Exercise 3 : Single Elevator

This exercise consists of two parts:

1. A practical milestone, after which you should have a single elevator servicing orders.
2. A project brainstorm, where you work on your plans for modularization.

Which part you do first is up to you – you may find it useful to get some code in an editor to realize exactly what parts are missing and will need to be designed and created, or you might find it useful to think long and hard before writing something that might end up getting thrown out. The most likely scenario is a combination of both.

Part 1: Running a single elevator

The elevator hardware in the lab is controlled via an Arduino, connected with USB as a serial device. On top of this raw IO layer, there is an elevator layer, which exposes functions for reading buttons, setting the motor speed, and so on.

Since we have a limited number of elevators (and you can't take them with you), we also have a simulator. Since the simulator needs to show what it does and take inputs to emulate button presses, it must be run in a separate window, which means it needs to be a standalone application.

In order to make swapping between the simulator and the real elevator as seamless as possible, we have chosen to make interfacing with the real elevator also happen through a standalone application. This also means you do not have to interface with any C code, but instead just a single TCP connection on localhost.

This means we have a simple client-server structure to the elevator:

- Two possible servers (both are already installed on the lab computers):
  - The Elevator Server (https://github.com/TTK4145/elevator-server)
  - The simulator (https://github.com/TTK4145/Simulator-v2)

- Language-specific clients (https://github.com/TTK4145?q=driver)
  - Choose the one you need for the language you are using on the project
  - (If none exists for your language, ask for help and we'll add it once it works)

You may want to modify the client end of the driver, or possibly create your own from scratch. There is no particular requirement or recommendation involved here.

If you for some reason want to interface directly with the elevator (i.e., bypassing the elevator server), you can find the necessary code to do this in the elevator-server repository. This is not recommended.

Up and down

Download the driver (for the programming language you are doing the project in) and test it on both the hardware elevator and the simulator.

- Using the hardware elevator
  - Run the elevator server, by typing `elevatorserver` in any terminal window.
  - If an `elevatorserver` is already running, the new server will not be able to bind to the socket. If you need to kill it, you can do so by calling `pkill elevatorserver`
- Using the simulator
  - Run the simulator server, by typing `simelevatorserver` in any terminal window.
  - Or if you are working from your own machine, download and run the simulator executable (https://github.com/TTK4145/Simulator-v2/releases/latest)
    - If you are on OSX, you will have to compile it from source yourself. See instructions here (https://github.com/TTK4145/Simulator-v2#compiling-from-source).
  - In order to run multiple simulators on the same computer (or a simulator and a real elevator), you will have to change the port on both the simulator (with `--port`) and in the driver (likely in a call to some init-function or in a config file)

If you want to use the real elevator with your own machine, you will need a USB-B cable, and the elevator server executable (https://github.com/TTK4145/elevator-server/releases/latest) (OSX is not supported, as this is not implemented yet).

Up and away

The elevator project can be roughly divided into two parts: Distributing the incoming requests (hall and cab calls) to the elevators and then servicing those requests. At this stage, you don't have the functionality implemented for the first part, but the latter part was a project in TTK4235. Since not all of you have taken this course, we'll have to get you up to speed on both the solution to this problem and the preferred implementation pattern.

The relevant part of that project is documented in Project-resources/elev_algo (https://github.com/TTK4145/Project-resources/tree/master/elev_algo).

Implementing the "single elevator control" component as a state machine is the preferred pattern, to the point where we might even dare call it the definitively correct approach. The details and analysis of this pattern are covered in greater detail in the lectures, but here is the short version of how to follow it:

- Analysis:
  - Identify the inputs of the system, and consider them as discrete (in time) events

    - Identify the outputs of the system
    - Combine all inputs & outputs and store them (the "state") from one event to the next
        * This creates a combinatorial "explosion" of possible internal state
    - Eliminate combinations that are redundant, impossible (according to reality), and undesirable (according to the specification)
        * This should give you a "minimal representation" of the possible internal state
    - Give names to the new minimal combined-input-output states
        * These typically identify how the system "behaves" when responding to the next event
        * Leave any un-combined data alone
- Implementation:
    - Create a "handler" for each event (function, message receive-case, etc.)
    - Switch/match on the behavior-state (within each event handler)
    - Call a (preferably pure) function that computes the next state and output actions
    - Perform the output actions and save the new state

You are encouraged to try to trace the analysis steps for the elev_algo code linked above, but you may also find that the vastly less rigorous approach of intuition quickly overtakes the methodical one. But for the implementation side, you should take a much closer look, especially on why we consider events first then state (as opposed to state-first), and where the divide goes between "code that is directly in the event-handler" and "code that is in a function called by the event handler".

Doing it yourself

You should now implement some way to control a single elevator, as a part of the elevator project. This is where you get started "for real", so set up your environment (build tools, repository, editor, etc.) the way you like it before you begin. Start using a version control system like git, and make sure to back up all your data. The computers at the lab might be reset at any time.

Since you don't have any way to distribute requests yet, you should use the button presses directly. You will also eventually have to implement features for various fault tolerance scenarios, button light handling, and so on. Since so many things will have to change later, use this knowledge to influence your module design and to find the boundaries between them.

Part 2: Project Brainstorm

This part does not have to be handed in, but you may find it useful to discuss with the student assistants.

At the end of the previous exercise, there was a list of questions and considerations

related to the networking aspect of the project. Figuring out what you need to communicate between the elevators is already a hard task, but you will at some point also have to do the even harder task of figuring out how to make it happen in code.

Below is another such list of questions and considerations, but aimed instead at what your program does (on an "individual" level), and how it is structured.

- The implementation
  - What is a module made of? Is a "module" a class, thread, file, function, . . . ?
    - This will depend on your programming language of choice.
  - These modules are part of the same software system. How will they interact?
    - Methods, functions, messages, channels, network, shared variables, file systems, . . . ?
- Data from the outside world
  - Something needs to be sent and received on the network
    - Are the network messages different "types" with different kinds of content?
    - Should the program do different things when receiving different kinds of messages?
  - You also get data from (and to) the elevator hardware
    - Should the different kinds of data feed into the same "top-level" module?
    - Or should they go directly to their own specific modules?
- The contents
  - Briefly, and as "seen from the outside": What modules do you need?
    - As in: what are the things that need to be done?
      - Try giving them names. A concise and descriptive name can be a good indication of a high-quality module.
    - Consider: what are the criteria you have used for deciding when to split one thing into two (or more) modules?
  - For a module to perform its task, it depends on information. What are the inputs, outputs, and state of the modules?
    - Think in terms of data structures, or at least "what the data is" (even if it has no structure yet).
    - Given the language and implementation choice - are you playing to the strengths of your choice?
      - That is: you are not trying to share variables with message passing or send messages with shared variables.

Some hints:

- If your module collection is just "elevator interface, networking, and a magic box that does the rest", you should create more modules.
- Do not consider how a module does what it does (i.e. algorithms and logic). For now, stick to what it does, and what it needs in order to do it.

- Try to draw the interactions between modules as a figure, in no more detail than circles for modules and arrows between them.
- Try tracing some scenarios like "button is pressed" to "elevator starts moving" in the figure, as this will often help you figure out what is missing.
- Be aware of what is the least clear (or most hand-wave-y) part, and ask for ideas and inspiration for it.