Person-Hour Estimate

As a part of design we laid out a few components: a map, a pathfinder, a search, and runtime. The map is made up of several classes: nodes, floors, buildings, and a final map class to contain them all. These are primarily containers for data and shouldn't take much work, so we'll estimate 15 hours to put them together. The pathfinder is a large chunk of the meat of the program, so we'll estimate 15 hours for that component. The search function represents another large chunk of functionality so we'll estimate 10 hours to get it running. Finally, we'll estimate that all of the runtime code and documentation will take another 10 hours. This leaves us with a total of 50 hours for our time estimate. These estimates are based on how long we took to make similar-sized components in the previous projects.

Design Paradigm

For this project we chose the object-oriented design paradigm. The script for our project is broken down into classes, each of which handle a specific portion of the functionality. The broad sections of functionality for the program are as follows: storage of map data, pathing through the map, and the search function.

We create the map using a set of Nodes which represent vertices on a path. These node objects are stored inside a container which we call a Floor. Floors represent the levels of a building, and a set of Floors are contained inside of a Building class. Finally, there is a Map class made up of one or more Buildings. This method of data storage was a natural consequence of object-oriented design, as we examined the real-world object we were trying to model, a set of buildings on campus, and broke it down into logical components.

The pathing is done by one primary class and two helper classes. The Pathfinder class uses Dijkstra's shortest path algorithm to generate the shortest path from a single starting location to any other location on the map. To do this it makes use of a min priority queue class along with a queue element class. The queue element class wraps the thing being stored in the queue with a numerical priority.

The Search class is more functional than object-oriented. It is responsible for validating the input of the search bar and for finding the node associated with a searched location. The search class works in tandem with runtime code to initiate the pathing based on user input.

We chose the object-oriented paradigm for a few reasons. One of the primary reasons was that all of us primarily have experience working with object-oriented languages like Java and C++, so it was only natural for us to gravitate towards an object-oriented implementation for this project. Also, it was very intuitive to design the map data storage in an object-oriented manner, as it is easy to visualize how a map fits together using Node objects; it's essentially a game of connect-the-dots.

Software Architecture

For our project we decided to go with a 2-tier architecture to go along with our object-oriented design paradigm. The two tiers are the client and the second is the data layer stored within the program. The client can interact with the website in a variety of ways which affect what occurs.

The program is displayed through a website which is put on the internet. The users can open this in whatever web browser they have access to, including using their phone as it is a responsive layout. This means that the user client can interact with the program in multiple ways from the web.

The data-layer contained within the program is what changes the website in response to what the client does. The intent of the program is to allow users to enter in the location of a classroom inside the engineering complex and to get detailed instructions on how to get there. Right now, for the prototype, we are only modeling one floor of Eaton Hall to demonstrate the technology and vision. For the final version we envision a complete, multi-floor Eaton Hall implemented. However, this could get applied to the entire engineering complex or campus in general.

The interactions that the client can currently do are to set the starting location, enter in a classroom, and to visit one of our githubs. The starting locations are preset from a drop-down menu in this prototype. It also includes things such as the elevator and staircase, which in the final version will probably not be an option as those two will be done from behind the scenes if trying to traversal multi-level.

For the classroom interaction, the user has to provide a valid classroom or else the program will return to the client that it was an invalid string. This is one example of the interaction between the client and the program in our 2-tier architecture. The 2-tier architecture allows us to have a flexible program which can deal with multiple client instances while having consistent intended results.

design patterns

For this project we used many design patterns in the creation of this project. They are Factory method, Singleton, Composite, Decorator, and Iterator.

We used the Creational Design Pattern Factory method by making a Building object. The Building object has multiple floors in it and we only instantiate a floor when we are going to use it for pathfinding.

We used the Creational Design Pattern Singleton by making a floor search object. We use that object to see if the user input is valid or not before it goes to the pathfinding object.

We used the Structural Design Pattern Composite when we made the floor object. Each floor has an array of nodes that contains an x and y location and if it is an end node a name of that location. The nodes are linked together by vertices so each node knows what other nodes are connected to it.

We used the Structural Design Pattern Decorator in the node object. Each node has an x and y location in it but there is a subclass of nodes called end nodes and they have an x and y location as well as a name and room number in it.

We used the Behavioral Design Pattern Iterator in multiple parts of our project. The first place we use it is in the pathfinding algorithm to see what nodes are connected to other nodes. The second and third time we use it is in the search class. The first iterator in the search class iterates over the node array to see if the user input is in the node array or not. The second iterator in the search class also iterates over the node array but it looks for the node number that matches the user input so that we can run the pathfinding algorithm with that node as the end point.