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 the thought process and procedure used 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 taken was to organize the given code into a visual studio project. The new project was included along side a series of other projects of a similar nature for which there is a github repository for safe keeping and version control. It was found that the received code was in an unfamiliar style; some effort was made to reformat it into a more familiar and comfortable way. This reformatting was done to improve efficiency of development. Once the project was set up the requirements in the README were read several times. Relating the described task to the incomplete code took thorough analysis of the code and several diagrams. The majority of the first several hours of development were spent on understanding the task.

Measure twice, cut once.

Implementation

Once the set-up and review stage was complete a list of all of the locations of the "Implement Me" comment was made. A *TODO* keyword was added to each so they could be easily found and tracked using visual studio's Track List feature. As development progressed a 'Done' tag was added to each *TODO* comment. This allowed for a efficient, more manageable development experience.

Elevator

From the preparation process it was found that the *Elevator* class was the most "low-leveled" in the class structure. It was decided that this would be an effective place to begin. It was noticed that the class did not have a member which kept track of the destination floor of the elevator instance. Though there may have been other ways to implement this class, the addition of this member seemed to be the most expedient and simple method. With this additional member variable the remaining implementation was relatively straightforward.
Near this time it was additionally noticed that there was no clear definition of the number value used to indicate the bottom floor of the elevator shaft. In North American culture the ground floor is commonly referred to as the 'first' floor however in Europe the first floor is commonly the first floor ABOVE the ground floor. A constant variable was added in Utils.h where the ground floor value can be set and retrieved. This allows for a more dynamic definition

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. This logic was used to implement the "HasWork" method. The step function seemed to need to only modify the current floor value by one on each iteration. For each call to "Step", a check was done to determine direction. The current floor value was incremented/decremented accordingly. Assuming the elevator 'HasWork' an additional 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. In theory, a building may have elevators of different heights. 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 the following was surmised:

- **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.

These interpretations were based off of the contents of the message objects they received; the members of the "MessageElevatorCall" and "MessageElevatorRequest" classes seemed to fit this interpretation.

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. Initially it was thought 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 assess 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 waiting calls. This method calls another method named "canService" on each message to determine if that message can be serviced. 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 were 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

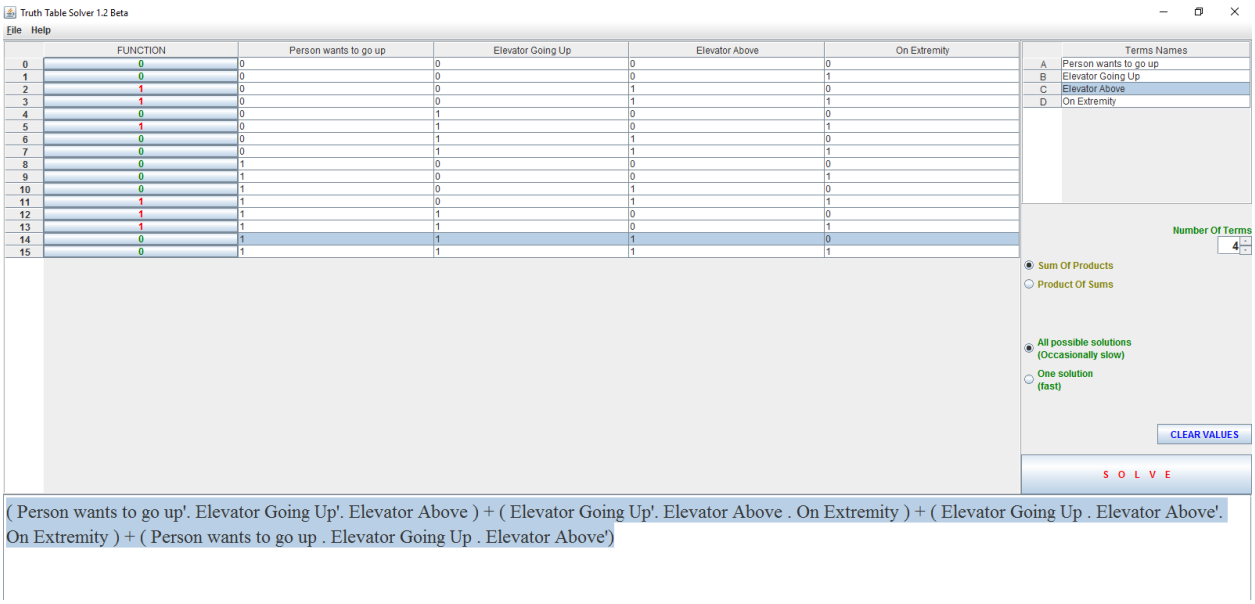

Figure 1: Truth Table Solver Use

formula which was used. 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.

Humans

The *Humans* class required only two methods to be implemented. The "OnMessageElevatorArrived" method was implemented first. It was 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 were 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. This implementation only allows for a single human to ride an elevator at a time. It was hoped that this could be remedied. When a person is picked up a "MessageElevatorRequest" object is generated and broadcast to the elevators. This emulates the notion that the human 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 a

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);
}
```

call is made to the elevators and the human's 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. The system ran successfully handling each new human who arrived.

Increased Number of Humans

To ramp up the abilities of the system an additional human was added to the "Humans::Start" method. This immediately showed a flaw; when the first human arrived at their destination the elevator was not being rerouted to pickup the second human. This issue was caused by the "canService" method. When an elevator arrived at the location to pick up the first passenger it caused the "HasWork" method to return false because the target and current floors were equal. This made the *Elevators* class think that it could remove the second passengers call from the queue. This was incorrect as there was no elevator to service the second call. Once the elevator has finished with the first passenger it checked the queue to find it empty.
A solution to this was found by adding an "onCall" and "onRequest" flag to the *Elevator* class. This allowed a check to be implemented to prevent the call queue from being modified when there is no elevator to service the message.

Increased number of elevators

When increasing the number of elevators, if the max floor value were all equal the system worked without issue. When elevators with different floor counts were created there was an issue with the "OnMessageElevatorArrived" and "OnMessageElevatorRequest" methods. When an elevator arrives it sends a "MessageElevatorArrived" object. This object only contains the elevatorId and the floor it has arrived to. There is no indication of the maximum floor this elevator can reach. In the "OnMessageElevatorArrived" method, a human uses the elevator that has arrived to their floor. Once inside, they find there is no way to get to their floor. This causes an issue as the human never reaches their destination. The only solution would be to add an additional parameter to the "MessageElevatorArrived" class.
This presented a second issue. Which elevator would service a call was based upon its availability and its proximity to the human who made the call. This decision is not based off of the end destination of the user; it can't be as the destination buttons are within the car of the elevator. The consequence is that when an elevator arrives to service a human who wants to go out of the range of that elevator it is now both on the same floor as the human and available. All other elevators (who may be able to service the human) are further away. Since it is not desirable to add a destination variable to the "MessageElevatorCall" class an alternative is needed. The best solution would be, when a call is made by a human, all elevators who do not have work will have their target floor set to the calling human's floor. This way, all available elevators will begin to travel towards the waiting human. At this time it was realized that the implementation for this

issue was left as a bonus. With the current structure of the code a solution was non-trivial so was left as a thought exercise.
Further, an additional bug was found in the "canService" method. After an "active" elevator delivered a human it would never pick up another. The bug was a complex issue with the logic described regarding the selection of elevators. Due to a lack of time, the ability to have an "active" elevator pickup a passenger on its way to a floor was scrapped. The "canService" method was modified to only select idle elevators. This introduced increased wait times for humans however it ensured the system worked as expected. Given more time this would be a desired feature to add.

Code Clean-up And Optimization

Donald Knuth, winner of the 1974 Turing award once said that "premature optimization is the root of all evil". This a sound philosophy which often makes development considerably more enjoyable when followed. Once the system was working relatively well it was reviewed for performance and cleanliness. Additional comments were added and unit tests were used to ensure proper functionality was maintained. By this point of the challenge the 48 hours was running low; code can almost always be better. The following is a list of general C++ code facets which were used to ensure a better performance.

- Use of prefix operators over postfix.
- Use of constructor initialization lists.
- Parameters passed by reference where possible.

Discussion

Unit Tests

Each of the unit tests have been commented out and moved to the bottom of the .cpp file of their respective classes. These tests employ the "Catch" framework's TEST_CASE macro which require the catch_main function to run. The catch_main is a main class within the "Catch" framework; when it and the system's main function are both present they are considered overloaded which, of course is not allowed. To run the test cases they must be uncommented. The main method for the system in main.cpp must be commented out to allow the catch_main to run. Finally, at the top of there is a "#define CATCH_CONFIG_MAIN" commented out at the top of main.cpp. This as well must be uncommented to run the test cases.

TODO List

Throughout development the TODO task list feature of visual studio was used extensively. These comments were intentionally left within the source code to give the reviewer a sense of the thought process during development.

Bonus Questions

Thread Class Enhancement

The current implementation of the threads relies on a timing construct to update the state of the elevators and humans. Since elevators take time to move their progress should still be based on time however the processing of input from users and elevators should not be delayed while waiting for the step methods to be run. Ideally the worker threads would sleep on their respective condition variables, waiting from input from other entities. They currently sleep until a predefined time and then wake to possibly do nothing. Both

the *Elevators* and *Humans* classes should be given methods which send receive signals via the condition variables.

Clean Exit

Currently the two worker threads run on an infinite loop. The only way to end the program is to force kill it. This is undesirable as it leaves stack objects for each thread in a random state which may or may not be clean-able. Ideally there would be a condition to end the while loops in the worker threads. Perhaps some unique human or elevator which triggers the end of both loops. From there it would be best to allow the worker threads to complete their execution. A join statement should be placed in the main function to allow both threads to fully complete and destroy before the main thread exits.

Different Sized Buildings

During development it was not realized that this feature was a bonus question. An attempt was made to implement a system such that the elevators could have a variable number of floors. This would emulate a building where a the primary elevator can not access the penthouse suite while a second secret elevator can. It quickly became apparent that the current code would need some restructuring in order to allow this feature. First, the messages passed between the human an elevator worker threads would need to change slightly.