FireAware

Version 3.3

Apr. 14,, 2019

Group 07

Rishi Arora, Jennifer Cheng, Esam Haris, Declan Wu

COMPSCI 2XB3, Lab 02

McMaster University, Department of Computing and Software

Report Revision History

Version Number	Date	Revisions Made
1.0	Mar. 15, 2019	Initial design document (Building, Read, and Sort modules)
2.0	Mar. 21, 2019	Added the Search module
2.1	Mar. 22, 2019	Modified Building module
3.0	Mar. 31, 2019	Added the Graph, GeoJson modules
3.1	Apr. 2, 2019	Removed GeoJson module - considered a design decision, not a requirement; added Application module
3.2	Apr. 5, 2019	Added UML class diagrams for Application and Graph
3.3	Apr. 14, 2019	Finished group contribution

Team Members

Rishi Arora, [redacted]
Full-stack developer, designer
Jennifer Cheng, [redacted]
Team Lead, designer
Esam Haris, [redacted]
Frontend developer, tester, log admin
Declan Wu, [redacted]
Backend developer, researcher

By virtue of submitting this document we electronically sign and date that the work being submitted by all the individuals in the group is their exclusive work as a group and we consent to make available the application developed through CS2XB3 project, the reports, presentations, and assignments (not including my name and student number) for future teaching purposes.

Contribution

Name	Roles	Contributions	Comments
Rishi Arora	Full-stack Developer, Designer	Application Read Building Testing Application	N/A
Jennifer Cheng	Team Lead, Designer	Requirements document Presentation Sort Search	
Esam Haris	Front-end Developer	Project log Testing Sort, Search	
Declan Wu	Back-end developer, Researcher	Graph Testing Graph	

Executive Summary

As technology evolves, companies incorporate increasingly more electronic elements into their venues. As a result, buildings are at a higher risk of a fire hazard. Although fire inspections should be a regular occurrence, some buildings go by unnoticed, and their fire safety is thus compromised. This project will inform users if a building is deemed to be safe or unsafe based on its previous fire inspection date combined with the typical frequency of fire inspections. Using open data from New York City, this web application allows users to search up a building and retrieve the dates that it has been inspected, as well as a metric of how safe the building is. The frontend is created using an open-source mapping library called Mapbox GL JS, and the Spring framework for the backend. Users will be able to see and interact with points on a map of New York.

Table of Contents

Overview of Modules

This project contains the following modules: **Read, Building, Sort, Search, Graph, Application.**

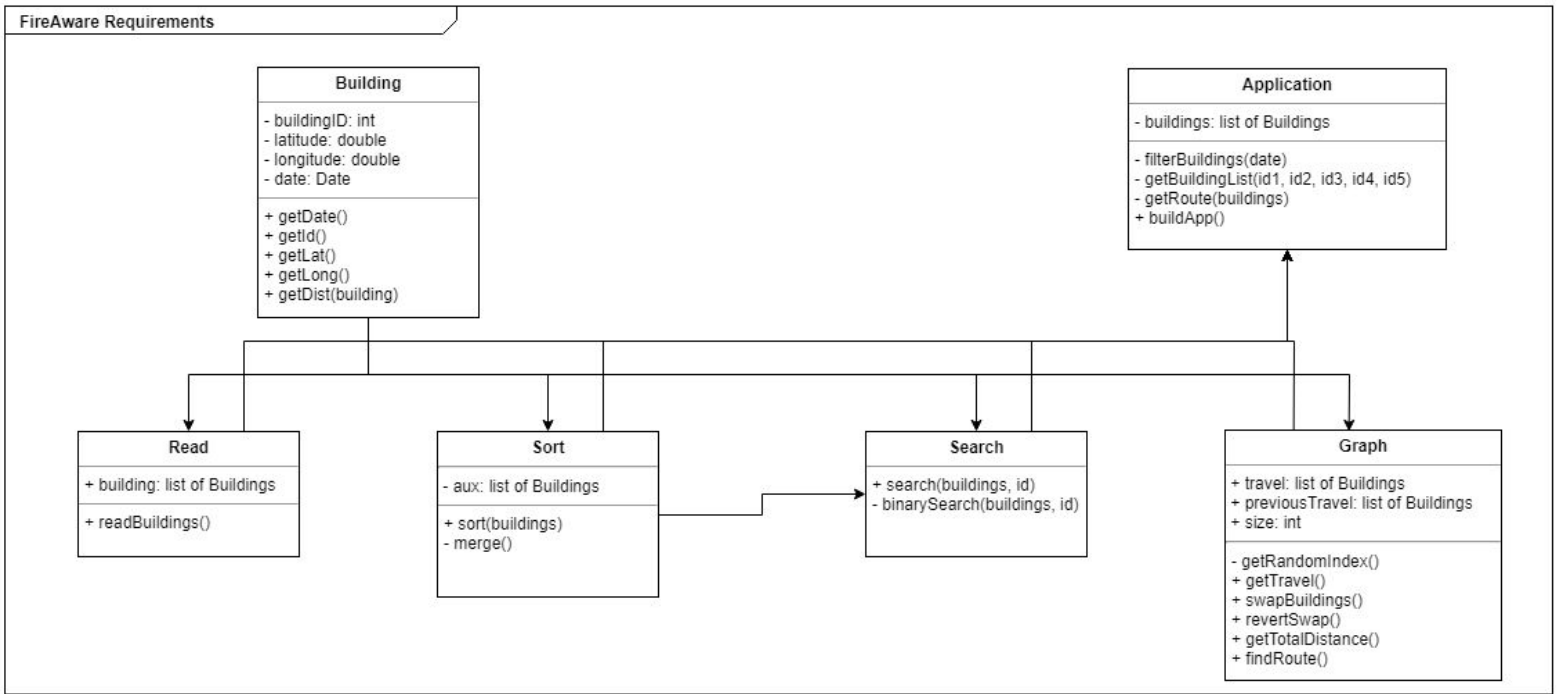
The modules were decomposed into the three main algorithms (Sort, Search, Graph), one main ADT to represent our main object (Building), a module to parse the data (Read), and one driver module to tie together the whole application (Application).

The algorithms are relatively self-explanatory:

- Sort is a module that contains merge sort, chosen for its stability and guaranteed performance.
- Search uses binary search, since our list of Buildings will be sorted and there is an easily comparable property of the buildings (building ID).
- Graph is created as an ADT, where the state variable is the optimal path between buildings. The Traveling Salesman Problem algorithm is used for this module.

The Building object contains the properties required to sort, search, and graph the buildings: latitude, longitude, building ID, and inspection date.
The Read module creates an array of Buildings by reading the data from the csv.
The Application module uses all aforementioned modules and connects the frontend with the backend.

Uses Relationship



Modules

Note that $None \rightarrow None$ represents no input/output, so the method just produces a transition.

Read

Comments:

This is the class that reads our dataset and parses it into a list of Buildings.

Uses:

Building, Sort, Search

State Variables:

building: list of Buildings

State Invariant:

None

Environment Variables:

input: csv file

Access Routine Methods:

(Public)

$String \rightarrow Building[]$

parse(file):

- Output: list of Buildings, created by the following method:
  - For all lines in the csv (excluding headers), create a new Building object using the building ID, latitude, longitude, and inspection date. Add that Building to a list of Buildings.
- Exception: IOException if file not found

(Private)

None

Building ADT

Comments:

This is the ADT that represents a building in NYC.

Uses:

None

State Variables:

id: int

long: double

lat: double

date: Date

State Invariant:

None

Environment Variables:

None

Access Routine Methods:

(Public)

new Building(id, long, lat, date):

- Transition: id := id, long := long, lat := lat, date := date
- Output: out := self
- Exception: None

$None \rightarrow DateTime$

getDate():

- Output: out := date

$None \rightarrow int$

getId():

- Output: out := id

$None \rightarrow double$

getLong():

- Output: out := long

$None \rightarrow double$

getLat():

- Output: out := lat

$Building[] \rightarrow Building$

getClosestBuilding(travel):

- Output: out := closest, where
  - closest := min(distance between current building and all buildings in arr)

$Building[] \rightarrow int$

getClosestBuildingIndex(travel):

- Output: out := index, where
    - index := index(min(distance between current building and all buildings in arr))

(Private)

None

Sort

Comments:

This class is required to effectively use the Search module.

Uses:

Building

State Variables:

aux: list of Buildings // for Merge Sort functionality

State Invariant:

None

Environment Variables:

None

Access Routine Methods:

(Public)

$Building[] \rightarrow Building[]$

sort(buildings)

- Output: out := using Merge Sort, s where
  - $\forall x \in |s| - 1 : s[x] \leq s[x+1]$

(Private)

$Building[] \rightarrow Building[]$

merge(buildings)

- Top-down implementation of merge sort
  - See Algorithms 4e. by Sedgewick & Wayne
- Output: out := s, where
  - $\forall x \in |s| - 1 : s[x] \leq s[x+1]$

Search

Comments:

This is the class that searches for a building by building ID. This satisfies the functional requirement of being able to search for buildings.

Uses:

Building, Sort

State Variables:

None

State Invariant:

None

Environment Variables:

None

Access Routine Methods:

(Private)

$Building[] \times int \rightarrow Building[]$

search(buildings, id)

- Output: out := bldg, where
  - bldg = Building(id ∈ buildings)

(Public)

$Building[] \times int \rightarrow Building[]$

binarySearch(buildings, id)

- Output: binary search

Graph ADT

Uses: Building

State Variables:

travel: a list of Buildings

previousTravel: a list of Buildings

size: int

State Invariant:

size $\leq$ n

- The fire companies are limited to examining n buildings a day, at most, where n is dependent on what the fire companies would like.

Environment Variables:

None

Access Routine Methods:

(Private)

$None \rightarrow int$

getRandomIndex():

- Output: out := random int from (0..size)

(Public)

new Graph(t):

- Transition: travel := t, previousTravel = t, size = t.length
- Output: out := self

$None \rightarrow Building[]$

getTravel():

- Output: out := travel

$None \rightarrow int$

getSize()

- Output: out := size

$None \rightarrow None$

swapBuildings()

- Transition: get two random indexes, a and b.
  previousTravel := copy of travel // in case we want to revert the swap
  travel[a], travel[b] := travel[b], travel[a]\

$None \rightarrow None$

revertSwap()

- Transition: travel := copy of previousTravel

$None \rightarrow double$

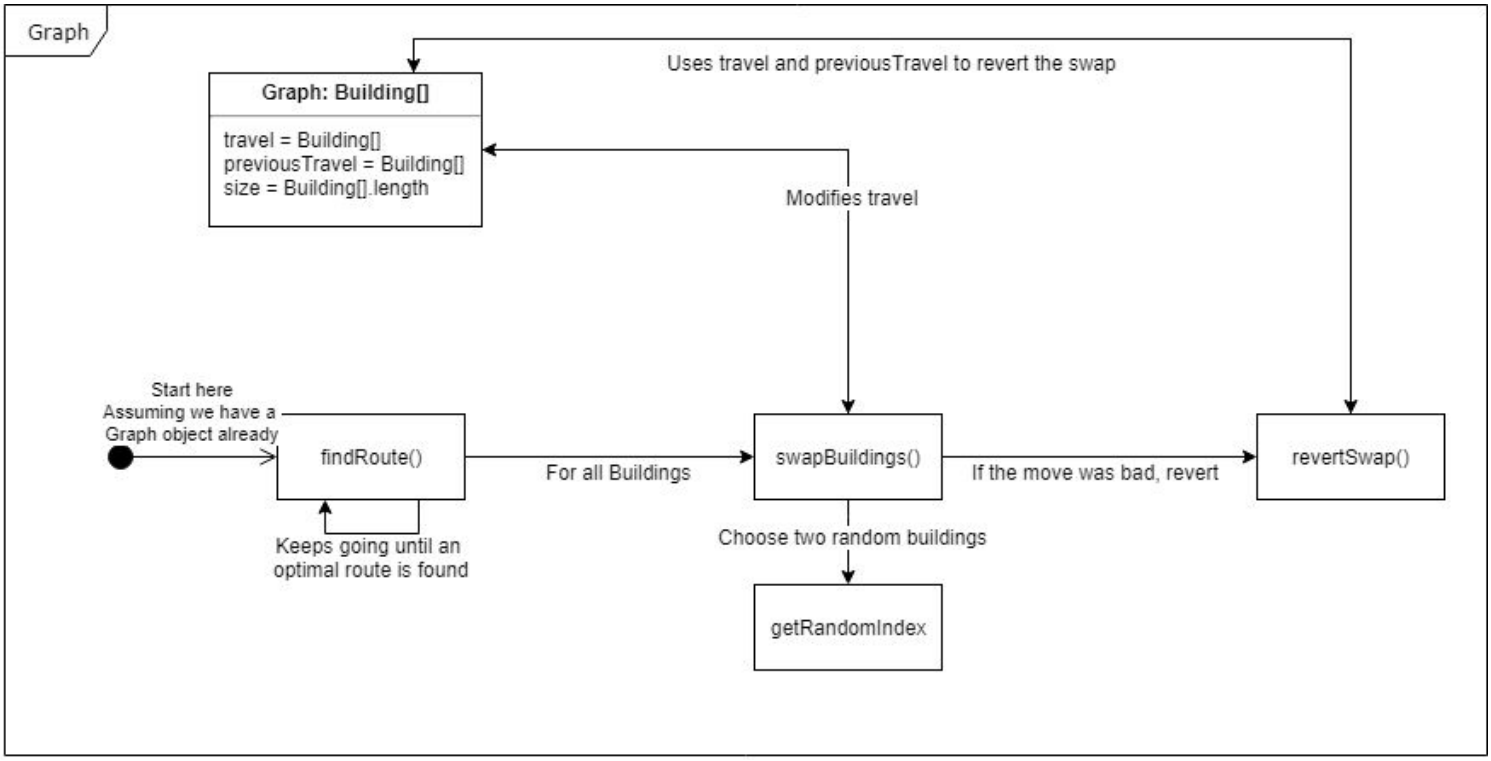
getTotalDistance():

- For all buildings in the route (travel), add the distance of each pair ((0, 1), (1, 2), .. , (size, 0)) to a variable called totalDist.
  Distance is calculated using the formula found here:
  https://www.movable-type.co.uk/scripts/latlong.html
- Output: out := totalDist

$None \rightarrow None$

findRoute()

- Using the Traveling Salesman Algorithm, modify the state variable *travel* to get the optimal path. See here:
  https://www.baeldung.com/java-simulated-annealing-for-traveling-salesman



Application

Comments:

This is the class that ties together all of the modules. It satisfies the functional requirements of buildings being mapped out, being able to click buildings and see their inspection date, filtering buildings by safety standards (unsafe/safe), and find the shortest route to travel through buildings.

Uses:

Building, Search, Sort, Graph, Read

State Variables:

b: list of Buildings

State Invariant:

None

Environment Variables:

screen: the screen of the user on a browser window

Access Routine Methods:

(Private)

$DateTime \rightarrow Building[]$

filterBuildings(date):

- Output: out := S, a list of filtered buildings, where $\forall x \in |S| - 1 : S[x].date() \leq date$

$String \times String \times String \times String \times String \rightarrow Building[]$

getBuildingList(id1, id2, id3, id4, id5):

- Output: out := list of {id1, id2, id3, id4, id5}

$Building[] \rightarrow Building[]$

findRoute(bList):

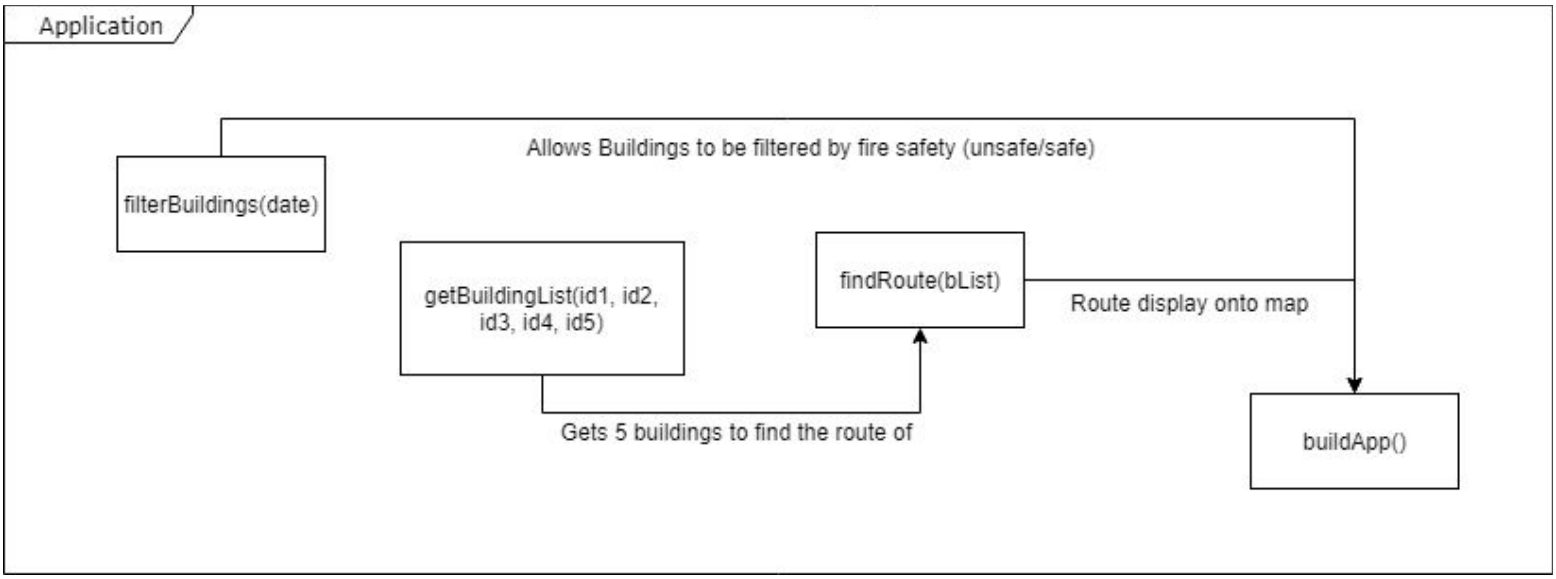
- Output: out := list of Building pairs of form {(u, v)} that connects buildings via the shortest path using Traveling Salesman (Graph) on the input, *bList* To do this, create a new Graph(bList) and use the findRoute() method.

(Public)

$None \rightarrow None$

buildApp():

- Transition: screen := print *b*, the list of buildings, on to the screen, as well as the control buttons (filterBuildings, findRoute)



Evaluation of Design

The design of this project, as mentioned earlier, segments the modules into four main parts: the algorithms, the data reader, the ADT, and the application itself.

The algorithms were segmented into three individual modules because they all had different functionalities. In the Search and Sort modules, the principle of information hiding was encapsulated by creating a public sort()/search() method, which calls a private method that has the actual algorithm (merge sort/binary search.) Both of these modules preserve minimality and essentiality, as their only function would be to sort/search the list of Buildings.

The Graph module was presented as an ADT, because the algorithm used (Traveling Salesman) requires there to be two variables to constantly be changed (travel and previousTravel); using an ADT allows us to make multiple instances of the shortest distance, in case the user needs multiple paths. The ADT is able to preserve the contents of travel/previousTravel between different sets of Buildings.

The Read module is simple, as it only contains one module, which has a single functionality: it reads the data, removes unnecessary fields (e.g. Borough), then creates Building objects to be put into an array. There could have been another method in the module to clean the data; however, it wouldn't be as simple as cleaning it in the same method. While essentiality was violated, the idea of separation of concerns makes it so that the user does not need to first clean then parse the data, and instead just do it in one step.

The ADT contains four fields: latitude, longitude, building ID, and date (inspection date). All of these fields have getters to preserve information hiding, as the state variables are private. This prevents the state variables from being modified externally. This module also contains methods used for Graph, which violates the aim of having low coupling; however, these methods make sense to be in the Building ADT, as they have multiple uses (e.g. finding the distance between two buildings), so this decision considers future improvements to the application, should it be deployed.

Lastly, the Application module ties all of the previous modules together. This module is essential to connect the back end with the front end so that the user is able to see a visualization of our data (one of the functional requirements). In fact, this module meets all of the functional requirements that were set out, because the main appeal of this application is having a visual representation of unsafe buildings in NYC, because it makes it easier for the user to determine if they're in a safe area.