U-Play: Programming Test

Due on Monday, March 27, 2017

BlueByte - Uplay 8:30am

Nick Houghton

Introduction

The *Acme Elevators* challenge was a fun and interesting challenge. This report will be an attempt to describe my thought process and procedure while working towards a solution. What follows will be a step by step recording of what took place.

Preparation

When the *Acme Elevators* challenge was received the first step I took was to organize the given code into a visual studio project. Being a windows user I've found Visual Studio (VS) to be one of the most enjoyable C++ development tools. The new project was included along side a series of other projects of a similar nature for which I have a github repository for safe keeping and version control.

Once the project was set up I began reading through the code however I found the formatting to be unfamiliar. I spent some time doing some simple rearranging to put the received code into a more familiar and comfortable format. Once finished I returned to carefully re-reading the problem statement and getting familiar with the code.

Implementation

Once I had reviewed the incomplete code several times I made a list of all of the locations of the "Implement Me" comment. I then added a *TODO* keyword to each so I could easily track them using visual studio's Track List feature. As I progressed through implementing the missing code I added a 'Done' tag to each *TODO* comment.

Elevator

Once I became familiar with the code structure I began looking at the *Elevator* class since it seemed the most 'low-leveled' in the class structure. I quickly noticed that it was missing a member which kept track of the destination floor of the elevator instance; I could not see a logical way of directing elevators without one. With this additional member variable the remaining implementation was relatively straightforward. Additionally, I noticed that there was no clear definition of number value was used to indicate the bottom floor. In North American culture the ground floor is commonly referred to as the 'first' floor however in Europe it is commonly floor zero. I added a simple get method in Utils.h where the ground floor value can be set and retrieved.

It seemed that the easiest way to determine whether an elevator has work to do was whether its current floor equals its target floor. Theoretically, should these equal the elevator is idle. The step function seemed to need to only modify the current floor value by one on each iteration. Assuming the elevator 'HasWork' a check is enacted to ensure it does not go outside of its bounds. Finally, to select a floor a simple check needed to be done to ensure that the selected floor was attainable by the current elevator. Should it be out of range an error message is printed to the standard error stream.

Elevators

The *Elevators* class proved to be slightly more confusing at first. The intended purpose of the "OnMessageElevatorCall" and "OnMessageElevatorRequest" methods was a little vague at first however I surmised the following:

- **OnMessageElevatorCall**: A human standing in the lobby presses either the "up" or "down" button to call an elevator to take them in a certain direction.

- **OnMessageElevatorRequest**: A human has entered the elevator and selects a desired floor.

I based these interpretations off of the contents of the message objects sent to each. The members of the "MessageElevatorCall" and "MessageElevatorRequest" classes seemed to fit this interpretation well.

Following these definitions, the "OnMessageElevatorRequest" method was relatively straightforward to implement. It simply needed to search the vector of elevators, find the requested instance and set its target floor value via the "SelectFloor" method.

The "OnMessageElevatorCall" method was considerably more complicated. My initial thought was that this method should populate a queue data structure. As humans arrive in the lobby to call an elevator the messages should be prioritized and acted upon in the order they arrived. However, it came to mind that the not all messages can be serviced immediately. As an example, if there is only a single elevator, currently on the 4th floor headed towards the 8th floor it is able to stop at any floor in between to pick up additional passengers. A request from a floor below will not be serviceable as the elevator is traveling away from the call source. In this scenario, a call from the second floor will need to wait until the elevator completes its current objective and returns to an idle state.
To implement this a queue which allows entries in the middle of the queue to be removed is needed. The STL Queue container does not allow this so a Vector was used. At every "Step", the *Elevators* class iterates through the vector from oldest to newest and services (as well as removes from the list) any call it is able to service.

The elevators class must accurately asses the state of all elevators and make a decision which messages are serviceable and how best to do so. A call to a method named "ServiceElevatorCalls" was added to the elevator step method before the step loop to assess calls. This method calls another method named "canService" on each message to determine if that message can be service. The "canService" method is relatively complex as the best decision for the current message needs to be made. The method loops through each elevator object and evaluates only those that are idle. Non-idle elevators are currently servicing another request and should not be altered. To decide which idle elevator is best to service the request a series of conditions are determined:

- The current distance away from the calling human.
- Whether the elevator is currently above or below the calling floor.
- Whether the person wants to go up or down.
- Whether the person is on the extremity of the elevators shaft (bottom or top floor).
- Whether the requesting floor is reachable by the current elevator.

If the calling floor is not reachable by the current elevator then it obviously can not be considered and is skipped. The remaining four conditions are evaluated using a boolean logic equation. A small java application named *Truth-Table Solver* was used to convert a truth table to the desired boolean formula. As can be seen in Figure 1 the truth table was filled, the solver was then able to compute a sum of products formula. Any elevator which meets the desired conditions is considered as a candidate to service the incoming call. A temporary distance variable is used to track the nearest elevator which meets the conditions.

The methodology described to select the best elevator relies on some complex logic. A unit test using the "Catch" C++ unit testing framework was used to verify the logic's validity. The initial test case for the "CanService" method can be seen in Listing 1. Several other test cases for the *Elevators* class methods were added but will not be included in this report.

Listing 1: Can Service Test Case

```
TEST_CASE("Can Service") {
        Elevators elev;
        std::vector<Elevator>   myElevators;

        // Test idle elevators
        myElevators.push_back(Elevator{ 1, 10, 6, Direction::Down });
        elev.setElevators(myElevators);
        MessageElevatorCall mTrue1, mTrue2, mTrue3;
        mTrue1.myDirection = Direction::Up;
        mTrue1.myFloor = 1;
        mTrue2.myDirection = Direction::Down;
        mTrue2.myFloor = 7;
        mTrue3.myDirection = Direction::Up;
        mTrue3.myFloor = 7;

        REQUIRE(elev.canService(mTrue1) == true);
        myElevators.clear();
        elev.setElevators(myElevators);
        myElevators.push_back(Elevator{ 1, 10, 6, Direction::Down });
        elev.setElevators(myElevators);
        REQUIRE(elev.canService(mTrue2) == true);
        myElevators.clear();
        elev.setElevators(myElevators);
        myElevators.push_back(Elevator{ 1, 10, 6, Direction::Down });
        elev.setElevators(myElevators);
        REQUIRE(elev.canService(mTrue3) == true);

        myElevators.clear();
        elev.setElevators(myElevators);


        // Test Moving Elevators
        Elevator movingElev(1, 10, 1, Direction::Down);
        movingElev.setTargetFloor(10);
        myElevators.push_back(movingElev);
        elev.setElevators(myElevators);

        MessageElevatorCall mTrue4, mFalse1;

        mTrue4.myDirection = Direction::Up;
        mTrue4.myFloor = 4;
        mFalse1.myDirection == Direction::Down;
        mFalse1.myFloor = 4;
        REQUIRE(elev.canService(mTrue4) == true);
        REQUIRE(elev.canService(mFalse1) == false);
}
```

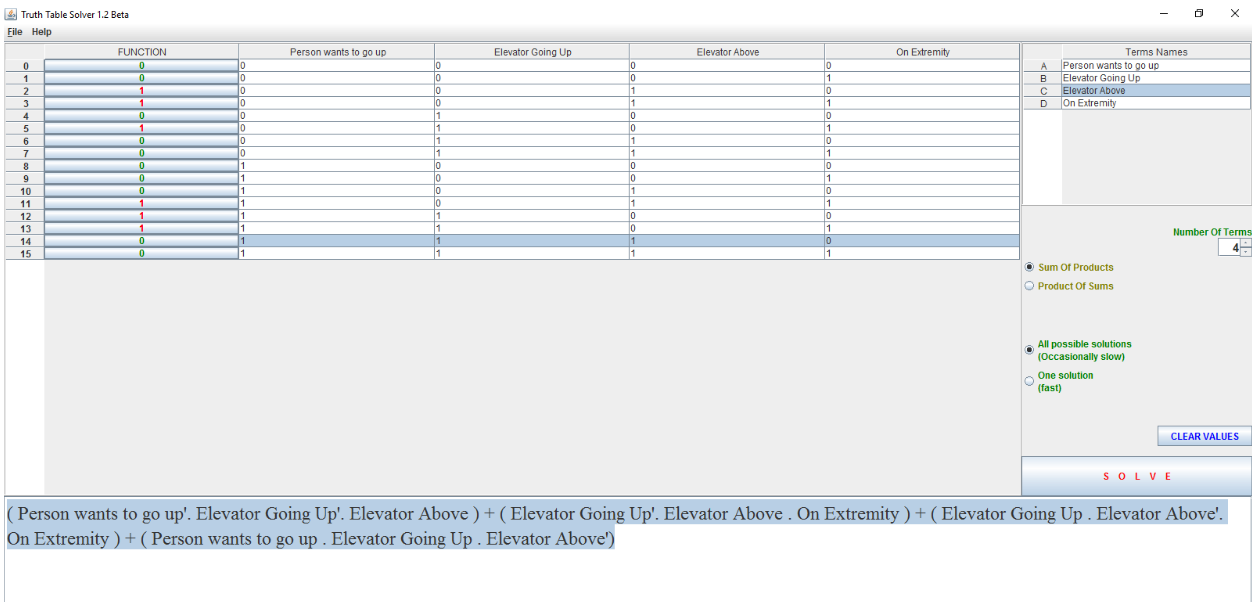

Figure 1: Truth Table Solver Use

Humans

The *Humans* class required only two methods to be implemented. I began by working on the "OnMessageElevatorArrived" method. I gathered that there were only two cases for which this method should be called:

1. When an elevator arrives to pick up a human.
2. When an elevator reaches its destination to drop off a human.

Knowing these two cases, simple checks can be easily implemented. When an elevator arrived message is received it must be checked against all humans. For each human the two conditions listed above are checked. If a person is in the waiting state and the elevator has arrived to their floor, that person transitions to the traveling state and the elevator becomes occupied. Currently, only one person can ride in an elevator at a time. Hopefully this will be remedied later. When a person is picked up a "MessageElevatorRequest" message is generated and broadcast to the elevators. This emulates the notion that the person has boarded the elevator and has selected a destination floor.

In the case where the person is in the traveling state and the elevator has arrived to their destination floor, the second case stated above has been met. The human transitions to the arrived state.
The "OnMessageHumanStep" method only required handling the cases where a new human arrives and the removal of humans who have completed their journey. When a human is created the constructor sets the state to "HumanState_Idle". The step method looks for humans in the vector with this state. If found the a call is made to the elevators and the humans state is changed to "HumanState_Waiting". A test case was added for the "OnMessageElevatorArrived" method to ensure proper function. Once all of the test cases passed the entire system could be run for the first time. A test case was not implemented for the human "OnMessageHumanStep" because it was the last of the implementations needing completion; it could be tested by running the entire system.

First-Run

When run for the first time some issues were quickly discovered. In the *Elevators* class, the "canService" method was not properly discerning between idle and active elevators. Secondly, a case was not implemented for when a human arrives and an elevator was already on their floor. These two bugs were found and easily remedied. To test the system the initial set up of one elevator and one human was run. Some additional code was added to the "OnMessageHumanStep" method; when a human arrives and is removed from the "myHumans" vector a new human was generated randomly. This allowed the *Elevators* class to run continuously, handling calls and requests as they came in. They system ran successfully handling each new human who arrived.