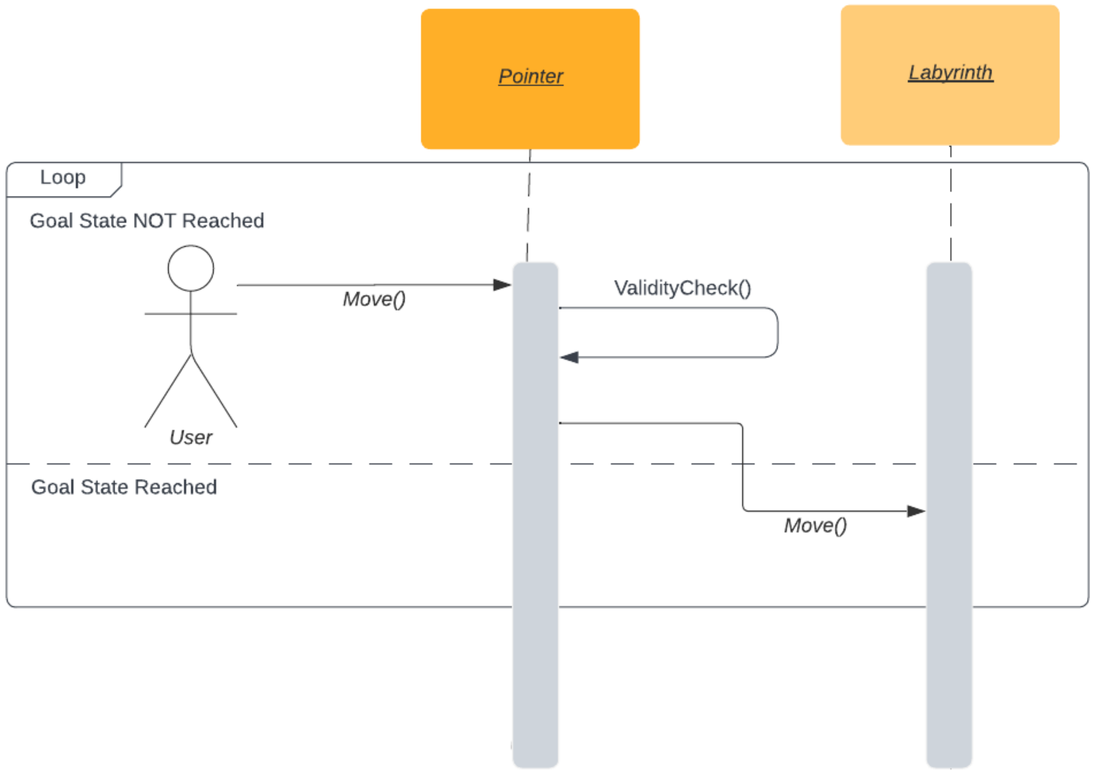

Figure 2. Sequence Diagram for User Traversal

3.3 Algorithmic Traversal

Algorithmic traversal shall occur similarly to user traversal, but the *Algorithm* will automatically execute move requests and will start traversal with the *NavigateLabyrinth(Labyrinth lab)* method, which will be the iterative solver of the labyrinth completed in accordance with the various algorithms.

4. Class Interfaces

This section will cover the interfaces for the classes planned for implementation into *Labyrinth of Algorithms*. These interfaces are not complete, but rather define loosely desired functionality of the product.

4.1 Tile

4.1.1 CreateWall()

This method will make the Tile into a wall obstacle that cannot be traversed by the movement of the pointer, fulfilling SF 3.1.1 of the SRS.

4.1.2 SetStart()

This method will set the Tile as the initial state in the Labyrinth where the Pointer will load in when the Labyrinth traversal is started, fulfilling SF 3.1.3 of the SRS.

4.1.3 SetGoal()

This method will set the Tile as the goal state of the Labyrinth that the pointer is supposed to be navigated to, fulfilling SF 3.1.3 of the SRS.

4.1.4 ClearTile()

This method will set the Tile as empty of any attributes and traversable by the Pointer movements, fulfilling SF 3.1.2 of the SRS.

4.2 Labyrinth

4.2.1 Tile AccessTile(int xloc, int yloc)

This method allows the user to access the specific Tiles of the Labyrinth for editing, which is necessary for FE-2 of the SRS.

4.2.2 bool CheckStates()

This method checks that the Labyrinth contains an initial state and a goal state Tile and that they are not in any obstacle space, which is required for the Save() method to successfully be carried out. This assures that the Software Quality Attributes from the SRS are upheld.

4.3 Algorithm

4.3.1 SelectAlgo(int algoindex)

This method will set the algorithm for navigation to the one desired by the user, fulfilling SF 3.2.1.

4.3.2 NavigateLabyrinth(Labyrinth lab)

This method will start the iterative solving of the selected labyrinth by the algorithm via movements of the pointer, fulfilling SF 3.2.2 and 3.3.1. This visualization will satisfy FE-5 requirements. If the goal state is not reachable, the traversal

4.4 Pointer

4.4.1 Select(int xloc, int yloc)

This method will place the pointer on the indicated Tile in the Labyrinth as long as that Tile exists, which will be checked by method 4.4.3. This is only available to the user while editing a Labyrinth.

4.4.2 Move(int movetype)

This method will move the pointer to an adjacent Tile in the Labyrinth. This method is only utilized during traversal of a Labyrinth by either a user or an Algorithm solution and is integral to FE-4 of the SRS.

4.4.1 *ValidityCheck(Labyrinth lab, int xloc, int yloc, int movetype)*

This method will be performed before any move is made and that move will only be executed if the call returns true. This method shall make sure that the pointer is not being moved out of the maze or into an obstacle Tile. If the tile moved into is a goal state then this will break and execute *LabyrinthComplete()*.

4.4.4 LabyrinthComplete()

This method will end the traversal of the Labyrinth.

5. Applicable Documents

Software Requirements Specification For Labyrinth of Algorithms (Version 1.2), September 20, 2022.

Appendix: Revision History

Date	Version	Review Outcome
10/02/22	1.0	*Initial Draft*
10/03/22	1.1	Introduction and Principal Classes finalized. Obstacle class removed and made into a feature of Tile class.
10/04/22	1.2	Final review for submission, additional graphics added for ease of reader understanding
12/5/22	1.3	Labyrinth generation has been removed, all editing takes place within the local session. - LL