Elevator Lab Report - TTK4235

Group 43: H. Slottan, J. Jahren Lauvrud, A. Farner

March 2024

1 Introduction

In the modern age, the elevator has become an ubiquitous commodity. Initially requiring an operator, there has since their time of creation arisen a need to create autonomous control systems capable of operating these handy machines with minimal human interaction.

In this report, we will detail our design for a software controller that tries to fulfills this purpose. The system is designed for a single elevator that service four floors, with behavior specified in the FAT requirements presented in the lab project notes for the elevator lab in TTK4235 [1]. The system has been designed using a slightly modified version of the V-model. UML diagrams diagrams were used to conceptualize the system during the design process, as well as to explain the system behavior. The system was designed to be modular, with each module being responsible for their own specific subsystem.

The controller was written in the C language, using git and GitHub as a version control system. Since documentation also was a consideration during production, Doxygen was used while writing the code to better clarify code function. The basis for development was a skeleton project handout [1], which will not be explained in this report.

2 Design

2.1 Architecture design

2.1.1 Core behavior

At an architectural level, the design of the elevator is quite simple: The elevator has two queues, one for requests for movement in the *up* direction, and one for requests in *down* direction. These queues hold a simple boolean yes or no value for each floor. The elevator also has two modes of operation, called service modes. The service modes are *UP* and *DOWN*. Whilst in the *UP* service mode, the elevator stops at floors found in the *up*-queue. Similarly, when the elevator is in the *DOWN* service mode, the elevator stops at floors found in the *down*-queue. The elevator does not have a target floor to which it is moving. Whenever the elevator reaches a floor, it simply checks the relevant queue for requests. Therefore, most of the complex behavior of the elevator is confined to deciding which direction the elevator should start moving after it has stopped and deciding whether the elevator should stop. The state *current floor* always refers to the most recent floor reached by the elevator.

The elevator control system is divided into five different modules, each with their own responsibilities. This is demonstrated in the class diagram, shown in Figure 2

- The Input Handler checks and processes button input.
- The Elevator Controller contains the main loop, and implements all operational logic
- The Queue module contains both queues and can modify them.

- The Light Controller switches lights off or on
- The Motor Controller starts and stops motor movement

The most complex problem that has to be handled by the architecture is the elevator behavior whenever the service mode switches. Service mode switch occurs when no floor are found in the corresponding queue, beyond the current floor (so in *UP* mode, no requests **above** *current floor*. In *DOWN* mode, no requests **below** *current floor*.) When switching, the elevator should service all requests found in the queue corresponding to the new service mode. Therefore, the elevator may enter **transition mode** upon switching. In **transition mode**, the elevator moves in the direction opposite of the service mode, and only stops at the last request in queue. So, if the elevator switches to the *UP* service mode, it will move to the LOWEST request in the *up*-queue before stopping. Figure 1 gives an example showcasing this behavior.

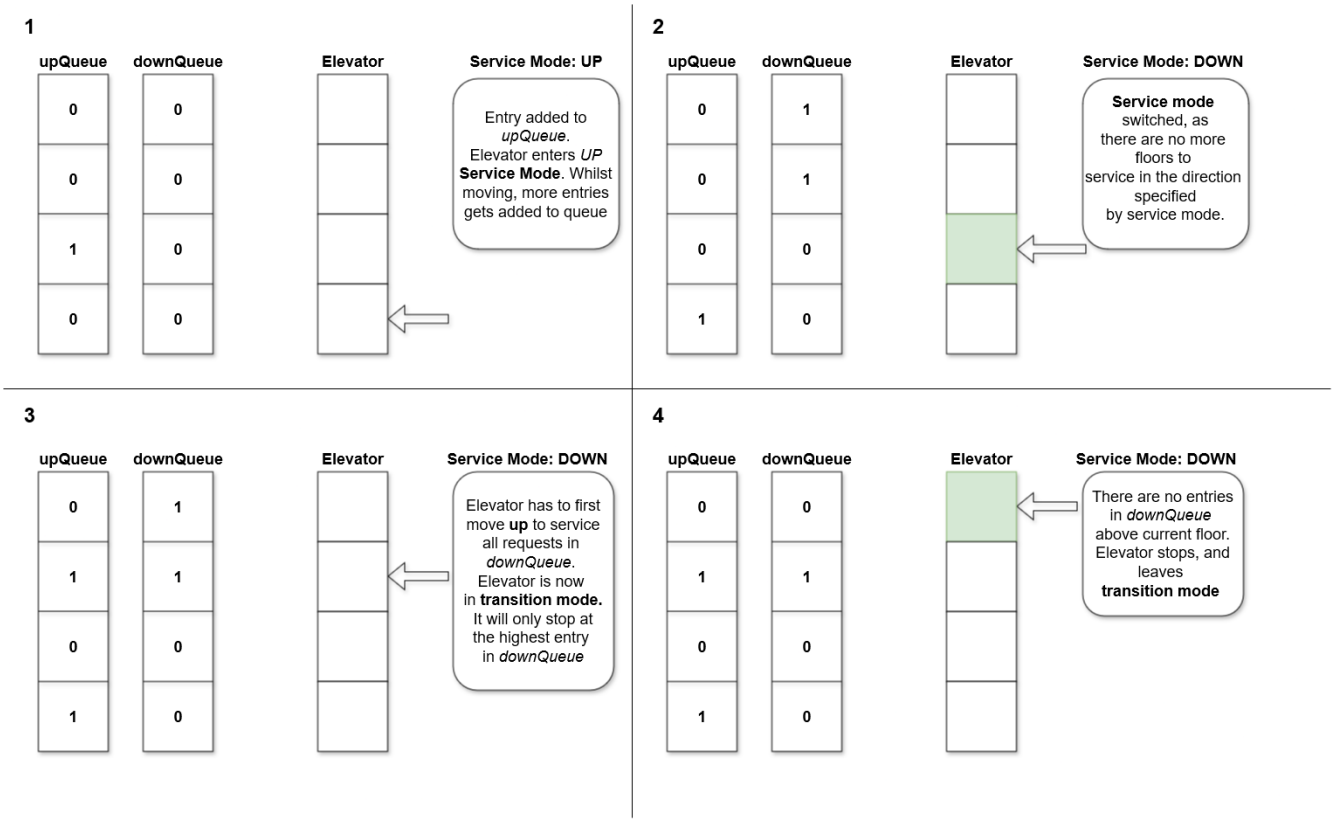

Figure 1: A demonstration of transition mode. Queue entries appear in the example to showcase desired behavior. *Up*-queue requests are ignored in *DOWN* **service mode**.

2.2 Module design

2.2.1 Input Handler

The input handler checks the state of all buttons and executes their associated actions.

It contains one public function, `processInput`. When called, it first checks the stop button state. If pressed, the queue is cleared and -1 is returned. No other input is processed, as required by the specifications. If stop is not pressed, all other button states are checked. If any is found to be pressed, it calls the helping function `createQueueEntry`, adding the request to queue.

The `createQueueEntry` helping function works as follows:

Whenever a `HALL_UP` button is pressed, the request is added to the *up*-queue at that specific floor. The same happens for `HALL_DOWN` buttons, this request is always added to the *down*-queue. Whenever a `CAB` button is pressed, the request gets placed in the *up*-queue if the floor is above the current floor and in the *down*-queue if the floor is below the current floor. If the `CAB` button is pressed for the current floor, then it gets added to *up*-queue if it is in the `DOWN` service mode and vice versa.

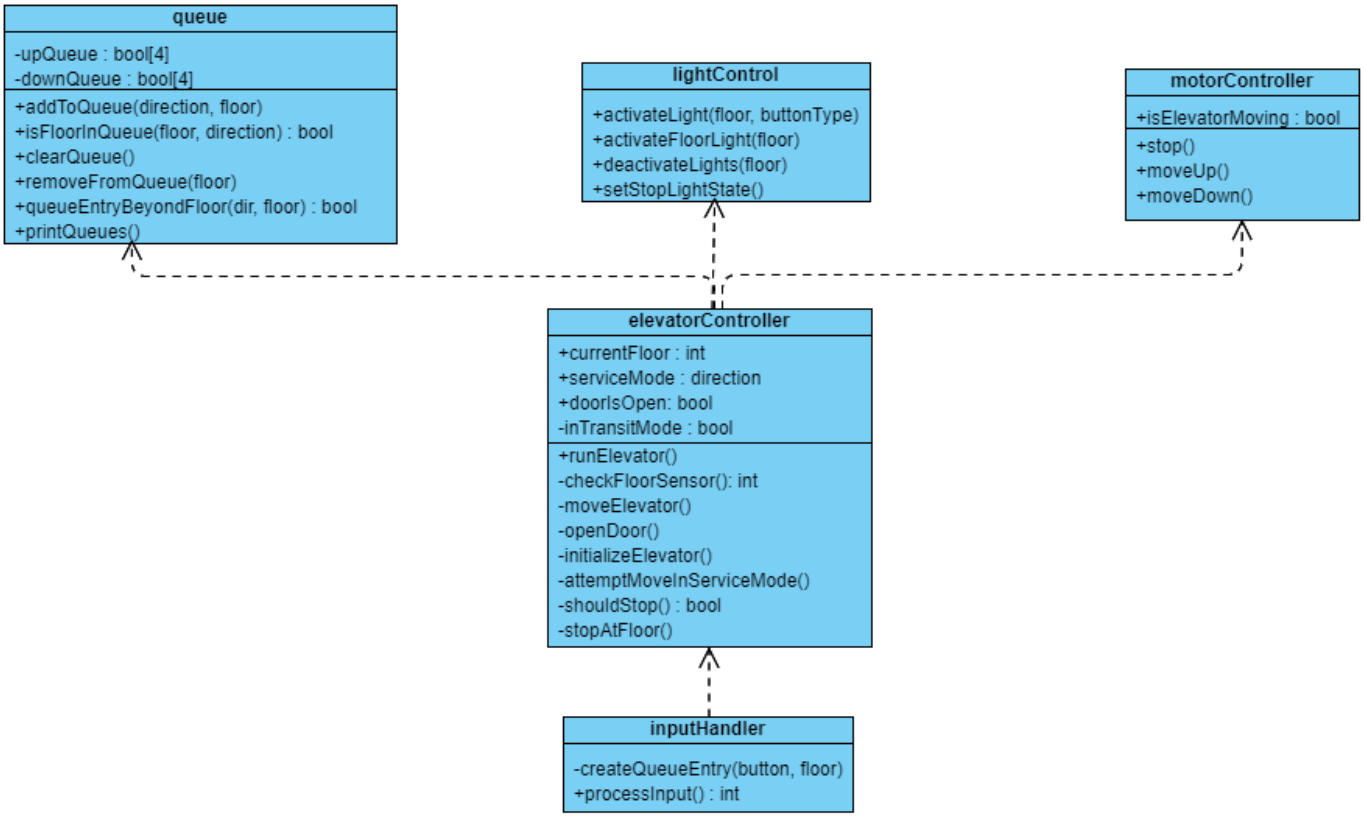

Figure 2: Class Diagram with module structure.

2.2.2 Elevator Controller

The elevator controller contains the main control program of the elevator, `runElevator`, which operates the elevator when called. This is the only function declared in the header. The elevator controller holds the flag `isDoorOpen`, which is asserted while the door is open. It also contains the externally available variables `currentFloor` and `serviceMode`, both of which behaves as described in subsection 2.1.

`runElevator` calls `initializeElevator` before starts a continous loop. At the beginning of the loop, buttons states are checked with `processInput`. If this returns -1, a stop has occured. All button lights except stop are extinguished, and the doors are opened if the elevator is at a floor. The loop then jumps back to the top. If the stop button was not pressed, the stop light is extinguished and `checkFloorSensor` is called. It is important that `checkFloorSensor` only gets called once in a loop. If the `currentFloor` variable gets updated in the middle of looping, it could lead to undefined behavior. If the elevator is at a floor, `shouldStop` gets called. If this returns true, finally `stopAtFloor` is called. At the end of the loop, `isElevatorMoving` is checked, and if negated, `moveElevator` gets called.

`initializeElevator` calls `elevio_init`, makes sure that all lights are turned off then moves the elevator down until it is at a valid floor before exiting.

`checkFloorSensor` takes no arguments and returns the floor sensor state. If the elevator is at a floor, `currentFloor` is updated and the floor light indicator is moved with `activateFloorLight`.

`shouldStop` is a function that returns a boolean value. It returns `True` if the current floor is in the queue, unless the elevator has just flipped service mode and needs to ignore the current floor in queue to first service a floor beyond. The function calls `isFloorInQueue` and `queueEntryBeyondFloor`.

`stopAtFloor` takes no arguments and halts elevator movement with `stop`. It then calls `openDoor`, removes the current floor from both queues and turns off all light corresponding to that floor. `removeFromQueue` and `deactivateLight` is used for this.

`openDoor` takes no arguments and opens the door of the elevator. It then sets the `isDoorOpen` flag to `True` and waits for 3 seconds. If `STOP` is pressed, the close door timer will reset until `STOP` is no longer pressed. The door will close again once the timer hits 0 and no obstruction is detected. Whilst doors are open, `processInput` is continously called, so that the buttons are still responsive.

`moveElevator` is a function that takes no arguments and tries to move the elevator. It first calls `attemptMoveInServiceMode`. If this does not move the elevator, it changes the service mode. It then checks if there is any floors it has to service in the new queue, beyond the current one, by calling `queueEntryBeyondFloor`. If this is true, the elevator enters **transition mode** and the flag **inTransitionMode** is asserted. For service mode *DOWN*, `moveUp` is called, and for *UP* `moveDown` is called. (To get a better understanding of this, see Figure 1.)

`attemptMoveInServiceMode` is a function that takes no arguments and checks if the elevator needs to continue moving in the current `serviceMode` direction. It iterates through the queue and if there is a floor further in the current direction it will call either `moveUp` or `moveDown`.

2.2.3 Queue

Queue holds two queues called `upQueue` and `downQueue`. Both are arrays of four boolean values, where index 0 corresponds to floor 0 and so forth.

Queue also defines an enum `Direction` that maps -1 to `DOWN` and 1 to `UP`.

Queue has several functions that aid in managing the queue;

`addToQueue` takes a direction and a floor, then adds that floor to the queue of the corresponding direction.

`isFloorInQueue` takes a floor and a direction and returns `True` if the floor is in the queue corresponding to the direction and `False` if it is not in the corresponding queue.

`removeFromQueue` takes an integer *floor* and removes that floor from both queues.

`clearQueue` takes no arguments and clears the entire queue.

`queueEntryBeyondFloor` takes a direction *dir* and an integer *floor* and returns `True` if there is a floor "beyond" the specified *floor*. If *dir* is `DOWN` it checks `downQueue` for entries above *floor*. If direction is `UP` it checks `upQueue` for entries below *floor*.

`printQueues` is a debugging function and takes no inputs and just prints out the current state of both queues.

2.2.4 Light Controller

The light Controller is responsible for enabling and disabling the different lights.

`activateLight` takes a floor and a button and lights up the specified button for the corresponding floor.

`activateFloorLight` takes a floor and updates the floor indicator light.

`deactivateLight` takes a *floor* number and turns off all button lights corresponding to that floor.

`setStoplightState` takes a number *states* and turns the stop light off if it is 0 and on if *state* is not 0.

2.2.5 Motor Controller

The motor controller controls the elevator motor and determines if the elevator should move upwards, downwards, or stop.

The functions `stop`, `moveUp`, and `moveDown` are the functions it uses to do this.

Motor Controller is also responsible for updating the `isElevatorMoving` flag, which is `True` if the elevator is moving and `False` if not.

3 Testing

In a project like this it is necessary to test the system on different levels of abstraction. After implementation of the initial design, it was necessary to test the different functionalities. These tests were split up into module testing and system testing. These will be described in detail in sections 3.1 and 3.2 respectively.

3.1 Module testing

3.1.1 Queue

Testing the queue class was mainly done by using the function `printQueues`. As mentioned in section 2.2.3, this function prints out the current state of the two queues. With this function the testing of this class boiled down to calling functions on the queues, and subsequently using `printQueues` to check what effect the function call had on the queues.

3.1.2 Light Controller and Motor Controller

Testing the light controller class and motor controller class followed the same concept. There was no extra functions needed, as the only functionality to test was whether the lights turned on and off at the correct function calls, and if the elevator moved the correct way either in the lab or simulation.

3.1.3 Input Handler

As described in section 2.2.1, the Input Handler is supposed to manage all input from and do the correct logic to be able to add an input to the queue. Testing this was done by pushing buttons on the operational box and using the `printQueues` function. Testing Input Handler included pushing one button and checking if the correct button lit up, and if the request was put correctly into the queue. The class was then stress tested by pushing many buttons very fast, and checking the queues.

3.1.4 Elevator Controller

The major part of testing the Elevator Controller is testing that the different modules operate well together. This will be described in section 3.2. The functions defined in Elevator Controller were tested by calling the functions. Most of them impose physical, observable changes. Therefore, no test functions were needed.

3.2 System testing

The complete system was tested by placing several `printf` functions to confirm completion of certain stages. This helped better an understanding of the flow of control. These stages include initialization, adding a request to queue, stop button pressed, door opening or closing, and change in motor movement. The system behavior was also compared to the requirements [1], to verify that the elevator operated as intended.

4 Reflections

4.1 The V-model

In accordance with the V-model, the general system architecture was designed before the modules. The concept of designing the overall architecture first worked well for this project. We begun with the state diagram, to get an overall idea of how the system was going to work. Then the class diagram was made to describe the different classes or modules that we imagined was needed. Lastly the sequence diagram was made to visualize the flow of information and communication in the system. This will be discussed more in the section 4.2.

While performing the module testing, some of the modules were not behaving as expected. Due to this, there was some deviation from the V model, as it does not describe any approach for improving a design in response of a testing. The modules were tested, then improved, and tested again. This is illustrated in Figure 3 as a V-model more accurate to the design and implementation process of this specific project.

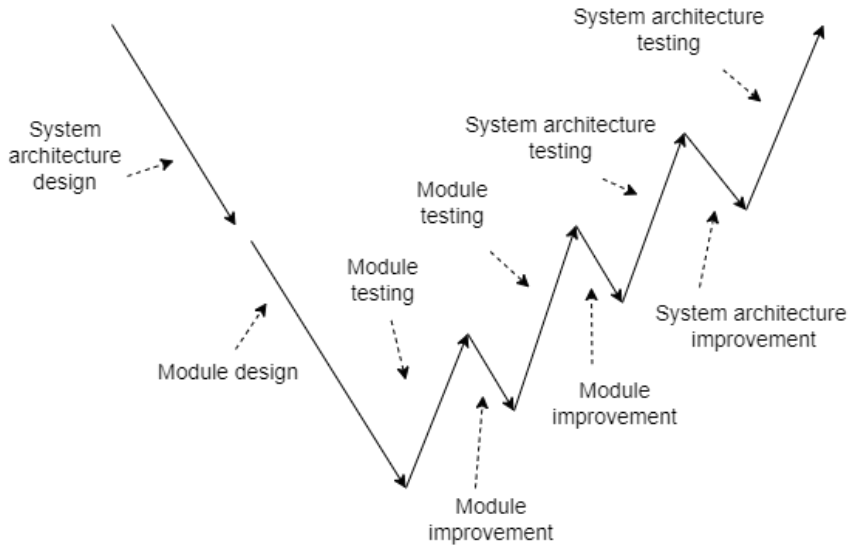

Figure 3: V-model more accurately describing the process in this project.

4.2 Usage of UML diagrams

As briefly described in section 4.1, the UML diagrams were used in the beginning phase of the design process to be able to get a better idea of how the system was going to function [2].

The state diagram in Figure 4, gave a good overview of the general functionalities necessary for elevator function. Thinking through the states of the elevator and figuring out the different ways it could move between states gave better insight to the desired behaviour of the system. This diagram was also a good starting point when planning which modules were needed for the system. Using the state diagram was useful in the sense that the early planning of the system did not just consist of talking, but making a visual representation of out idea that was easier to work

from than our own memory. There was also an attempt to create a preliminary sequence diagram at this stage, but it was not completed because it was not deemed necessary.

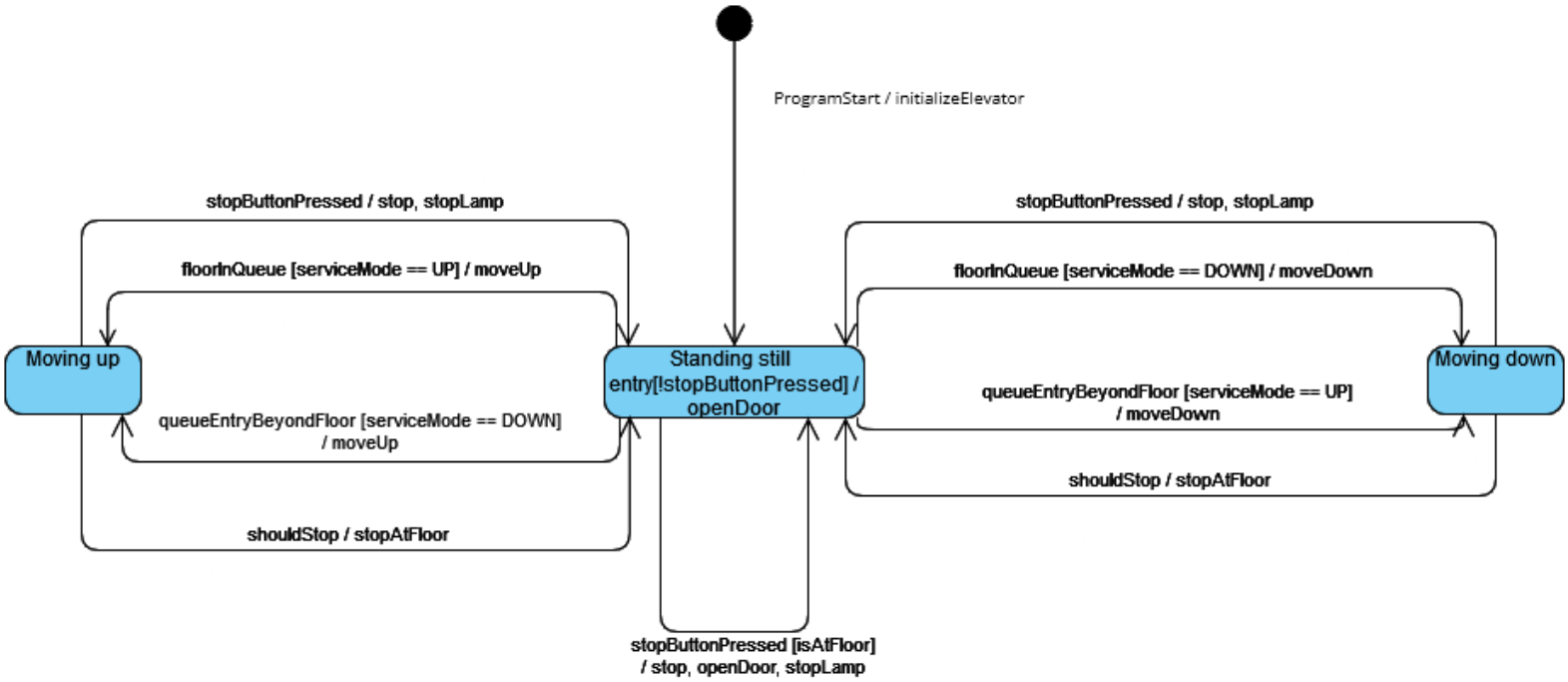

Figure 4: State diagram for the elevator.

Further on in the process, the class diagram was made, found in Figure 2. At this stage there was a clear vision of the general operations of the elevator, and module design was started. The class diagram served the same purpose for the module design as the state diagram did for the architecture design, as well as confirming whether the planned modules were working as intended. Sorting out thoughts and writing them down in an organized way made it easier to imagine which properties each modules should have. Having a class diagram made the implementation of the modules very easy, as the member functions and variable were already defined. As mentioned in the section 4.1, the testing process showed that it was necessary to modify some of the modules. Due to this there was some changes made to the class diagram throughout the testing phase, see Figure 5 for the first class diagram. Nevertheless, having a first draft of the class diagram was a lot easier to work with compared to working from simply thoughts alone. This also made cooperation easy, as different people could implement different modules in parallel.

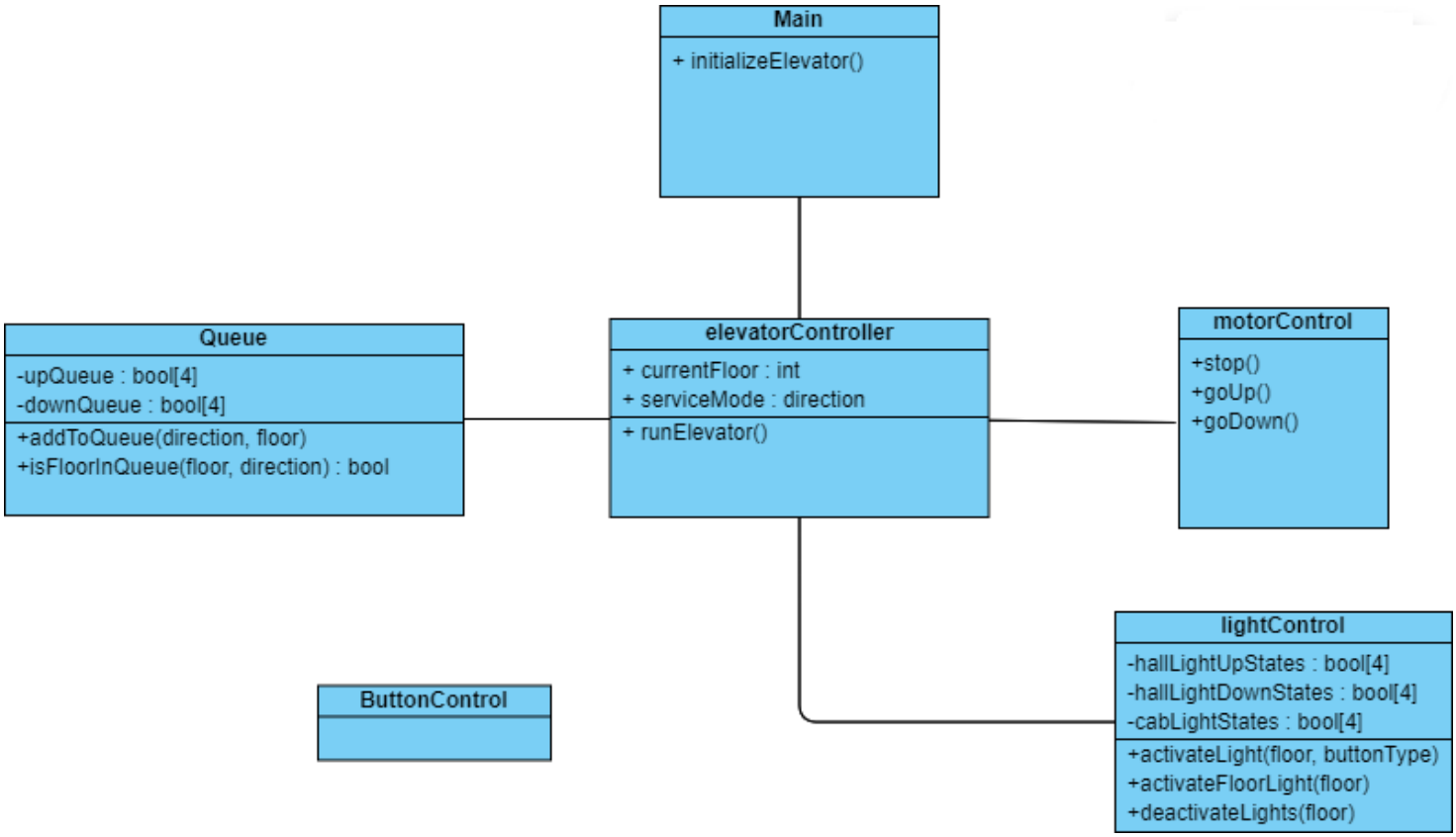

Figure 5: First edition of the class diagram for the elevator. This edition was since improved upon, as shown by the now nonexistent buttonControl class.

The sequence diagram was designed even later in the design process, Figure 6. This diagram did not make a huge difference in the design process. After designing both the state diagram and the class diagram, we had a pretty good idea of the communication flow in the system, and therefore the sequence diagram was unnecessary for the design before this point, but it is an organized way of visually describing the flow of communication in the elevator.

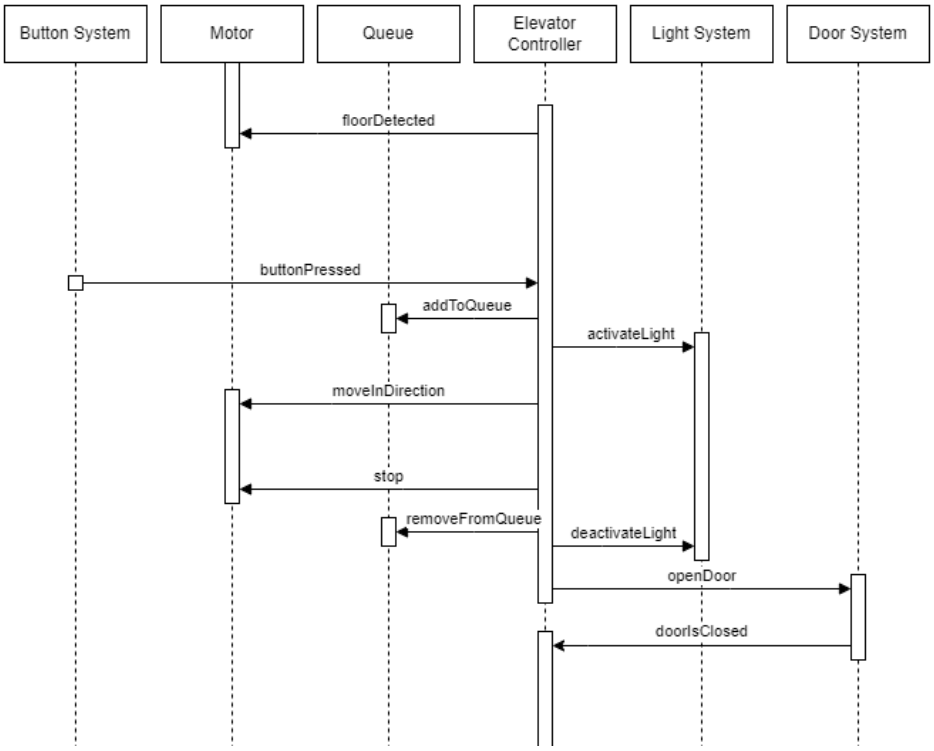

Figure 6: Sequence diagram for initialization and a single button press.

4.3 Areas of improvement

4.3.1 Test phase

As described in section 4.1, the module tests revealed some sections of the implementation that could use some improvement. This lead to several cycles of trial and fixing. Some problems that we bumped into were absence of functionality for central operations like `shouldStop` and `attemptMoveInServiceMode`. The functionality of the `Elevator Controller` should have been better documented before implementation, as we encountered multiple unforeseen issues that warranted restructuring on both the modular and architectural level.

The absence of the `shouldStop` was shown as an inability to check whether the elevator should stop at a floor. Furthermore, the first attempt at implementing this logic without its own function was very disorganized and made the code difficult to follow. This should have been planned beforehand.

The implementation of `attemptMoveInServiceMode` was a bit more complicated than `shouldStop`, as described in section 2.2.2. It also made it necessary do design and implement the function `queueEntryBeyondFloor`. This ended up being a part of a larger revision of the control logic for the elevator.

As seen in Figure 5, there were a lot of changes done after the initial idea of the elevator was formed. When the time came for testing the modules, the system was more developed than the first edition of the class diagram, but it shows how there has been several rounds of trying one implementation, and then changing it in the favour of another version. Even though the first draft for the class diagram ended up being changed, it helped a lot to have a specific idea to work with in the earlier stages of the project.

4.3.2 Code Quality

Some functions should have names that better explain what they do. This is especially true for `queueEntryBeyondFloor`, a quite cryptic name. Some functionality could also have been moved to avoid code repetition, where `processInput` should have been responsible for enabling the stop light and disabling all other button lights.

4.3.3 Finished product

The finished product passed all but one FAT requirement, which reads:

S6: Husker heisen hvor den er ved nødstopp mellom etasjer (dvs. kreves ikke ny initialisering)?

If the elevator has stopped between floors, and gets a `CAB` request to the *current floor*, there is a 50% chance that the elevator moves in the wrong direction, without stopping. This happens because in the idle state, the elevator continuously swaps back and forth between the *UP* and *DOWN* service modes. As defined in `createQueueEntry`, a floor request to *current floor* is added in the opposite direction of the current service mode. So there is no way to predict which queue the request ends up in. This could be fixed by remembering the last service mode from before the elevator stopped.

5 Conclusion

The elevator lab has been a valuable lesson on the great utility UML offers for the design and implementation of a system. Our finished product was not perfect, but passed all but one FAT requirement. UML was invaluable in sharing ideas, and made cooperation with programming much easier. The V-model, however, was found to be too a rigid model for development, as there was no way to make adjustments to the design once the testing phase had commenced. All in all the project went well, there was some deviation from the initial plan for design and implementation, but the finished product was a (mostly) well functional elevator.

References

[1] T. N. Terje Haugland Jacobsson. "Lab 2 heisprosjektet." (2024), [Online]. Available: `https://github.com/ITK-TTK4235/lab_2/releases/tag/0.1.0`.

[2] M. Fowler, *UML distilled : a brief guide to the standard object modeling language*. Pearson Education, 2004.